

07-380 Notes
Pre-reading: Classical Planning

Carnegie Mellon University
School of Computer Science

Contents

1	Logic, Search, and “Classical” Planning	2
1.1	Logical Challenges: Time, Fluents, and the Frame Problem	2
1.2	What Planning Adds to Search	4
1.3	Automated Planning and Classical Planning	6
2	States, Goals, and Actions	7
2.1	States as Sets of Facts	8
2.2	The Closed World Assumption	8
2.3	Goals	9
2.4	Actions	10
2.5	Applicability and Progression	10
2.6	Example: A Plan for Blocks World	11
2.7	All the Notation, in One Place	12
3	Domains, Problems, and PDDL	12
3.1	Action Schemas	12
3.2	Grounding	13
3.3	The Planning Problem	13
3.4	PDDL	14
4	Planning as State-Space Search	17
4.1	The State-Transition Graph	17
4.2	Forward Search	17
4.3	Example: Searching the Blocks World	18
4.4	The Branching Factor Problem	18
4.5	Time and Space Complexity	18
A	PDDL in Detail	20
A.1	S-Expressions Without Lisp	20
A.2	A Complete Annotated Domain File	20
A.3	The Problem File, and What Goes Wrong	22
A.4	A Second Example: Air Cargo	22
B	The Wider Map of Automated Planning	24

1 Logic, Search, and “Classical” Planning

“What is planning? How does it relate to logic? We already covered search algorithms; how is planning different? Why are we calling it ‘classical’ planning?”

Let’s start with a definition of planning:

Planning is the problem of choosing a sequence of decisions that drives a system from a given initial configuration into a configuration satisfying a stated objective, given a model of how each decision changes the configuration.

Note that a *configuration* can be a physical arrangement, such as the location of objects in a warehouse or the pose of a robot arm, or it can be more abstract, such as the set of theorems proved so far. Similarly, a *decision* can be a physical action like loading a truck or rotating a robotic joint, but it can also be applying an inference rule to move a proof to the next step.

“Wait, this still sounds a lot like search. And, can’t we just solve this a few propositional logic sentences?”

Well, yes, we can apply both search and propositional logic to automated planning problems. But generic state-space search and propositional logic both hit (metaphorical) walls that are better scaled with classical planning. We’ll discuss more in the following sections.

1.1 Logical Challenges: Time, Fluents, and the Frame Problem

Picture a logical agent that has to act. For example, in Wumpus World, a logical agent must represent not only what is true but how the world changes when it moves, shoots, or grabs. How do we represent using an arrow in propositional logic? Perhaps:

$$HaveArrow \wedge ShootArrow \Rightarrow \neg HaveArrow$$

But this can’t work. $A \wedge B \Rightarrow \neg A$ doesn’t make sense. We need a different way to deal with the fact that we had an arrow and now we don’t. Facts whose truth value changes over time are called **fluents**. Propositional logic has no notion of time at all, i.e., a fluent is not a new kind of sentence. So, the only way to express a fluent in propositional logic is to include a different symbol for that fluent at every time step. $At_{5,5,0}$ represents the agent at position (5,5) at time 0 and $At_{5,4,1}$ says it is at (5,4) at time 1. Actions get the same treatment, so $Shoot_3$ says the agent shoots at time 3. If a plan runs to T steps, the vocabulary is multiplied by T and the number of possible models scales by 2^T .

But the number of symbols isn’t the only problem; writing the axioms to deal with change is challenging. The obvious move is to say what each action changes, so shooting at time t gives $Shoot_t \Rightarrow \neg HaveArrow_{t+1}$, and moving south from (5,5) gives $At_{5,5,t} \wedge South_t \Rightarrow At_{5,4,t+1}$. That is not enough because of how a knowledge base works: *it never retracts anything*. Adding $At_{5,4,1}$ does not remove $At_{5,5,0}$, which is fine, since those are different symbols and

both are true. The trouble is the opposite one. Nothing in the axioms says that the agent still has its arrow at time 1 if it did not shoot at time 0, so $HaveArrow_1$ is undetermined, and the knowledge base entails nothing about it. Everything the agent knows evaporates the instant the clock ticks, unless some axiom carries it forward.

So we have to write the axioms that carry facts forward. The clean way is a **successor-state axiom**: one biconditional per fluent per time step, saying exactly when that fluent holds at $t + 1$, in terms of what happened at t . For the arrow,

$$HaveArrow_{t+1} \Leftrightarrow (HaveArrow_t \wedge \neg Shoot_t),$$

read as: you have an arrow next step exactly when you had one and did not shoot. The general shape, writing $ADD(a)$ for the facts action a makes true and $DEL(a)$ for the facts it makes false, is

$$f_{t+1} \Leftrightarrow \underbrace{\left(\bigvee_{a: f \in ADD(a)} a_t \right)}_{\text{something made it true}} \vee \underbrace{\left(f_t \wedge \bigwedge_{a: f \in DEL(a)} \neg a_t \right)}_{\text{it was true and nothing broke it}} .$$

All of this is correct, and at any interesting scale it is also unusable. Two separate things go wrong.

- **Size.** One axiom per fluent per time step. The Wumpus world already needs a symbol per square per predicate, so multiply that by the horizon and the encoding is enormous before the agent has taken a single step.
- **Maintenance.** Each axiom has to name *every* action that could possibly affect its fluent. Add one action to the domain and you have to go back through every axiom in the file asking whether it belongs in the disjunction. Nothing warns you when you miss one. The symptom is a plan that is quietly wrong.

This is the **frame problem**. Notice what kind of problem it is: the logic is right, and nobody can maintain it. It is an engineering failure, not a logical one.

What rescues us is that almost everything a successor-state axiom says is that *nothing happened*. The second half of the biconditional, that long conjunction of $\neg a_t$, is pure bookkeeping. So stop writing it down. Make “nothing happened” the default of the representation, and let an action declare only what it *changes*. That one move is what the rest of these notes are built on, and Section 2 makes it precise. It is the difference between a formalism nobody can maintain and one you can write on half a page.

Two things to file away. The trick is a change of *representation*, not of logic; the frame problem does not get solved here so much as handed to somebody else. And the encoding above does not actually die. Fix a horizon k , write out all the axioms, and hand the resulting CNF formula to DPLL: any satisfying model is a k -step plan. The idea was hopeless when it was first tried and became competitive once SAT solvers got good, and it comes back in lecture as one of the three surviving legacies of classical planning.

