Automated Planning: Theory and Practice

Prof. Roveri

Martina Panini

2025–2026

Contents

Chapter 1

Classical Planning and PDDL

Formal representation of planning uses a First-Order language. The Classical Representation is the simplest representation that is reasonable to use for modeling.

We are interested in **objects** in the world. The modeling issue is to determine which objects exist and are relevant for the problem and the objectives.

1.0.1 Predicates, Terms, Atoms, Ground Atoms, States

We are constructing a first-order language **L**. Each object is modeled as a constant.
L contains: {`c1, c2,c3,p1, p2,loc1, loc2,r1,crane1,...`}.

An STS (State Transition System) only assumes there are **states**. The definition of a state do not depend on what s "represents" or "means". For an STS, s is a state if it can execute action a in s and $\gamma(s, a) = \{s'\}$ where γ is a **transition function**.

Predicates are properties of the world, of single objects, Non-Boolean properties, or they can represent relations between objects.
E.g. `attached(pile, location)` means that the pile is in the given location.
To define predicates is important to determine what is relevant for the problem and objective.
Predicates can be *fixed* or *dynamic* (can be modified by actions).

Term is a constant symbol or variable. E.g. `location`.

Atom is a predicate symbol applied to the intended number of terms. E.g. `occupied(location)`.

Ground Atoms are atoms without variables: fact. E.g. `occupied(loc2)`. They work only with constants.

A state of the world should specify exactly which facts (ground atoms) are true/false in the world at a given time instant. We know all predicates that exist and also which objects exist. From this knowledge we can calculate all ground atoms, these are the facts to keep track of. Every assignment of true/false to the ground atoms is a distinct state. Fo classical planning the number of states is $2^{number\ of\ atoms}$.

First-Order Representation is an efficient specification and storage of a single state. Specify which states are true, all other facts have to be false. A classical state is a set of all ground atoms that are true.

We assume a complete information about initial state s_0. We must know everything about predicates and ojects we have specified.
A **goal** g is a finite set of ground atoms.

1.1 Operators and Actions

Actions have an internal structure. They require a state to be executed and make an effect on the state. In the classical representation do not define actions directly but define a set O of operators. Each operator is parameterized, defines many actions.

E.g. "crane k at location l takes container c off container d in pile p": `take(k,l,c,d,p)`.
Has precondition and effects.

In the classical representation every ground instantiation of an operator is an action. Also has instantiated preconditions and effects.

$$A = \left\{ \begin{array}{l} a \text{ is an instantiation} \\ a \text{ of an operator } o \in O \\ \text{using constants in } L \end{array} \right\}$$

There are actions which preconditions can never be satisfied in reachable states, and those actions can be filtered out before planning even starts. An action a is applicable in a state s if its positive preconditions belong to set s and negative preconditions does not belong to state s. Applying an action will add positive effects and delete negative effects. Formalizing:

$$\gamma(s,a) = \begin{cases} \emptyset & \text{if } \mathrm{precond}^+(a) \not\subseteq s \text{ or } \mathrm{precond}^-(a) \cap s \neq \emptyset, \\ (s \setminus \mathrm{effect}^-(a)) \cup \mathrm{effect}^+(a) & \text{otherwise.} \end{cases}$$

1.2 Domains and Problem Instances

Domain-Independent planning provide an high level problem description. These planner are written for generic planning problem, but are difficult to create.
Domain description: the world in general, predicates and operators.
Instance Description: our current problem. Objects, initial state and goal.
To solve problems in other domains keep the planning algorithm and write a new high-level description of the problem domain.

1.3 PDDL

The Planning Domain Definition Language (PDDL) is recognized as the **standard specification language** for classical planning.

1.3.1 Basic Structure of PDDL

PDDL separates the description of the domain from the description of the specific problem instance. The domain file describes the "world in general".

