# *A Gentle Introduction to Symbolic Planning, Reasoning, and Formalisms*

*Chris Thierauf*

*January 2026 (version 5)*

When you plan to accomplish a complex goal, you break it down into a series of tasks that you know how to accomplish. For example, to make yourself a sandwich you must first go to the kitchen, then open the fridge, then grab the bread, and so on. How can robots solve the same type of problem? We know that we can hand-write behavior using code, but this does not provide adaptability. For example, our sandwich-making code can't be used to instead bake a cake (even though they use the same core steps of going to the kitchen, retrieving food, and preparing it).

This is fine for warehouse robots, where we know the exact objects they will receive and the exact output they must produce. But as robots move toward the real world, this is much less acceptable. We need robots that are intelligent enough to understand our changing priorities, that can adapt to changes in the world around them, and that can make their own decisions about what to do next. We're describing a form of AI here, and this problem is typically solved by a form of AI called "symbolic planning". With symbolic planning, we define a symbolic model of the world, and then search over possible sequences of actions to find one that satisfies a goal. This model contains facts, actions, and how actions change facts. The logic defines what transitions are possible, and planning is the process of exploring those transitions to discover what sequence accomplishes the desired outcome.

The goal of this document is to provide a human-readable introduction to the field of symbolic planning and first-order logics, which produce the foundation for most of the symbolic AI field. This document focuses on providing concise introductions to the mathematical foundations, and an intuition of how these problems are generally solved and deployed in real-world robot systems.

You should also check out Ghallab's "Automated Planning: Theory and Practice", but it's a long textbook. This is more of a quick intro.

To introduce these concepts, I'll first start with a bit of history to connect Classical AI to other forms of AI, and some conceptual background. I'll then introduce a motivating example to show how these concepts allow a robot to solve a challenging planning problem. Finally, I'll describe the theoretical language and symbols that are typically used in the academic literature, and tie back in to real-world use-cases.

## A Very Brief History of AI

When you think of "AI" or "machine learning", you probably think of deep learning, reinforcement learning, computer vision, neural nets, etc. However, these only represent "contemporary" AI. Prior to this new wave were the "symbolic" or "classical" approaches[1]. These approaches revolve around symbols, logic, and explicit mathematical reasoning.

[1] For the sake of consistency I'll stick with the term "symbolic" since it's more descriptive and accurate, but it's worth being aware of both terms.

Based on this concept, systems were produced that could consume rules and generate novel conclusions, most famously with the creation of LISP (which is still in use today). This allowed the creation of "expert systems": if you have a human expert sit down and write a bunch of basic facts about their area of expertise, you can have an AI that captures that knowledge. It can be queried, and can even generate new conclusions that were not encoded – they might even be unknown to the original expert! In one famous example, the now-defunct Digital Equipment Corporation used one of these expert systems to allow technicians to configure their extremely complicated supercomputers without requiring the assistance of a specialist, saving tens of millions of dollars in half a decade.

This was very exciting: if we can just encode more and more rules, we can produce more and more sophisticated behaviors. Researchers famously believed that they would crack something close to full AGI within the decade[2]. Obviously, this didn't happen. That's because unfortunately, there's a catch: you need a *lot* of rules, and they need to be written by hand. These rules are firm, and firm rules are brittle. Further, each rule introduces the need for slightly more computation, which quickly adds up.

[2] The 1956 Dartmouth Workshop, which gathered prominent researchers in the field, is known for producing the term "artificial intelligence" and for believing that they would make "substantial progress" to full language-capable problem-solving systems within "a few summers", with the completed solution being predicted to arrive within 10 years after that.

Despite that, symbolic methods still have their applications. In very controlled domains, like a warehouse or factory, the rules can be written and assumed to be unchanging. In these domains, symbolic planners are reliable and interpretable, and remain effective at producing long-horizon goal-directed behavior. In less controlled domains, like field robotics, the world is only partially observable and the rules are often changing or underdescribed. However, symbolic methods remain valuable at a higher level of abstraction, where they reason over goals, constraints, and long-horizon outcomes. As a result, allowing symbolic methods to direct machine learning methods has been a powerful tool for extending their capabilities: for example, a symbolic planner can select a sequence of reinforcement learning policies, allowing symbolic planning to cover the long-horizon weaknesses of RL and allowing RL to handle the control policies planners are unable to perform.

This document focuses on the symbolic side of these systems, which remains largely the same independent of how it is (or isn't) integrated with other systems. Symbolic systems will require states, actions, and goals to be represented so that a plan solving them can be found. The concepts that enable this planning are described next.

## Conceptual Background for Symbolic Planning

Let's consider a classic example of logical reasoning:

1. All humans are mortal.
2. Socrates is human.
3. Therefore, Socrates is mortal.

From facts 1 and 2, we can conclude fact 3. If you could convert facts like this into code, you would produce a system capable of making the conclusion to fact 3 on its own. And from that 3rd fact, additional rules can be chained together to allow for further conclusions: all mortals need to eat, therefore Socrates needs to eat, etc.