1.2 What Planning Adds to Search

First, let us disambiguate what we mean by “search”. By search, we mean specifically *state-space search*, rather than the more liberal use of search used to capture all sorts of optimization techniques, such as gradient descent and grid search.

In state-space search, a problem is a start state, a successor function, and a goal test, and a solution is a sequence of actions, which by the definition above is a plan. So planning is not a new problem. It is a new *representation* of the same problem, and everything in these notes comes out of that one change.

Here is the change, on a single example. A robot sits in a room and has to reach a charging dock. Let us describe the situation twice.

Description A. “The world is in state s_{17} . From s_{17} the available actions are a_3 , a_8 and a_{11} , leading to s_4 , s_9 and s_{23} respectively. The goal is s_{62} .”

Description B. “The robot is at $x = 3$, $y = 4$, facing north, carrying nothing, with the battery at 40%, and the door to the corridor is open. The goal is: the robot is at the dock and the battery is charging.”

The two descriptions carry the same information about the world. Given a dictionary translating one into the other, you could recover either from the other. And both are enough to *search*: hand a successor function to breadth-first search, and it will expand s_{17} , or expand the coordinate description, and find a path either way.

What differs is how we represent a state.

In Description B **there is no separate object called “the state”**. The list of attribute values *is* the state. Nobody assigns the number 17 to it, nobody keeps a table matching numbers to configurations, and “which state is this?” is answered by reading off the attributes. The same goes for the blocks world we will use throughout: when we say it is in a certain state, we mean nothing more than that C is on A, A and B are on the table, and the hand is empty. That sentence does not describe the state. It is the state.

This distinction has a standard name, and three levels are usually distinguished. They differ in how much internal structure a state is allowed to have.

Representation	A state is	Reasoning it supports
Atomic	indivisible; a bare label	search over a black-box successor function
Factored	an assignment to state variables	search <i>and</i> inference in propositional logic
Structured	objects and relations among them	search <i>and</i> inference in predicate logic

The two-room vacuum world shows this off nicely, and it is small enough to write out in full. There are two adjacent rooms, Left and Right, and each is either clean or dirty. A robot vacuum sits in one of them. It has three actions: *Suck*, which cleans whichever room it is in, *MoveLeft*, and *MoveRight*. Figure 1 shows one situation, with the robot in the left room, the left room dirty, and the right room clean.

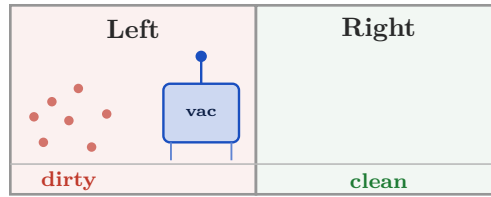


Figure 1: The two-room vacuum world, in the situation $RobotIn = Left$, $DirtLeft = \text{true}$, $DirtRight = \text{false}$.

Count the situations first. Two choices for where the robot is, two for whether Left is dirty, two for whether Right is dirty, so $2 \times 2 \times 2 = 8$ altogether. Now describe them both ways.

Atomically. Number the eight situations $s_1, \dots, s_8$ and record the transitions in a table: from s_1 , *Suck* leads to s_5 ; from s_1 , *MoveRight* leads to s_2 ; and so on for all 8×3 entries. Nothing is wrong with this description, but look at what it costs and what it hides. Somebody has to write the table out by hand, entry by entry. Nothing in it explains why *Suck* from s_1 lands on s_5 and not somewhere else. And adding a second robot gives $2 \times 2 \times 2 \times 2 = 16$ situations and an entirely new table, since none of the old numbers mean what they used to.

Factored. Name three variables:

$$RobotIn \in \{Left, Right\}, \quad DirtLeft \in \{\text{true}, \text{false}\}, \quad DirtRight \in \{\text{true}, \text{false}\}.$$

A situation is an assignment to these three, and there are exactly eight assignments, so nothing has been lost. But now an action can be described instead of tabulated:

Suck: if $RobotIn = Left$ then set $DirtLeft := \text{false}$; if $RobotIn = Right$ then set $DirtRight := \text{false}$.

Three lines, and they cover all eight situations at once, because they never mention which situation we are in. Add a second robot, and you add *one variable*, $Robot2In$; the description of *Suck* does not change at all. In general, n Boolean variables describe 2^n situations, so the description is logarithmic in the size of the state space.

That compression is real, but it is not why we care. We care because the factored description is something an algorithm can *read*. Ask a program whether the goal “both rooms clean” is satisfied. With the atomic description it has to look up a stored list of goal-numbered situations; with the factored one it checks two variables. Or ask whether *MoveLeft* brings us closer to the goal. With the atomic description, the question means nothing. With the factored one, the program can see that *MoveLeft* touches only $RobotIn$, that $RobotIn$ never appears in the goal, and conclude that this action achieves nothing by itself, whatever it may enable later. That last inference, drawn automatically from the text of an action, is the seed of every heuristic in this subject.

So the whole idea is this: keep the assumptions of ordinary state-space search, and change the state representation from atomic to factored. Everything that follows, including every heuristic in lecture, lives off that one change.

1.3 Automated Planning and Classical Planning

Two names get used around here, and they are not synonyms, so it is worth separating them.

Automated planning is the whole field: any setting where a program is given a model of how decisions change configurations and has to come up with a course of action. That covers stochastic outcomes, partial observability, several agents at once, continuous time, numeric resources, and adversaries.

Classical planning is one corner of automated planning, and a demanding one. “Planning under certainty” is a fair informal gloss for the first two rows, but it is not the whole story. The discreteness assumption is doing a lot of work here, and it is the one that gives way in the motion planning lecture. What makes this corner worth starting from is that it is where the guarantees live.

Assumption	Meaning
Deterministic	each action has exactly one outcome, $s' = f(s, a)$
Fully observable	the agent knows the current state exactly
Static	the world changes only when the agent acts
Discrete	finitely many facts, finitely many actions
Single agent	nobody else is acting
Known model	the agent has been given a correct description of every action

With all six in place, the agent can work out the entire plan before doing any of it, because nothing can surprise it along the way. A plan of that kind is called **open-loop**: a fixed sequence $a_1, a_2, \dots, a_n$, settled in advance and then executed without looking. Every one of the assumptions is holding that sentence up. Drop determinism and the third action may not be applicable by the time we get there. Drop full observability and we cannot tell whether it is. Drop static and the world may have moved on without us. Drop single agent and somebody may have undone step two. Lose any of them and the right output is no longer a sequence but a **policy**, a rule saying what to do in whatever situation we find ourselves in.