1.3.2 Operators (Actions)

Action applicability and state update can be expressed mathematically:

$$a \text{ is applicable in } s \iff precond^+(a) \subseteq s \text{ and } precond^-(a) \cap s = \emptyset$$
$$s' = (s \setminus effects^-(a)) \cup effects^+(a)$$

1.3.3 PDDL Language Components

Objects and Types
Objects are "things in the world". In classical representation, they must be a finite set. PDDL supports type hierarchies, where object is the default top-level super-type. Instance-specific objects are defined in the problem file. Constants existing in all instances are defined in the domain file.

1.3.4 Predicates

Predicates describe properties of objects and relations between them. In PDDL, they are defined using a Lisp-like syntax, and variables are prefixed with '?' (e.g., `(adjacent ?l1 ?l2 - location)`).

Predicates can be:

- Rigid: Properties that do not change with actions.
- Fluent/Dynamic: Properties changed by actions.

Sometimes redundant predicates are added to assist planners' heuristics due to limited quantification support in some PDDL levels.

1.3.5 Initial State and Goal

- Initial State (:init): In classical planning, complete information about the initial state s_0 is assumed. The initial state is a set (list) of true atoms.
- Goal States (:goal): Goals specify what we want to be true. At the `:strips` level, goals only support positive conjunctions, e.g., `(and (in c1 p2) (in c3 p2))`. More expressive PDDL levels allow negation, disjunction, and quantifiers.

1.3.6 Operators (Actions)

Operators in PDDL represent ways to change the world. Each operator is parameterized and defines multiple actions. An operator includes parameters, precondition and effects.

Chapter 2

Planning as Search

2.1 Introduction

Planning is formally defined as a search problem: using knowledge about the world, available actions, and their outcomes to deliberate on a course of action before execution begins.

This paradigm shift requires modeling the planning process not as the linear addition of steps, but as the exploration of a state space to find a trajectory from an initial state to a goal state.

2.2 Formal Components of Search

To instantiate a planning problem as a search task, six distinct components must be defined. These components bridge the gap between the domain description (e.g., PDDL) and the solver's algorithmic core.

1. **Node Structure:** A search node differs from a world state. While a state represents a snapshot of the environment (e.g., a set of ground predicates like $\{on(A, B), clear(A)\}$), a search node encapsulates the state of the *search* itself. In State Space Search, a node wraps a world state; in Partial Order Causal Link (POCL) planning, a node represents a partial plan structure.
2. **Initial Node Generation:** A function is required to transform the problem instance (initial facts) into the root node of the search tree.
3. **Successor Function (Branching Rule):** Since the full state space is often too large to represent explicitly, it is generated incrementally. The successor function $\Gamma(n)$ returns the set of all nodes reachable from node n by applying a single valid action.
4. **Solution Criterion:** A deterministic test to check if a node satisfies the goal conditions defined in the problem instance.
5. **Plan Extraction:** Upon reaching a solution node, the solver must be able to reconstruct the sequence of actions (the plan) that led from the initial node to the current state.
6. **Search Strategy:** The policy used to select the next node from the frontier (the set of open nodes) for expansion. This choice determines the completeness and optimality of the planner.

2.3 Search Algorithms: Tree vs. Graph Search

The topology of the search space dictates the memory requirements and the robustness of the algorithm. We distinguish between searching a tree (where paths are unique) and searching a graph (where cycles and redundant paths exist).

2.3.1 The General Search Scheme

The core loop of a search planner manages a set of candidates, typically referred to as the `open` list.

Algorithm 1 Basic Search Procedure

```
function SEARCH(problem)
    $initial \leftarrow$ MAKE-INITIAL-NODE($problem$)
    $open \leftarrow \{initial\}$
    while $open \neq \emptyset$ do
        $node \leftarrow$ SELECT-AND-REMOVE($open$)                ▷ Strategy dependent
        if IS-SOLUTION($node$) then
            return EXTRACT-PLAN($node$)
        end if
        for each $successor$ in EXPAND($node$) do
            $open \leftarrow open \cup \{successor\}$
        end for
    end while
    return Failure
end function
```

2.3.2 Graph Search and Cycle Detection

In a graph, multiple sequences of actions can lead to the same state, and reversible actions can create infinite loops (e.g., moving block A to B and back to A).