The connection to robot planning here is that we can reason about anything we have facts for. We can provide facts about actions and the environment, and then logically reason about how actions might impact the environment in a way that solves our problem. Consider a robot-behavior-oriented version of the previous problem:

1. There are 2 rooms, labeled $X$ and $Y$.
2. Calling the "move" behavior brings us from room $X$ to room $Y$.
3. Therefore, to get to room $Y$ starting from $X$, run 'move' from $X$ to $Y$.

With fact 1, we provide basic information about the world. For fact 2, we provide a fact about the behavior the robot is capable of. We can then conclude fact 3, which is about a specific action that can be employed for a specific result. This particular fact is useful if our goal is to be in room $Y$. But unlike this example, planning isn't about deriving new static facts: it's about using static facts to see how actions can influence the environment and work towards complicated goals.

To apply this knowledge towards getting a full plan, then, we have to go several steps further. We can expand this logic with more rules about the world and more rules about how behaviors impact the environment. For example, we might add information about "picking up" resulting in a condition of "holding", and that "holding" while "moving" produces "carrying". As the complexity of the world grows, so does the task of finding a solution. A symbolic planning system aims to find new knowledge about how these predicates can be chained together to accomplish a complex goal. For this reason, we have to perform a long-horizon search across many different combinations of possible actions.

To actually implement this, we'll need to construct some formal representation of the logic being used here. Once it's math, then we can convert it to code.

## Propositional Logics

If you're a programmer, you're probably already familiar with propositional logic: you use it every time you write code, and you probably encountered it if you took a Discrete Math course. Here's a classic example you've probably seen before: in propositional logic, we start with propositions (statements that are true or false): for example, "it is raining" or "the ground is wet". We can then use logical operators like negation, conjunction, disjunction, and implication to describe more complex combinations of factual statements. For example, we might state that $R$ represents if it is raining and $W$ represents if the ground is wet. Then, we can make claims like"$W$ is false, therefore $R$ is false".

We'll go beyond that typical example here, so that we can extend it into the first-order logics we'll use for planning.

In propositional logic, we define a handful of components:

**Variables**. Variables, like in traditional math or programming, are symbols that represent some unspecified value that can be filled in later: $x$, $y$, $a$, $b$, $cat$, $dog$. Different communities use different conventions. In many conventions, variables are left lowercase if they have no fixed value, but uppercase if they do[3]. I'll use that convention here. Sometimes you'll also see variables as a single character: $c$, not $cat$. I'll stick to letting variables be arbitrary lengths for legibility.

[3] For example, $cat$ is just some variable that we're calling 'cat' but $Cat$ is a specific cat.

**Operators**. From propositional logic, we keep the concept of operators. There are some familiar faces here:

- $\Leftrightarrow$, meaning bicondition: $a \Leftrightarrow b$ states that "$a$ if and only if $b$"; stating they are fully equivalent.
- $\neg$, meaning negation. For example: $\neg\top \Leftrightarrow \bot$
- $\wedge$, meaning conjunction ("and"). Ex: $\top \wedge \top \Leftrightarrow \top$ but $\top \wedge \bot \Leftrightarrow \bot$.
- $\vee$, meaning disjunction ("or"). Ex: $\top \wedge \top \Leftrightarrow \top$ and $\top \wedge \bot \Leftrightarrow \top$.
- $\Rightarrow$, meaning implication: $a \Rightarrow b$ states that "if $a$, then $b$".

'$\Rightarrow$' can also be read as "implies" or "it follows that"

## Basic First-Order Logic

First order logic is an extension of propositional logic. Functionally, it's the same: any problem you can solve with a first-order logic can also be solved using a propositional logic. However, they differ significantly in expressiveness and practicality: first-order logics provide powerful tools to simplify the syntax of propositional logics. They provide tools for compact, abstract descriptions that would be too long to practically communicate in propositional form.

First-order logic is often abbreviated "FOL"

With our propositional logics, we can represent facts about what our robot is capable of: for example, "the robot can move from room one to room two", "the robot can move from room one to room three", "the robot can move from room two to room one", and so on. But what we really want to communicate here is

that "the robot can move from any room to any other room". To do this, we need to introduce the two new tools that first order logics provide:

**Predicates**. Predicates get a bit fancier in first-order logics. In the simplest form, predicates represent properties: we might define the predicate $IsBox(x)$, which states that "x is a box". Predicates can also get more complex: we might define the predicate $On(x,y)$, indicating that $x$ is on $y$. We also keep the concept of "true" (which has its own symbol: $\top$) and "false" (which also has a symbol: $\bot$).