It is worth being clear about what these assumptions are not. They are not a claim that the world is like this; obviously it is not. They are the price of a guarantee. Inside this corner, a planner can hand back a plan that provably reaches the goal, or a proof that no plan exists, and do it for problems written on half a page. Nowhere outside the corner offers that.

When you get a chance, take a look at the appendix section [B The Wider Map of Automated Planning](#), where we have a great table viewing classical planning from a larger perspective that includes search, MDPs, and RL.

2 States, Goals, and Actions

Now let us make the factored representation precise. Our running example throughout will be **blocks world**: a table, some labelled blocks that can be stacked, and a robot hand that can hold at most one block at a time.



Before anything can be written down, we have to choose the objects and the predicates. Choosing them *is* the modelling work, and everything else follows mechanically.

Objects	Predicates	Meaning
A, B, C, the table	$on(x, y)$	block x is directly on block y
	$onTable(x)$	block x is directly on the table
	$clear(x)$	nothing is on top of x
	$holding(x)$	the hand is holding x
	$handEmpty$	the hand is holding nothing

A predicate applied to specific objects, such as $on(C, A)$, is called a **ground atom**, or in planning simply a **fact**. A fact is just a propositional symbol in the sense of the previous notes: true or false, with no internal structure as far as the logic is concerned. Writing it as $on(C, A)$ is a convenience for us. The reasoner gets no information from the parentheses.

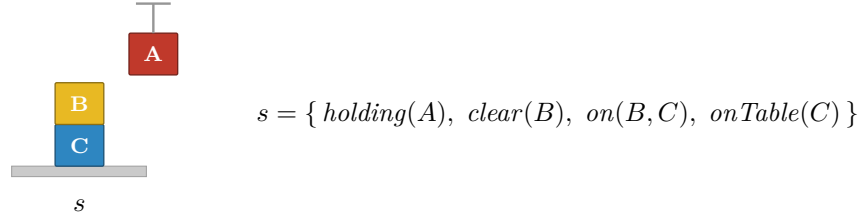
Fact, fluent, atom: one thing, three names. We have now used three words that look like three concepts. Let us settle them here and then stop thinking about it.

A **fact** is a ground atom: $on(C, A)$, $handEmpty$, $clear(B)$. It is a symbol that is true or false. **This is the only kind of object in the language.**

Ground atom is the logician's word, *fact* is the planner's word, and **fluent** is what Section 1.1 called the same thing when the point was that its truth value changes over time. There is no technical difference between them. We will say *fact* from here on, and use *fluent* only where the contrast with something permanently true matters. A **state** is then a set of facts and nothing more, an **action** is a triple of sets of facts and nothing more, and the whole formalism is built out of that one kind of thing.

2.1 States as Sets of Facts

A **state** is a set of facts, understood as their conjunction. Recall from the previous notes that a knowledge base containing $\{\alpha_1, \alpha_2, \alpha_3\}$ is shorthand for $\alpha_1 \wedge \alpha_2 \wedge \alpha_3$; a state is read the same way.



The initial state of our running example is

$$s_0 = \{ \textit{on}(C, A), \textit{onTable}(A), \textit{onTable}(B), \textit{clear}(C), \textit{clear}(B), \textit{handEmpty} \}.$$

Six facts, and that is the whole description. Notice what is missing: nothing here says that A is not clear, that C is not on the table, or that the hand is not holding B, even though all three are true of the picture.

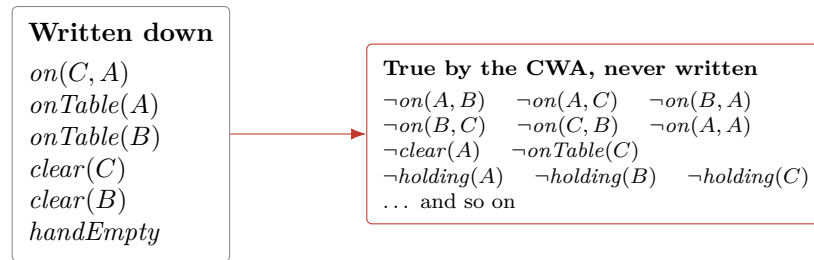
2.2 The Closed World Assumption

We handle those missing facts by convention rather than by writing them out.

Closed world assumption (CWA). Any fact not listed in a state is *false* in that state.

This is a real departure from the previous notes and worth pausing on, because the two settings look alike and behave differently. A knowledge base is an *open* world. If KB entails neither $Rain$ nor $\neg Rain$, the agent does not know whether it is raining and several models are still in play. A state is a *closed* world. It is one specific, completely determined situation, and every fact has a definite truth value in it. In the vocabulary of the previous notes, a state is a **model**, not a knowledge base.

The convention is also what keeps the representation compact. With n objects the predicate *on* alone has n^2 ground instances, of which at most n can be true at once, so the false ones would swamp the description. Our three-block state lists six facts. Spelling out the negative ones would take more than twenty.



One consequence is worth keeping. Because a state is a set, comparing two states is set equality, and applying an action will turn out to be set arithmetic. That is the whole reason this representation is cheap to compute with.

2.3 Goals

A **goal** G is also a set of facts, but it is read differently. A state s is a **goal state** if and only if

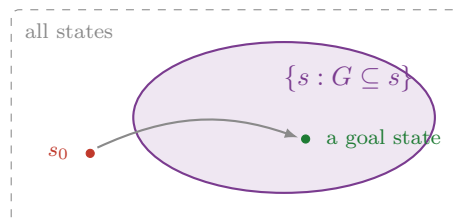
$$G \subseteq s,$$

that is, s contains all the goal facts and may contain any others. Equivalently, reading both as conjunctions, $s \models G$: the goal is *entailed* by the state, in exactly the sense defined in the previous notes.

Watch the asymmetry between the initial state and the goal, because it matters. The initial state has to be a *complete* description, since the CWA will fill in everything we leave out as false. The goal is a *partial* description, naming a set of acceptable states rather than one state. Our goal

$$G = \{ on(A, B), on(B, C) \}$$

says nothing about whether the hand is empty or where C sits, and any state containing those two facts will do.



Since G is a set, it is a *conjunction of subgoals* $g_1 \wedge g_2 \wedge \dots \wedge g_n$. Each has to be made true, in some order we have to discover, and any of them can be wrecked while we work on another. None of that structure is visible in an atomic representation, and every heuristic in lecture will make use of it.

2.4 Actions

An **action** is described by three sets of facts.

- PRE(a), the **precondition**: facts that must all hold for a to be applicable.
- ADD(a), the **add list**: facts that become true after a .
- DEL(a), the **delete list**: facts that stop being true after a .