- **Tree Search Approach:** Does not track visited states. It is memory efficient but risks infinite loops and redundant work.
- **Graph Search Approach:** Maintains a "closed list" (or `added` set) of all generated states. If a successor leads to a state already in the closed list, it is discarded.

Algorithm 2 Graph Search with Duplicate Detection

```
function GRAPH-SEARCH(problem)
    $initial \leftarrow$ MAKE-INITIAL-NODE($problem$)
    $open \leftarrow \{initial\}$
    $added \leftarrow \{initial\}$                              ▷ Tracks all visited states
    while $open \neq \emptyset$ do
        $node \leftarrow$ SELECT-AND-REMOVE($open$)
        if IS-SOLUTION($node$) then
            return EXTRACT-PLAN($node$)
        end if
        for each $newnode$ in EXPAND($node$) do
            if $newnode \notin added$ then
                $open \leftarrow open \cup \{newnode\}$
                $added \leftarrow added \cup \{newnode\}$
            end if
        end for
    end while
    return Failure
end function
```

2.4 Search Strategies

The efficiency of the planner depends entirely on the order in which nodes are removed from the `open` list.

2.4.1 Uninformed Search

These strategies rely solely on the structure of the graph without domain-specific knowledge.

- **Breadth-First Search (BFS):** Guarantees finding the shortest plan (in terms of steps) but suffers from exponential memory usage.
- **Depth-First Search (DFS):** Memory efficient but not optimal and susceptible to getting lost in deep sub-trees.
- **Uniform Cost Search (Dijkstra):** Expands the node with the lowest path cost $g(n)$.

2.4.2 Informed (Heuristic) Search

These strategies utilize a heuristic function $h(n)$ to estimate the distance to the goal.

- **Greedy Best First:** Minimizes $h(n)$. Fast but greedy.
- **A*:** Minimizes $f(n) = g(n) + h(n)$. Optimal if $h(n)$ is admissible.
- **Hill Climbing / Enforced Hill Climbing:** Local search techniques often used for satisficing planning due to their speed, despite lacking completeness guarantees.

2.5 Practical Tools

Implementation of these concepts is typically done via established planning frameworks accessible via command-line interfaces.

- **Planutils:** A package manager for Linux that facilitates the installation of various IPC (International Planning Competition) planners.
- **Fast Downward:** A widely used heuristic planning system based on the classical search paradigm.
- **Planning.domains:** An online environment for prototyping PDDL domains and problems.

Chapter 3

Forward State Space

3.1 Introduction to State Space

The state space serves as the fundamental search space for automated planning. In the context of *forward planning* (also known as forward-chaining or progression), the structure is a directed graph where:

- **Nodes** represent the states of the world.
- **Edges** represent the actions (transitions) that transform one state into another.

To formalize a search problem within this space, five key components are required:

1. **Structure**: The implicit graph defined by states and actions.
2. **Initial Node** (n_0): A node corresponding directly to the initial state s_0 of the problem instance.
3. **Branching Rule**: The transition logic. For a given state s, the branching rule generates all successor states $\gamma(s, a)$ for every applicable action a. In forward planning, actions are applied in their natural direction (progression).
4. **Solution Criterion**: A logical test to determine if the state associated with a node satisfies the goal formula (G).
5. **Plan Extraction**: Upon finding a solution node, the sequence of actions leading from n_0 to the goal node must be reconstructable.

3.2 General Properties of the State Space

The topology of the state space dictates the complexity of the search. Two critical properties often misunderstood are symmetry and connectivity.

3.2.1 Symmetry and Reversibility

The state space is not necessarily symmetric. While some actions (e.g., `open(door)` and `close(door)`) allow for reversibility, returning the agent to the previous state, others are inherently irreversible. For example, an action `crack(egg)` transitions the state to one where the egg is broken; typically, no `uncrack(egg)` action exists. Consequently, the search graph is directed, and one cannot assume that edges are bidirectional.