As with variables, the math convention is to keep predicates single-character: $B(x,y)$ instead of $IsBox(x)$), but again, we'll stick with the more verbose programming convention here.

**Quantifiers**. We already have the ability to check for the truth of a statement thanks to variables and predicates: that's the foundation for propositional logic that we build on here. Quantifiers are introduced in first order logics to describe the "range" of the elements that we are reasoning over, allowing us to take advantage of set theory. There are two main quantifiers: $\forall$ and $\exists$.

$\forall$ is the "universal quantifier", and it is read as "for all". It states that all elements hold true: for example, we might write

$$\forall x(\text{IsCat}(x) \Rightarrow \text{IsMammal}(x))$$

to state that "for every $x$, if the $x$ is a cat, the $x$ is a mammal". We can also make use of set theory here:

$$\forall x \in \{\text{German Shepard, Black Lab, Beagle}\}(IsDog(x))$$

is true, because everything listed in the set satisfies "IsDog".

$\exists$ is the "existential quantifier", and it is read as "there exists". It states that at least one element holds true. For example, we can see that:

$$\exists x \in \{1,2,3,4,5\}(x > 4)$$

evaluates to true, because there exists at least one value greater than 4 in the set.

There are other quantifiers in other first-order logics not worth covering here: first-order logic is actually a class of logic with many different slight variations, but these are the core features that are fairly universal.

In practice, planners do not generally reason over these abstract variable-based rules directly. Instead, we convert "move from any room to any other room" through a process called "grounding". For example, since our "move" rule takes two rooms, we find every combination of two rooms and produce a predicate for each one. Most planner implementations require this because it produces fully specified states and actions that can be evaluated and simulated. However, this process quickly becomes a bottleneck. However, this grounding process quickly becomes a bottleneck. A small number of abstract actions can expand into thousands or millions of concrete actions as the number of objects grows, leading to the "state explosion problem". Although it remains conceptually feasible, it is also a major driver of computational cost in symbolic planning, and much ongoing research in this space focuses on reducing the need for this computation.

## Foundations of Symbolic Planning

Planning is all about modifying states by finding what series of actions modify the environment until we arrive at the goal state. To do this, there are a handful of core building blocks.

**States**. When we talk about a "state" in planning, we're referring to some representation of the facts about the robot and the world the robot is in. Thanks to first order logic, we have the formal language to describe these states. For example, we might be in the state of $\text{At}(X)$. However, we could also introduce new predicates:

- Let $\text{ObjectAt}(x,y)$, which takes an object and a location, represent an object $x$ being at a location $y$.
- Let $\text{On}(x,y)$, which takes two objects, represent $x$ being on top of $y$.
- Let $\forall x \in \{Kitchen, DiningRoom\}(\text{IsRoom}(x))$
- Let $\forall x \in \{Mug, Table\}(\text{IsObject}(x))$

With this, we could represent a much more complex state. For example, we could state:

$$\text{On}(Mug,Table) \wedge \text{ObjectAt}(Table,Kitchen) \wedge \text{At}(DiningRoom)$$

Also notice that we've never stated outright that the mug is in the kitchen, and yet we can logically infer that it must be! We know that since the mug is on the table, and the table is in the kitchen; therefore the mug must be in the kitchen.

This represents "The mug is on the table in the kitchen and the robot is in the dining room". And, since we're dealing with first-order logics here, would could get substantially more complicated if we really wanted to (the mug is one the table, and it's colored blue, and it's filled with coffee, and coffee is a liquid, and the liquid…).

**Actions**. An action is something the robot can do which will impact the environment in some way. To make this concept useful for planing, actions have preconditions (predicates that must be true before the action can be performed) and effects (predicates that will be true if the action has been succesfully performed). Effects are generally split into 'positive' and 'negative' effects, meaning that they add or remove predicates, respectively.

This is the "STRIPS-Style" of describing planning, referencing a conceptual origin for this style of work.

**Planning Domain**. When we combine a set of actions and possible states, this constructs the 'domain'. If we then add a start state and a goal state, we've produced a planning problem. The solution to this problem will be a series of sequential actions (the plan). Sometimes the plan is called a policy, but that gets confusing when dealing with other types of AI (where a policy has a similar but different meaning) so it's best to avoid this phrasing if possible.

## Commonly used notation

These concepts are commonly described using some specific symbols: States can come in a set as $S$, or be on their own as $s$. One special state is $s_0$, the starting state of a plan. Another is $s_g$, the goal of the plan. The goal is never specified

Sometimes the goal is described as a set, $S_g$, because there might be infinitely many methods of satisfying a goal: there will be parts of the problem that have no impact on if it is successful, and so many solutions might be equally valid

in terms of things that must be done (that would be an action), it is specified in terms of things that must be true for us to consider the plan successful. This distinction is what allows us to produce plans that are adaptable, because we allow these abstract goals to produce the actions (rather than confining them).