Anything not named in ADD(a) or DEL(a) is left alone by a . That convention is the **STRIPS assumption**, named after the 1971 planner that introduced it, and it is the fix promised back in Section 1.1: the whole apparatus of successor-state axioms, replaced by a default. Compare the two on one fluent. A successor-state axiom for *handEmpty* would have to list every action in the domain that grasps or releases anything, and would need rewriting the day somebody adds a second gripper. The STRIPS description below says *handEmpty* $\in$ DEL(*pickUp*) and stops. We pay for this: an action can no longer say arbitrary logical things about the next state, only add facts and delete them. It is the same bargain we struck with definite clauses, restricting the language to buy tractable inference.

Here are the four actions of blocks world.

Action		
<i>pickUp</i> (? x)	PRE	<i>clear</i> (? x), <i>onTable</i> (? x), <i>handEmpty</i>
	ADD	<i>holding</i> (? x)
	DEL	<i>clear</i> (? x), <i>onTable</i> (? x), <i>handEmpty</i>
<i>putDown</i> (? x)	PRE	<i>holding</i> (? x)
	ADD	<i>onTable</i> (? x), <i>clear</i> (? x), <i>handEmpty</i>
	DEL	<i>holding</i> (? x)
<i>stack</i> (? x , ? y)	PRE	<i>holding</i> (? x), <i>clear</i> (? y)
	ADD	<i>on</i> (? x , ? y), <i>clear</i> (? x), <i>handEmpty</i>
	DEL	<i>holding</i> (? x), <i>clear</i> (? y)
<i>unstack</i> (? x , ? y)	PRE	<i>on</i> (? x , ? y), <i>clear</i> (? x), <i>handEmpty</i>
	ADD	<i>holding</i> (? x), <i>clear</i> (? y)
	DEL	<i>on</i> (? x , ? y), <i>clear</i> (? x), <i>handEmpty</i>

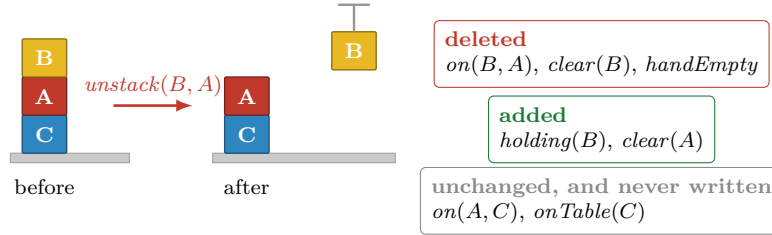
Two details are worth noticing here. All the preconditions are *positive*: basic STRIPS does not let an action require that something be false, and that restriction is what makes checking applicability a plain subset test. Also, the same fact can turn up in the ADD list of one action and the DEL list of another, and *clear*(x) shows up in both lists of *stack* and *unstack* for different x . Nothing is wrong with that. It is precisely the sort of interaction that makes planning hard.

2.5 Applicability and Progression

Action a is **applicable** in state s when $\text{PRE}(a) \subseteq s$. Applying it gives the successor state by **progression**:

$$\text{RESULT}(s, a) = (s \setminus \text{DEL}(a)) \cup \text{ADD}(a).$$

Delete first, then add, so a fact appearing in both lists survives. That one line is all we need to say what does not change, and it is why a state transition costs a couple of set operations instead of an inference.



A **plan** is a sequence $a_1, a_2, \dots, a_n$ such that each a_i is applicable in the state produced by its predecessors, and the final state contains the goal:

$$s_i = \text{RESULT}(s_{i-1}, a_i) \quad \text{for } i = 1, \dots, n, \quad \text{and} \quad G \subseteq s_n.$$

2.6 Example: A Plan for Blocks World

Start from s_0 with goal $G = \{on(A, B), on(B, C)\}$. Here is a plan traced in full, showing only the resulting set at each step; the deleted facts you can strike through in your head as you read.

Action	Resulting state
s_0 —	$on(C, A), onTable(A), onTable(B), clear(C), clear(B), handEmpty$
s_1 $unstack(C, A)$	$holding(C), clear(A), onTable(A), onTable(B), clear(B)$
s_2 $putDown(C)$	$onTable(C), clear(C), handEmpty, clear(A), onTable(A), onTable(B), clear(B)$
s_3 $pickUp(B)$	$holding(B), clear(A), onTable(A), onTable(C), clear(C)$
s_4 $stack(B, C)$	$on(B, C), clear(B), handEmpty, clear(A), onTable(A), onTable(C)$
s_5 $pickUp(A)$	$holding(A), on(B, C), clear(B), onTable(C)$
s_6 $stack(A, B)$	$on(A, B), on(B, C), clear(A), handEmpty, onTable(C)$

Both goal facts hold in s_6 , so $G \subseteq s_6$ and the plan works. It is also optimal, and no shorter plan exists, which you can check by breadth-first search over this small state space.

Step 1 repays a second look. The goal never mentions C at all, and yet the first two actions do nothing but move C out of the way. Any method that reasons only about facts appearing in the goal will miss this entirely. What is in the way is that C sits on A, and A has to be picked up eventually, so $clear(A)$ must be achieved even though nobody asked for it. Detours like this are a large part of why planning problems are hard.

2.7 All the Notation, in One Place

Every symbol we use from here to the end is in the table below. If some piece of notation later looks unfamiliar, it is on this list.

Symbol	Reads as	Is a
p, q	a fact, e.g. $on(C, A)$	ground atom: true or false
F	the set of all facts	set of ground atoms
s, s'	a state	set of facts (the true ones)
I	the initial state	state
G	the goal	set of facts, read as a requirement
a	an action	triple (PRE, ADD, DEL) of sets of facts
A	the set of all ground actions	set of actions
$PRE(a)$	preconditions of a	set of facts
$ADD(a)$	add list of a	set of facts
$DEL(a)$	delete list of a	set of facts
$RESULT(s, a)$	the state after doing a in s	state
$\Pi = (F, A, I, G)$	a planning problem	the whole instance
$\pi = a_1 \dots a_n$	a plan	sequence of actions
Obj	the set of objects	set of constants (A, B, C, ...)
$?x, ?y$	a schema variable	placeholder for an object
$h^*(s)$	the true distance from s to the goal	number of actions

Two things trip people up here. One is that s and G are both sets of facts and look identical on the page; what differs is how we read them. A state is a *complete* description, so anything not in it is false. A goal is a *partial* requirement, so anything not in it is simply unconstrained. The other is the letter O , which we do not use for anything. It appears in the taxonomy table on page 24 as the observation function of a POMDP, which is a different subject altogether. Our objects are Obj .