The planning problem is usually labeled as $\mathcal{P}$. The plan itself is usually labeled as either $\pi$ or $P$, with each step in the plan being $\pi_0, \pi_1, \pi_2 \ldots$ (or $p_0, p_1$, etc). A planning domain is marked as $\Sigma$, and it contains the states, actions, and effects. The effects are usually described more mathematically as a "transition function taking a": this is just an annoying way of saying " we have a method of taking an action and finding out its effects (like by looking them up)" It's typically represented using lowercase gamma (as in $\gamma(s, a)$).

When this is written more formally, it's common to see papers that state something like:

"The planning domain $\Sigma = \langle S, A, \gamma \rangle$; where each $s \in S$ is a state represented by a first-order logic; each $a \in A$ is an action which can be performed by the agent, and the transition function $\gamma(s, a)$ describes the transition from some state $s$ to $s'$ given $a$. With this, we produce the planning problem $\mathcal{P}$, where $\mathcal{P}(s_0, S_g, \Sigma)$ returns the sequence of actions $\pi$ which satisfies $S_g$ when starting from $s_0$."

That just translates to:

"We have states, actions, and the ability to transition between them. With this, we can perform planning from any start state to a set of goal states."

## Solving Planning Domain Problems

### A simple conceptual example

Let's explore how these conceptual tools allow us to solve complex problems. We'll use a simple domain example: there are three boxes (red, blue, and green) and three rooms (room one, two, and three), and one robot. The robot can carry boxes from one room to another.

To describe this domain, we'll provide the rooms, boxes, and robot as variables. We can describes the rooms: $room(one)$, $room(two)$, $room(three)$; and we can describe the boxes $box(red)$ $box(blue)$, $box(green)$. The robot is its own thing: $robot$. We'll add a predicate: $At$, so we can state things like $At(box(red), (room(one)))$ to mean "the red box is at room one".

So now we can describe a scene:

$$\begin{aligned} &At(\texttt{box(red)}, && room(one)) \ \wedge \\ &At(\texttt{box(blue)}, && room(one)) \ \wedge \\ &At(\texttt{box(green)}, && room(one)) \ \wedge \\ &At(robot, && room(two)) \end{aligned}$$

This means:

The red box is in room one, and

The blue box is in room one, and

The green box is in room one, and

The robot is in room two.

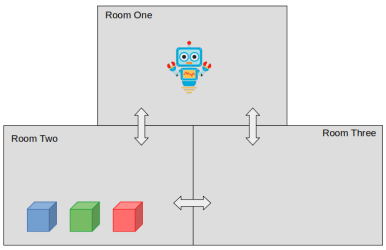



Figure 1: The environment we're describing using formal logics.

Now let's describe our action space. The robot can go from one room to another, and the robot can carry a box from one room to another. We'll call the first action "`goto`" and the second "`carry`".

Because the `goto` action changes the robot's location, it will require we be in one room, will have the effect of no longer being in that room and of being in the goal room. So we have a precondition of $At(robot, room_a)$, a negative effect: $eff^-(At(robot, room_a))$, and a positive effect: $eff^+(At(robot, room_b))$. Notice here that I'm not using a specific room, just the stand-in variables $a$ and $b$ which are the room type: this is shorthand that allows us to re-use the `goto` command across all different combinations of rooms.

We'll do the same thing with the carry action, but it'll be a bit more sophisticated. We require that the robot and the box be in the same room, so the precondition is $At(robot, room_a) \wedge At(box_b, room_a)$. Then, we can add the effects that change both the robot and box location: $eff^-(At(robot, room_a) \wedge At(box_b, room_a))$, and $eff^+(At(robot, room_c) \wedge At(box_b, room_c))$.

Now we can start planning. We'll describe a goal:

$$At(box(red), room(two))$$

Note that it's an underspecified goal: we don't actually say anything about the location for the robot, or the other two boxes! That's OK – we define "solved" as arriving in any state where each stated predicate in the goal is met. So being underspecified will actually work to our advantage: there's more potential states that satisfy this goal.

So we could have all the boxes in room two and the robot in room one, or we could have all the boxes in room two and the robot in room two, or...

Finding the plan that produces this outcome is nontrivial. The only knowledge the system has is the symbolic description of the world, the actions it can perform, and the goal it is trying to satisfy; it does not have any intuitive understanding of which actions are "reasonable" or "useful." From the planner's perspective, every applicable action is a candidate, and the only way to determine whether an action sequence is successful is to apply it and observe the resulting state.

As a result, planning becomes a search problem over possible sequences of actions. Starting from the initial state, we must systematically consider what happens if we try one action, then two actions, then three, and so on, until we encounter a state that satisfies the goal. A breadth-first tree search provides a simple baseline: it explores all plans of length one before any plans of length two, all plans of length two before any plans of length three, and so forth. Assuming a solution exists for this problem within this depth, we know that we will eventually find it.

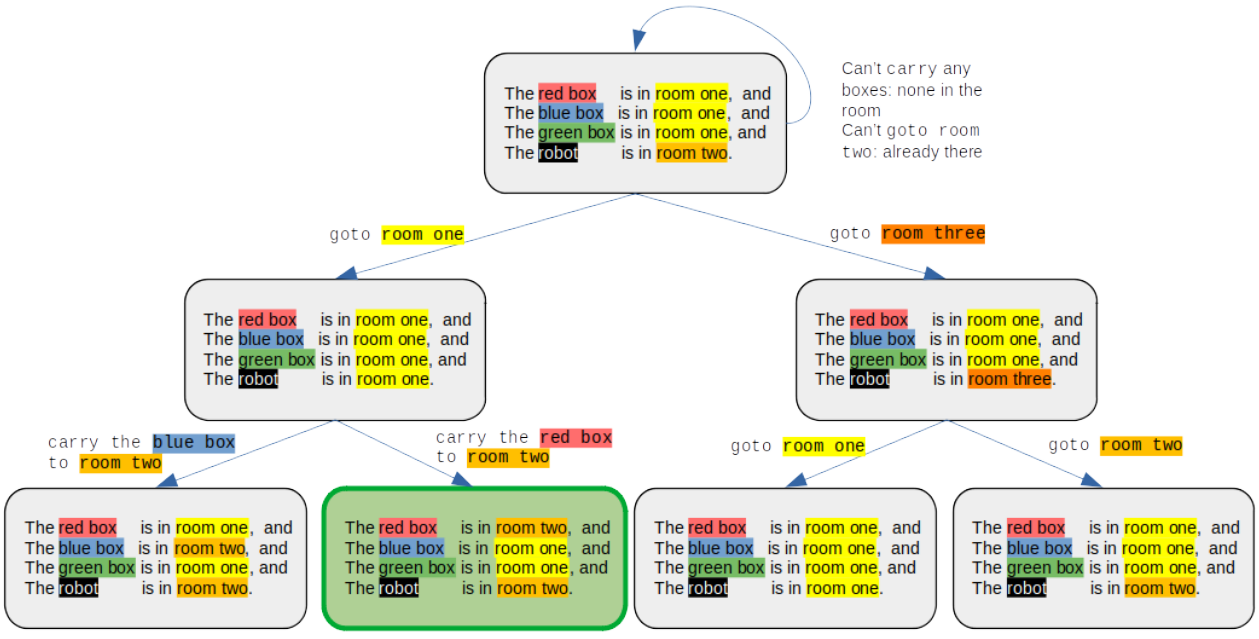
This makes it a reasonable baseline for understanding this problem, but it is computationally impractical for most real problems. Practical planners generally use heuristics to guide search and aggressively prune the search space.

We have a start state, and we have our two actions. Thanks to the first-order logic, these two actions are actually *six* possible actions: `goto` actually represents `goto` room one, `goto` room two, and so on.

This is part of how you get the "state explosion" problem: one of the more major drawbacks of this kind of search is that one simple addition to your problem actually becomes many additions that grow very quickly.

At the first layer of the search, the robot can't `goto` room two (we're already there), and it can't `carry` anything (there's nothing in the room to carry). But it can `goto` room one, and it can `goto` room three. From room three, we again can't carry anything since we don't meet the preconditions: there's nothing in this room. From room one, though, we can try a few things: we can `carry` the red box, we can `carry` the blue box, we can `goto` back to room one.... If we compute each of these, we notice that one of them meets our goal state! We now have our plan: go to room one, and carry the red box to room two.

For a more visual description of this problem solution, see the figure below. Each box is a different state, and each arrow is an action that brings us to that new state. The successful goal is highlighted in green. By generating this tree, we can search through it for the steps that accomplish our goal.



Notice several things here. First, we went off in a branch that wasn't very helpful. That's actually pretty common: we don't know what to try until we compute it, and so a lot of that computation will end up being wasted. A lot of the optimization strategies in this field focus on finding and eliminating those branches before we waste too much time on them.

Second, notice that there is redundancy in this tree. The bottom-right state is the same as our start state! If we were to continue planning, we'd certainly see more cases of this. It's possible for planners like this to produce plans that are sub-optimal, yet valid. For example, a valid plan would be to "go to room three, then go back to room one, then go to room two, then carry the red box to room two".

For this reason, we need some way to measure our plans, so we can find the "best" one. A common strategy is to say that the best plan is the one with the fewest steps. Another common strategy is to assign a cost to each step, and then try to find the plan that costs the least. This cost might be time, battery power, or some other metric you can compute.

**A implementation example**

To keep the implementation accessible here, I'll only describe pseudocode here. A full Python implementation is available at the end of this document.