3 Domains, Problems, and PDDL

3.1 Action Schemas

Writing out $stack(A, B)$, $stack(A, C)$, $stack(B, A)$ and the rest as separate actions would be tedious, and we would have to do it again for every new set of blocks. Instead we write an **action schema** once, with variables standing in for objects:

Action($stack(?x, ?y)$)
PRE: $holding(?x) \wedge clear(?y)$
ADD: $\{ on(?x, ?y), clear(?x), handEmpty \}$
DEL: $\{ holding(?x), clear(?y) \}$

A schema is not itself an action; it is a template. Some sources write a single EFFECT line instead of separate lists, with positive literals meaning additions and negated literals

meaning deletions, so that

$$\text{EFFECT} : on(?x, ?y) \wedge clear(?x) \wedge handEmpty \wedge \neg holding(?x) \wedge \neg clear(?y)$$

means exactly the two lists above. Both notations appear in the literature and in lecture.

3.2 Grounding

Before we can search anything, the schemas have to be instantiated with actual objects. **Grounding** replaces the variables in every schema with every legal tuple of objects, producing the set of concrete **ground actions** the planner will search over. A schema of arity k over a set Obj of objects yields up to $|Obj|^k$ ground actions, so in total

$$|A| = \sum_{\text{schemas } \sigma} |Obj|^{\text{arity}(\sigma)}.$$

For our three blocks: *pickUp* and *putDown* are unary, so three ground actions each, while *stack* and *unstack* are binary, so $3 \times 2 = 6$ each once we throw away the meaningless case $x = y$. That comes to **18 ground actions**, which is a comfortable number to have.

The comfort does not last. Take the standard air cargo domain, with predicates $At(c, a)$, $In(c, p)$ and the schemas $Load(c, p, a)$, $Unload(c, p, a)$ and $Fly(p, from, to)$. Take ten airports, each holding five planes and twenty pieces of cargo, so fifty planes and two hundred packages altogether:

Schema	Ground actions
$Fly(p, from, to)$	$50 \times 10 \times 10 = 5,000$
$Load(c, p, a)$	$200 \times 50 \times 10 = 100,000$
$Unload(c, p, a)$	$200 \times 50 \times 10 = 100,000$
Total	205,000 ground actions

The domain description that produced those numbers fits on half a page. That gap, between a compact description and an enormous ground problem, is the main practical difficulty in planning, and it is the complaint from the logic lecture with a number attached: writing a rule once with variables is a convenience for the human, and every copy comes back the moment a machine has to reason. The schema language buys us the half page. It does not buy us a smaller problem.

3.3 The Planning Problem

It helps to separate what is reusable from what is specific to one situation.

A **planning domain** consists of a set of predicates and a set of action schemas. It describes a kind of world and is reusable across every instance of it.

A **planning problem** (or instance) consists of a domain, a set of objects *Obj*, an initial state *I*, and a goal *G*.

A **solution** is a plan: a sequence of actions applicable from *I* whose final state contains *G*.

A planner that takes any legal domain and problem pair, with no code written for that particular domain, is called **domain-independent**. That is the ambition of the whole field, and it is why the representation matters so much: the planner has nothing to work with except the description we hand it.

We can ask three questions of a planning problem, and they get harder as we go.

1. **Does a plan exist?** Is there a path in the state space from *I* to a state containing *G*? Equivalently: is the problem *satisfiable*?
2. **Find one.** Any valid plan will do. This is called **satisficing** planning.
3. **Find the best one.** Shortest, or of minimum total action cost. This is **optimal** planning, and it is strictly harder.

The word *satisfiable* in the first question is not a loose analogy. The previous notes already remarked that when a planning problem is encoded as a SAT instance, the satisfying model *is* the plan. Lecture will write out that encoding.

3.4 PDDL

The **Planning Domain Definition Language** (PDDL) is the standard notation for writing domains and problems. Essentially every planner reads it, and the International Planning Competition is run on it, which is why a benchmark written in 1998 can still be handed to a planner written last year.

The syntax, if you have never seen Lisp. PDDL files look strange the first time you meet one, because they are written as **s-expressions**, the notation of the Lisp family. The whole idea fits in three rules.

1. **Everything is a parenthesised list.** (*on c a*) is a list of three things, and nesting is just lists inside lists.
2. **The first element of a list says what the list means**, and the rest are its arguments. This is prefix notation. Where we write *on(C, A)*, PDDL writes (*on c a*); where we write *p ∧ q*, PDDL writes (*and p q*); where we write *¬p*, PDDL writes (*not p*). The operator moves to the front and the commas go away.
3. **Keywords beginning with a colon** (*:action*, *:precondition*, *:effect*) are field labels, like keys in a JSON object. Everything from a label to its matching parenthesis is that field's value.

Nothing here is functional, and nothing is evaluated. There is no programming going on. The parentheses are punctuation for a structured data file, so PDDL has more in common with XML or YAML than with a programming language, it just picked a different bracket. Read `(and (holding ?x) (clear ?y))` as the sentence $holding(?x) \wedge clear(?y)$ and you have read all there is.

Reading a domain file. A domain file declares the predicates and the schemas:

```
(define (domain blocksworld)
  (:requirements :strips)
  (:predicates (on ?x ?y) (onTable ?x) (clear ?x) (holding ?x) (handEmpty))
  (:action stack
    :parameters (?x ?y)
    :precondition (and (holding ?x) (clear ?y))
    :effect (and (on ?x ?y) (clear ?x) (handEmpty)
                 (not (holding ?x)) (not (clear ?y)))))
```

Line by line: `(define (domain blocksworld))` opens the file and gives it a name. `(:requirements :strips)` says which dialect of PDDL we are using, and `:strips` is the basic one these notes describe. `(:predicates ...)` lists the predicates with their arity, using `?x` and `?y` as placeholders. After that comes one `(:action ...)` block per schema, each with a `:parameters` list, a `:precondition`, and an `:effect`. Hold that block up against the *stack* row of the action table on page 12 and you will see the same information written twice.