First, we'll set up some actions. For the sake of our grounded example, these will be their own algorithms: GoTo(r) and Carry(b,r), as Figure 1.

Algorithm 1: Action Definitions.

1: **Let** GoTo$(r)$ **be an action such that:**
2: **Preconditions:** the robot is not already in room $r$
3: **Effect:**
4: remove the robot's current location
5: add At$($robot,$r)$

1: **Let** Carry$(b,r)$ **be an action such that:**
2: **Preconditions:** the robot and box $b$ are in the same room
3: **Effect:**
4: remove the robot's current location
5: remove the box's current location
6: add At$($robot,$r)$
7: add At$(b,r)$

These become our action set $A$. Once we have some states to plan to and from, we'll have our domain. Then, we can transition from our facts to an explicit algorithm (Figure 2). This figure makes breadth-first search slightly more formal than the algorithm we casually described in the previous section, but it does the

same thing.

Algorithm 2: Breadth-First Planning

**Require:** initial state $S_0$, goal facts $G$, actions $A$, maximum depth $D$
**Ensure:** a plan achieving $G$, or failure
1: create an empty list of state–plan pairs
2: add $(S_0,$ empty plan$)$ to the list
3: **while** the list is not empty **do**
4: remove the oldest $(S, \pi)$ from the list
5: **if** every fact in $G$ holds in $S$ **then**
6: **return** $\pi$
7: **end if**
8: **if** the number of actions in $\pi$ equals $D$ **then**
9: continue
10: **end if**
11: **for** each action $a \in A$ **do**
12: **if** $a$ is applicable in $S$ **then**
13: $S' \leftarrow$ result of applying $a$ to $S$
14: $\pi' \leftarrow \pi$ followed by $a$
15: add $(S', \pi')$ to the end of the list
16: **end if**
17: **end for**
18: **end while**
19: **return** failure

Although this procedure is intentionally simple, it captures the essential structure shared by many classical planning systems. The explicit separation between action definitions, goal specifications, and the search procedure allows the same planner to be reused across different domains simply by changing the actions and predicates involved. The planning system doesn't "know" anything about rooms or boxes or what they do, it just "reasons" about them

By "reasoning" here, I mean formal, algorithmic reasoning over symbolic representations, not human-level understanding. Despite this, the resulting behavior can still be complex and goal-directed.

The core structure begins by setting up the initial state (lines 1-2), and maintains a growing list of states to explore, each paired with the sequence of actions that led to it. At each iteration (lines 3-18), we check a previously-unexplored state plus plan, checks to see if it currently satisfies the goal, and if it doesn't, expands it by applying every single valid action to see what might happen next. By expanding the states in this order, we guarantee that the first time we encounter the state it's with the shortest number of actions.

Of course, this implementation isn't very efficient or novel. This is just a simple version to provide you a better intuition. More sophisticated build upon this same structure but add features for pruning redundant states, recognizing repeated configurations, and prioritizing more promising branches to search.

These limitations have motivated the development of better algorithms and

common tools to describe and solve them, which we'll start to explore next. Rather than hard-coding actions and predicates in the planner, the way we've done here, modern systems rely on a formal language that specializes in representing these types of problems. PDDL is the most notable of these. The planner gains facts about the domain from these languages, and then solves for a goal once provided with one.

#### PDDL

So that's the formal background and the most common terminology. With that in mind, there's many parallel projects that attempt to solve elements of this space.

PDDL, the "planning domain definition language", is the most common method to formally encode states, actions, preconditions, and effects described earlier so that general-purpose planners can operate on them. There's a handful of different versions; the most commonly used is 3.1 This language provides us with the ability to make use of existing planners (sometimes called solvers) to return the sequence of actions that solves this problem. The value of PDDL is that it provides a common interface to all of these solvers, so you can compare them or swap them out with each other.

The Planning Wiki describes PDDL in more detail: `https://planning.wiki/guide/whatis/pddl`

Many existing planners are out there; the most common are the ones which extend the "fast-forward" planner[4]. "ff-downward", for example, is one popular implementation. Generally, systems will take these and wrap them for their specific implementation. ROSPlan[5] is one example of this, integrated into ROS: by using PDDL, the planner can be provided an understanding of what behaviors are available. The robot's understanding of the current environment can be converted to predicates using pre-implemented code: for example, some vision processing ROS node that estimates if an object is on another object. When the goal is specified, we've produced a problem that can be solved. The result of running the planner on this domain + problem is a list of actions that we can then call. Of course, this does again require some preimplemented code to handle converting from a list of actions to actually calling code.

[4] `https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/download/1572/1471/`

[5] `https://kcl-planning.github.io/`

Another method is the language "LISP". Its variants and versions are widely used in these circles as well, even though it's *ancient* for a piece of software (60+ years). But there continue to be updates to the language: Clojure is one more modern and popular example.