Reading a problem file. A problem file supplies the objects, the initial state and the goal:

```
(define (problem bw-3blocks)
  (:domain blocksworld)
  (:objects a b c)
  (:init (on c a) (onTable a) (onTable b) (clear c) (clear b) (handEmpty))
  (:goal (and (on a b) (on b c))))
```

One feature of the notation is worth flagging, since it is the only place PDDL parts company with these notes. **There is no `:add` and no `:delete`.** The `:effect` field merges both lists: a bare literal such as `(on ?x ?y)` means *add this*, and a negated literal such as `(not (holding ?x))` means *delete this*. So

$$\text{ADD}(a) = \{\text{un-negated literals in :effect}\}, \quad \text{DEL}(a) = \{\text{literals under a not}\}.$$

Notice the asymmetry between the two halves of the problem file. `:init` has to be *complete*, since the closed world assumption reads everything we leave out as false, while `:goal` is

deliberately *partial*. Leaving something out of `:init` is the most common beginner's error, and nothing warns you about it. The planner just cheerfully solves the wrong problem.

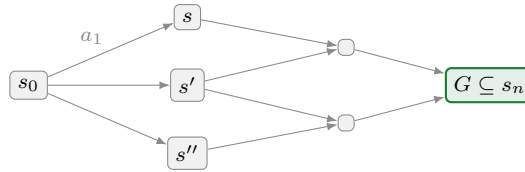
Appendix [A](#) works through a complete domain and problem file annotated line by line, and adds a second worked example. You will want it for the homework rather than for lecture.

4 Planning as State-Space Search

We have the definitions; now we need an algorithm. The most direct one is already familiar.

4.1 The State-Transition Graph

Section 2 hands us a graph for free. Its vertices are the states, and there is an edge from s to $\text{RESULT}(s, a)$ for every action a applicable in s . Planning is then a matter of finding a path from I to any vertex containing G .



Finitely many facts means finitely many states, so the graph is finite. That buys us something immediately: *any* complete graph search algorithm is a complete planning algorithm. Breadth-first search would do the job, and would even return an optimal plan when all actions cost the same.

4.2 Forward Search

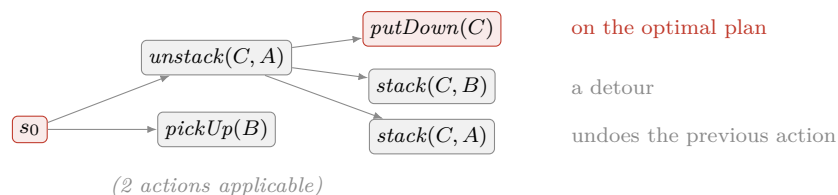
Searching the graph from the initial state towards the goal is called **forward search**, or **progression search**. In outline:

```
FORWARD-SEARCH( $F, A, I, G$ )
  frontier  $\leftarrow$  a queue containing the node  $\langle I, \text{empty plan} \rangle$ 
  while frontier is not empty do
     $\langle s, \pi \rangle \leftarrow \text{POP}(\text{frontier})$ 
    if  $G \subseteq s$  then return  $\pi$ 
    for each  $a \in A$  with  $\text{PRE}(a) \subseteq s$  do
       $\text{PUSH}(\langle \text{RESULT}(s, a), \pi \cdot a \rangle, \text{frontier})$ 
  return failure
```

Every line here is something we have already defined. Applicability is a subset test, expansion is RESULT , and the goal test is another subset test. Which search strategy this turns into depends entirely on how POP chooses: a FIFO queue gives breadth-first search, a stack gives depth-first, and a priority queue ordered by $g + h$ gives A^* . That last one is what we want, and the whole of lecture is about where the h comes from.

4.3 Example: Searching the Blocks World

Our three-block instance is small enough to state the numbers exactly. The reachable state space has **22 states**. Between one and three actions are applicable in any given state, for an average branching factor of 1.91, and the optimal plan takes six steps. Breadth-first search finds it after touching a good fraction of those 22 states.



Two of the three successors of s_1 go nowhere useful, and one of them simply undoes the action we just took. The search knows none of this. It generates every applicable action at every state, whether or not that action could conceivably help. This is the **irrelevant action problem**, and with three blocks it is merely wasteful.

4.4 The Branching Factor Problem

With a realistic domain it is fatal. Go back to the air cargo instance: ten airports, fifty planes, two hundred packages, and a goal of moving all twenty packages at airport A to airport B .

How large is the branching factor? At the low end, every package sits at an airport with no plane in it, so the only applicable actions are flights: $50 \times 9 = 450$. At the high end, all the cargo and all the planes are at one airport, so every cargo-plane pair can be loaded or unloaded and every plane can still fly: $(20 \cdot 10) \cdot 50 + 450 = 10,450$. Averaged over a search, something like $b \approx 2000$ actions are applicable per state.

How deep is the goal? Load twenty packages, fly once, unload twenty times: $d = 20 + 1 + 20 = 41$.

$$O(b^d) = O(2000^{41}).$$

A human can state the plan in one sentence, and uninformed search will not find it before the heat death of the universe. Notice where the difficulty actually lies. It is not so much the size of the state space as the branching factor: the search is drowning in applicable actions, almost none of which help.

4.5 Time and Space Complexity

None of this pessimism is the fault of a lazy algorithm. The decision problems really are hard.

Problem	Complexity
PLANSAT: does a plan exist?	PSPACE-complete
BOUNDED PLANSAT: a plan of length $\leq k$?	NP-complete
... with empty delete lists	a plan is easy; optimising is NP-hard
... with positive preconditions only	PLANSAT in P; optimal plan NP-hard

Keep the third row in mind. Deleting the delete lists makes the problem dramatically easier, since facts can then only accumulate and never have to be achieved twice. That one observation turns out to be the most productive idea in classical planning, and most of lecture is built on it.

The space cost is mild, just as it was for model checking in the previous notes: a depth-first forward search holds one path at a time, so it uses space linear in the plan length while taking exponential time. And just as with SAT, the worst case is a poor guide to practice. Benchmark domains are highly structured, and planners exploit that structure hard enough to solve problems with hundreds of thousands of ground actions.

How they manage that is the subject of lecture. The short version: the factored representation we have gone to such trouble to set up lets a planner *read* the problem description and work out its own heuristic, automatically, knowing nothing about the particular domain. In 07-280, a heuristic was something you invented by staring at a puzzle. From here on, it is something the planner computes for itself.

This appendix is a reference that is related to this pre-reading. Nothing in lecture depends on it, so read when you have time but not necessarily before the planning lecture.

A PDDL in Detail

A.1 S-Expressions Without Lisp

PDDL inherits its punctuation from Lisp and nothing else. The table below translates between the mathematics of these notes and the syntax you will actually type. Read it left to right the first time and right to left after that.