### Symbolic AI in real life

There are many uses of symbolic AI in real life. Many widely used formal decision-making methods trace important conceptual roots back to symbolic reasoning and planning, and it's helpful to understand how these principles are translated into modern real-life applications.

**Deep Submergence Laboratory / WHOI**

I'll start with some of my work: this forms the core of an architecture I've developed for AUV Sentry at the Deep Submergence Laboratory and National Deep Submergence Facility. I've published more on this elsewhere[6].

The existing architecture, MC, is more of a strictly-defined state machine: you tell it the mission is started, and it's pretty hard-coded to go through the different steps of the mission, in order. It reads from a ".tracks" file, which has commands like "go to" and "change altitude". These steps are executed in order, and generated prior to deployment by hand-typing the high-level steps of a mission (which are then converted into the individual tracks file steps).

In an at-sea trial of DYNOS, symbolic planning provided a handful of important new features. Rather than specifying individual steps of the mission, we provided a high level goal (complete a survey). It sequenced all the necessary behaviors, like descent and the individual tracklines. Upon deployment, Sentry followed the plan that the symbolic planning system produced, which included generating tracklines and waypoints. During the dive, we sent a command that a new goal (the mission being done) should be followed. In response, a new plan (containing only the abort behavior) was followed.

[6] See `https://www.cthierauf.com/publications/teleoreactive-cognitive-sentry-mission-executive`

**Jet Propulsion Laboratory / NASA**

The JPL/NASA robotics researchers have been big proponents of the popular "three-layer architecture" approach[7]. In this approach, we break our architecture into three conceptual chunks: feedback-control that reacts to stimulus in real-time (which is usually not symbolic, it's a real-time controls system), a short-term planning system (which is generally symbolic), and a long-term planning system (the long-term planning system is most often symbolic in deployed systems, particularly where safety, interpretability, or hard constraints matter.).

Most recently of NASA's missions, the Curiosity and Perseverance rovers landed on Mars. Although missions here are more firmly constrained, they are still symbolicly-represented plans: in this case, they make use of a timeline-based representation to make the exact behavior clearer to understand and edit[8]. Note here one of the central benefits of a symbolic approach: grounding in symbols makes communicating and modifying complex behavior much more achievable.

Symbolic decision-making played a central role in enabling Curiosity and Perseverance to plan, inspect, and modify complex mission behavior. Goals are provided to the system to produce a high-level sequences, and those sequences can be communicated to the rover for execution.

[7] See Gat's "*On Three-Layer Architectures*" for more: `https://robotics.usc.edu/~maja/teaching/cs584/papers/tla.pdf`

[8] The system is called *COCPIT*, and NASA has screenshots of it that are easy to find if you google around for it.

**Amazon Robotics / Kiva Systems / Warehouse Robots**

Amazon Robotics is a subsidiary of Amazon: they make the robots that enable Amazon's warehouses to operate with increasing autonomy. Prior to the acquistion that made them "Amazon Robotics", they were Kiva Systems, and they pushed the concept of autonomous warehouses that have now become an almost universal part of consumer delivery. It's become a multi-billion dollar industry[9]: 24 billion as of this year, with projections of nearly doubling in the next 5 years.

Their primary innovation was a class of robots called an "AMR". AMR is an acronym for Autonomous Mobile Robot, which is very broad. It's come to be interpreted as the class of warehouse robots that bring loads of product from point A to B so that the product can be packed and shipped. This dramatically cuts down on their time-to-delivery, which produces reduced costs and faster delivery times.

These systems work because of coordination at scale. By maintaining a fleet of robots that are provided with a retrieval task, one at a time, a large set of products can be shipped to individual homes. To make all this work, there needs to be an impressive amount of internal coordination and long-horizon planning.

Public documents[10] about the Amazon Robotics software architecture describe a cost-optimized planning problem, which is an extremely common extension of the planning problem we've described earlier (the difference being that we assign a cost to each action, and then pick the 'best' plan as the one with the lowest cost instead of the one with the fewest steps).

symbolic planning.

**IBM CPlex, OptaPlanner: Resource Scheduling**

Let's say you run a hospital. You have a known set of rooms and beds. You'll need to figure out how many nurses to have on staff and where they'll need to be allocated. This is very hard to do as the hospital gets bigger. This is where products like OptaPlanner or IBM's CPlex are handy: these use symbolic constraint-based optimization rather than classical action-based planning, but they share the same core idea: explicitly modeling structure and constraints so that a solver can search for valid solutions. We can make this schedule by treating 'work in room *x*' as an action, and then provide as many rooms as need. Then the planning problem becomes to add staff until the plan can be solved. This scales more effectively than doing it by hand: as we add lunch breaks, overtime, and other real-life constraints, the problem becomes harder to solve. But this method allows us to make a computer solve it for us.