In these notes	In PDDL	Rule
$on(C, A)$	<code>(on c a)</code>	operator first, no commas
$handEmpty$	<code>(handempty)</code>	zero-argument predicates keep their parentheses
$p \wedge q$	<code>(and p q)</code>	prefix, any number of arguments
$\neg p$	<code>(not p)</code>	prefix, one argument
$p \vee q$	<code>(or p q)</code>	needs <code>:disjunctive-preconditions</code>
$\{p, q, r\}$ as a state	<code>(:init p q r)</code>	a field label plus its contents
schema variable $?x$	<code>?x</code>	a leading <code>?</code> marks a variable

Three practical notes. PDDL is **case-insensitive**, so `handEmpty` and `handempty` are the same symbol; we use camel case here for readability, and real files usually stick to lower case. A semicolon starts a **comment** that runs to the end of the line. And the parentheses have to balance exactly, which is where most errors come from. Leave off a close paren at the end of an action block and most planners will complain, confusingly, about the *next* action.

A.2 A Complete Annotated Domain File

Here is the whole blocks world domain, with every line explained.

PDDL	What it says
<code>(define (domain blocksworld)</code>	Open the file; the domain is named <code>blocksworld</code> . A problem file will refer to this name.
<code>(:requirements :strips)</code>	Declare the dialect. <code>:strips</code> means: positive preconditions, conjunctive goals, add and delete lists. A planner that cannot handle a requested feature refuses the file rather than guessing.

PDDL	What it says
<pre>(:predicates (on ?x ?y) (onTable ?x) (clear ?x) (holding ?x) (handEmpty)) (:action pickUp :parameters (?x) :precondition (and (clear ?x) (onTable ?x) (handEmpty)) :effect (and (holding ?x) (not (clear ?x)) (not (onTable ?x)) (not (handEmpty))))</pre>	<p>The vocabulary. This fixes the arity of each predicate, so <code>(on c)</code> would now be rejected. Nothing here is true or false yet; these are just the names we have to work with.</p> <p>Start a schema. <code>:parameters</code> lists the variables that grounding will replace with objects, so a unary schema over three blocks turns into three ground actions.</p> <p>PRE. All three have to hold in the current state for the action to be applicable, which is a subset test and nothing more.</p> <p>ADD and DEL merged. Bare literals go to ADD, so <code>ADD = {holding(?x)}</code>; negated literals go to DEL, so <code>DEL = {clear(?x), onTable(?x), handEmpty}</code>. The final <code>)))</code> closes <code>and</code>, <code>:action</code>, and <code>define</code>.</p>

The remaining three schemas follow the same shape.

<pre>(:action putDown :parameters (?x) :precondition (holding ?x) :effect (and (onTable ?x) (clear ?x) (handEmpty) (not (holding ?x)))) (:action stack :parameters (?x ?y) :precondition (and (holding ?x) (clear ?y)) :effect (and (on ?x ?y) (clear ?x) (handEmpty) (not (holding ?x)) (not (clear ?y)))) (:action unstack :parameters (?x ?y) :precondition (and (on ?x ?y) (clear ?x) (handEmpty)) :effect (and (holding ?x) (clear ?y) (not (on ?x ?y)) (not (clear ?x)) (not (handEmpty))))</pre>	
---	--

Notice that `putDown` has only one condition, so its `:precondition` needs no `and`. Notice too that `stack` adds `clear(?x)` and deletes `clear(?y)`: same predicate, different arguments, no conflict, because after grounding those are two different facts.

A.3 The Problem File, and What Goes Wrong

```
(define (problem bw-3blocks)
  (:domain blocksworld)           ; must match the domain's name
  (:objects a b c)                ; grounding replaces ?x by each of these
  (:init (on c a) (onTable a) (onTable b) (clear c) (clear b) (handEmpty))
  (:goal (and (on a b) (on b c))))
```

Three failure modes account for nearly every problem file that refuses to work.

1. **An incomplete `:init`.** Leave out `(clear b)` and the planner will not complain. The closed world assumption makes it false, B is treated as covered, and you get back a plan for a different world, or a report that none exists. Write out every true fact.
2. **A goal that is accidentally complete.** `:goal` is a partial description, so `(and (on a b) (on b c))` is satisfied by any state containing those two facts. Adding `(not (holding a))` and other tidying is unnecessary, and under `:strips` it is often not even allowed.
3. **Objects that were never declared.** A fact in `:init` that mentions an object missing from `:objects` is an error in most planners and silently ignored in a few.

A.4 A Second Example: Air Cargo

Here is the domain we used in Section 3.2 to count ground actions, written out in full. Compare it with blocks world: the predicates carry more arguments, and that is what makes the ground action count explode.

```
(define (domain aircargo)
  (:requirements :strips)
  (:predicates (at ?x ?a) (in ?c ?p) (cargo ?c) (plane ?p) (airport ?a))
  (:action load
    :parameters (?c ?p ?a)
    :precondition (and (at ?c ?a) (at ?p ?a) (cargo ?c) (plane ?p) (airport ?a))
    :effect (and (in ?c ?p) (not (at ?c ?a))))
  (:action unload
    :parameters (?c ?p ?a)
    :precondition (and (in ?c ?p) (at ?p ?a) (cargo ?c) (plane ?p) (airport ?a))
    :effect (and (at ?c ?a) (not (in ?c ?p))))
  (:action fly
    :parameters (?p ?from ?to)
    :precondition (and (at ?p ?from) (plane ?p) (airport ?from) (airport ?to))
    :effect (and (at ?to ?p) (not (at ?p ?from)))))
```

Two things to take away from this file. The predicates `cargo`, `plane` and `airport` do nothing but record what kind of object something is. They sit in every precondition purely

so that $Load(Plane_1, Cargo_2, \dots)$, with the arguments in the wrong order, comes out inapplicable rather than nonsensical. Basic STRIPS has no type system, so we fake types with predicates; richer PDDL dialects add `:typing` and let you write `:parameters (?c - cargo ?p - plane ?a - airport)` instead. The other thing is the ratio. Seventeen lines here describe, for the instance in Section 3.2, a problem with **205,000 ground actions**. That is the whole reason schemas exist, and the whole reason grounding is expensive.

B The Wider Map of Automated Planning

Recall

Think of the six assumptions as dials. Every setting of them has a name, a literature, and its own characteristic kind of answer. The table on the next page turns the dials one at a time and lists what comes out. This is not a summary of anything you have been taught; it is a reference to come back to, and the only row you are responsible for this week is the shaded one.

Three passes over it are useful.

Pass 1: down the “State representation” column. The first four rows are atomic and the rest are factored, and classical planning sits right on the boundary. It is the first row that keeps the goal-directed sequence-finding of plain search while taking on the structured states of a CSP, which is the claim of Section ?? restated as a table.

Pass 2: across the classical planning row, against its two neighbours. Against the CSP row above it, the state representation is the same and the *solution form* is not: a CSP wants an assignment and we want a sequence. That is where all the extra difficulty comes from, since a CSP has no notion of an action that undoes an earlier one, and we do. Against the MDP row below it, the demand for a world model is the same and the *action model* is not: theirs is stochastic, ours is a deterministic $s' = f(s, a)$, and that one difference is why our answer can be an open-loop sequence rather than a policy.

Pass 3: down the “World model” column, to find where the guarantees stop. The first six rows are handed a correct model: f , or T and R , is given to them. The last two are not. They learn from samples, and what they return is a policy that works well rather than a plan that is provably right. Classical planning is the last row on the map where we can demand a guaranteed answer.

Where this sits in 07-380. This week is the module called *Reasoning Under Certainty*, and it is three lectures long: logical agents, then this one, then motion planning. What happens to the six assumptions afterwards is worth stating honestly, because the answer is not “we relax them one at a time.”

- **Discreteness goes first, and immediately.** The very next lecture, motion planning, keeps determinism and full observability and drops the assumption that there are finitely many facts and finitely many actions. That is enough to break every algorithm in these notes, and the repair is sampling.
- **Determinism does not come back inside planning at all.** Markov decision processes, value iteration, policy iteration, Q -learning and Monte Carlo tree search were 07-280 and are not revisited here. When uncertainty returns in this course it returns first as a problem of *inference*, not of acting: MAP estimation, Bayesian networks, and finally hidden Markov models and particle filtering, which is the state-estimation half of what a partially observable agent needs.
- **Full observability is never fully relaxed in either course.** POMDPs appear on

the table for completeness and are not covered in 07-280 or in 07-380. What you will get is the state-estimation half, in the HMM and particle filtering lecture, without the policy half that turns a belief into a decision.

- **Acting under uncertainty gets exactly one lecture**, on policy-gradient methods of the PPO family. Note where that lands on the table: not the MDP row in the middle, but the model-free row at the bottom. The course skips the model-based middle entirely, so the contrast you should carry forward is between a guaranteed plan computed from a known model and a good policy learned from samples, with nothing in between.
- **The single-agent assumption** is picked up much later, in the game theory lecture. **Static** and **known model** are never formally relaxed in this course.

The table, then, is a map of a field rather than a syllabus. What it does tell you about the syllabus is that we are about to spend a week at the last point where the model can be trusted completely, and the rest of the semester on methods that never had one to begin with.

Approaches to automated planning and sequential decision making. Rows are ordered by how much is assumed known. The shaded blue row is the subject of these notes.

Paradigm	State repr. / Observability	Action model / Opponent	World model $s' = f(s, a)$ or T , cost/ R given?	Solution form	Starting state	Termination	Optimization criterion
Search <i>BFS, DFS, ID, DL-DFS, UCS, A*</i>	Atomic, fully observable	Deterministic	$s' = f(s, a)$ given cost $c(s, a)$ given	Sequence	Single fixed s_0	Goal state (finite)	Min total path cost
Adversarial <i>Minimax, Alpha-Beta</i>	Atomic, fully observable	Deterministic + Adversary	$s' = g(s, a, \text{opp})$ given Utility $U(s)$ given	Contingent sequence	Single fixed s_0	Terminal state (game over)	Max worst-case utility U
Adversarial <i>Expectimax</i>	Atomic, fully observable	Stochastic opponent + chance nodes	$T(s, a, s')$ given Utility $U(s)$ given	Contingent sequence	Single fixed s_0	Terminal state (game over)	Max $\mathbb{E}[U]$ at terminal
CSP <i>Backtracking, Arc consistency</i>	Factored, fully observable	Constraints only	Constraint functions given (no cost function)	Assignment	Partial / empty assignment	All vars assigned (finite)	Satisfy all constraints
Classical planning <i>PDDL, GraphPlan, h^{FF}, SAT</i>	Factored, fully observable	Deterministic cond. + effects	$s' = f(s, a)$ given cost $c(s, a)$ given	Sequence	Single fixed s_0	Goal state (finite)	Min total action cost
MDP <i>Value / policy iteration</i>	Factored, fully observable	Stochastic, $T(s, a, s')$ known	$T(s, a, s')$ given $R(s, a)$ given	Policy $\pi(s) \rightarrow a$	Any state in $\mathcal{S}$	Absorbing state or ∞ horizon	Max $\mathbb{E}[\sum_t \gamma^t R_t]$
POMDP <i>PBVI, POMCP</i>	Factored, <i>partially</i> observable (belief b)	Stochastic, T + observation model O	$T(s, a, s')$, $R(s, a)$, $O(o s', a)$ given	Policy over beliefs $\pi(b) \rightarrow a$	Any belief b_0	Absorbing state or ∞ horizon	Max $\mathbb{E}[\sum_t \gamma^t R_t]$ over belief sequence
Q-learning <i>Model-free RL</i>	Factored, fully observable	Stochastic, T , R <i>unknown</i>	None explicit: learns $Q(s, a)$ from samples	Policy via $\arg \max_a Q(s, a)$	Any state in $\mathcal{S}$	Potentially ∞ horizon	Max $\mathbb{E}[\sum_t \gamma^t R_t]$, no model used
Policy gradient <i>PPO, REINFORCE</i>	Factored; partial observability possible	Stochastic, T , R <i>unknown</i>	None: no T , R , no value function; learns $\pi(a s; \theta)$	Parametric policy $\pi(a s; \theta)$	Any state in $\mathcal{S}$	Potentially ∞ horizon	Max $\mathbb{E}[\sum_t \gamma^t R_t]$ via ∇_θ directly

Search / Adversarial / CSP
Classical planning
MDP / POMDP
Model-free RL

$\mathcal{S}$: state space, $\mathcal{A}$: action space

$f : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$: state successor function g : as f , but including the opponent's action

$U : \mathcal{S} \rightarrow \mathbb{R}$: utility function $c : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$: cost function

$T : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$: transition function, the probability of $s \xrightarrow{a} s'$

$R : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$: reward function (R_t is the reward gathered at time t) $Q : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$: Q-function

$O : \mathcal{S}' \times \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$: observation function

$\pi : \mathcal{S} \rightarrow \mathcal{A}$: deterministic policy $\pi : \mathcal{S} \rightarrow [0, 1]^n$: probabilistic policy over n feasible actions