[9] `https://www.statista.com/statistics/1094202/global-warehouse-automation-market-size/`

Figure 2: A forklift-based AMR, like this one produced by TUDOR, is capable of autonomously finding pallets on a warehouse floor, and carrying them to their destination. Image from Wikipedia: `https://commons.wikimedia.org/w/index.php?curid=7838942`

[10] Wurman, D'Andrea, and Mountz: *Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses.*

## Appendix A: PDDL construction

Here's an example of PDDL. First we define the domain:

```
(define (domain blocks)
    (:requirements :strips)
    (:predicates
        (on-table ?b - block)
        (on ?b - block ?o - block)
        (clear ?b - block)
        (holding ?b - block)
    )
    (:action pick-up
        :parameters (?b - block)
        :precondition (and (clear ?b) (on-table ?b))
        :effect (and (holding ?b) (not (clear ?b)) (not (on-table ?b)))
    )
    (:action put-down
        :parameters (?b - block)
        :precondition (holding ?b)
        :effect (and (clear ?b) (on-table ?b) (not (holding ?b)))
    )
    (:action stack
        :parameters (?b - block ?o - block)
        :precondition (and (holding ?b) (clear ?o))
        :effect (and (on ?b ?o) (clear ?b) (not (holding ?b)))
    )
    (:action unstack
        :parameters (?b - block ?o - block)
        :precondition (and (on ?b ?o) (clear ?b))
        :effect (and (holding ?b) (clear ?o) (not (on ?b ?o)))
    )
)
```

And then we can define the problem:

```
(define (problem blocksworld)
    (:domain blocks)
    (:objects A B C - block)
    (:init
        (on-table A) (on B A) (on C B)
        (clear C) (clear A)
    )
```

```
    (:goal (and (on A B) (on C A)))
)
```

## Appendix B: A more legible python example.

```python
rooms = ["room(one)", "room(two)", "room(three)"]
boxes = ["box(red)", "box(blue)", "box(green)"]
robot = "robot"

AT = "At"  # predicate name (just a string)


def at(thing, room):
    return (AT, thing, room)


class Actions:
    @staticmethod
    def goto(destination):
        # Precondition: can't "goto" where you already are
        def is_applicable(state):
            return at(robot, destination) not in state

        def apply(state):
            new_state = state.copy()

            # Remove old At(robot, room_?) facts
            for r in rooms:
                fact = at(robot, r)
                if fact in new_state:
                    new_state.remove(fact)

            # Add new At(robot, destination)
            new_state.add(at(robot, destination))
            return new_state

        apply.name = "goto(" + destination + ")"
        apply.is_applicable = is_applicable
        return apply

    @staticmethod
    def carry(unique_box, destination):
        # Precondition: robot and box are in the same room
        def is_applicable(state):
            for r in rooms:
                if at(robot, r) in state and at(unique_box, r) in state:
                    return True
            return False
```

```
        def apply(state):
            new_state = state.copy()

            # Find the current shared room, then move both robot and box there
            current_room = None
            for r in rooms:
                if at(robot, r) in new_state and at(unique_box, r) in new_state:
                    current_room = r
                    break

            if current_room is None:
                return new_state  # should not happen if precondition checked

            # Remove old robot + box locations
            for r in rooms:
                new_state.discard(at(robot, r))
                new_state.discard(at(unique_box, r))

            # Add new locations
            new_state.add(at(robot, destination))
            new_state.add(at(unique_box, destination))
            return new_state

        apply.name = "carry(" + unique_box + ", " + destination + ")"
        apply.is_applicable = is_applicable
        return apply


def breadth_first_search(start_state, goal_facts, actions, max_depth):
    queue = [(start_state, [])]  # (state, path)

    while queue:
        state, path = queue.pop(0)

        # Goal test: all goal facts must be present (predicate-generic)
        if goal_facts.issubset(state):
            return path

        if len(path) >= max_depth:
            continue

        for action in actions:
            if action.is_applicable(state):
                next_state = action(state)
                queue.append((next_state, path + [action.name]))

    return None


# Ground the domain.
# This is where we go from "carry some hypothetical box to some hypothetical
```

```
# location" to "carry the red box to room one" and "carry the red box to room
# two" and "carry..."
grounded_actions = []
for r in rooms:
    grounded_actions.append(Actions.goto(r))
for b in boxes:
    for r in rooms:
        grounded_actions.append(Actions.carry(b, r))


# Start state s0
s0 = {
    at("box(red)", "room(one)"),
    at("box(blue)", "room(one)"),
    at("box(green)", "room(one)"),
    at(robot, "room(two)"),
}

# Goal: At(box(red), room(two))
goal = {at("box(red)", "room(two)")}

plan = breadth_first_search(s0, goal, grounded_actions, max_depth=5)

print("Goal:", goal)
print("Plan:", plan)
```