3.2.2 Connectivity

The state space is not guaranteed to be connected. It may be partitioned into disjoint sub-graphs. A classic example involves resource constraints or "dead-end" logic:

- Partition A: States where the agent possesses a rocket.
- Partition B: States where the agent does not possess a rocket.

If no actions exist to buy or sell a rocket, an agent starting in Partition B can never reach any state in Partition A. The search is strictly confined to the connected component containing the initial state s_0.

3.3 Exploring the State Space: Modeling and Reachability

A fundamental distinction in automated planning is the difference between the *logical state space* (all combinatorial possibilities of predicates) and the *reachable state space* (states actually attainable from s_0).

Combinatorial Explosion vs. Reachability

In the example of the Tower of Hanoi, if we define a state simply as a combination of truth values for all ground atoms:

- With 6 objects (3 pegs, 3 discs) and predicates like `clear`, `on`, and `smaller`, the number of logical combinations can reach 2^{78}.
- This huge number includes "impossible" states, such as a disc being on top of itself or multiple objects occupying the same coordinate logic that the physics of the domain would forbid but the raw logic allows if not constrained.

Depending on the modeling strategy (e.g., removing redundant predicates like `clear` or static predicates like `smaller`), the logical space can be reduced (e.g., to 2^{42} or 5^3). However, the *reachable* state space from a valid initial configuration is vastly smaller (e.g., only 27 states for the 3-disk Hanoi problem).

Key Insight: States are not inherently "reachable" or "unreachable" in absolute terms; they are reachable relative to a specific starting state s_0. If the planner initializes in a "physically impossible" state (e.g., a floating disc), the set of reachable states will be disjoint from the standard solution space.

3.4 Forward State Space Search Algorithm

The Forward State Space Search algorithm explores the graph from s_0 looking for a state s_g that satisfies the goal. Crucially, the search graph is **not pre-computed**. It is generated dynamically because the full graph is too large to store in memory.

3.4.1 Characteristics

- **Soundness:** The algorithm is always sound; if it returns a plan, that plan is a valid sequence of actions leading to the goal.
- **Completeness:** Completeness depends on the specific search strategy used to manage the `open` set (e.g., Breadth-First Search is complete, Depth-First Search may not be in infinite spaces).
- **Nodes:** A search node is technically a pair $\langle s, \pi \rangle$, storing both the current state and the path (plan) taken to reach it.

3.5 Pruning

To handle the exponential size of the search space, pruning is essential. It allows the algorithm to discard nodes that are provably suboptimal or redundant.

3.5.1 Cycle and Cost Pruning

If the search reaches a state s via a path with cost C_{new}, and s has already been visited via a path with cost C_{old}:

- If $C_{new} \geq C_{old}$, the new path is discarded (pruned).
- If $C_{new} < C_{old}$, the new path is kept, and the old one (if still active) is effectively updated.

3.5.2 Subset Pruning

A more aggressive pruning strategy involves comparing the logical facts of states. If we reach a node n' with state s' and cost C', and we have previously visited a node n with state s and cost C such that:

1. $s \subseteq s'$ (The facts in s are a subset of facts in s'), AND
2. $C \leq C'$ (The previous path was cheaper or equal),

Then, under the assumption of **positive preconditions and effects**, node n' can be pruned. Logic dictates that if a goal can be reached from the "smaller" state s, it can also be reached from s', but since s was cheaper, s' is suboptimal.

Warning: This pruning rule fails if the domain includes *negative preconditions* (e.g., action requires `not (ontable B)`). In such cases, the "larger" state s' might fail a negative check that s satisfies, making s' a dead end while s is viable. Thus, strictly subsumed states cannot be safely pruned in the presence of negative preconditions.

Chapter 4

Backward State Space

Logical planning also allows searching **backward** from the goal to the initial state. In Backward Search, the direction of exploration is inverted.

Although the concept seems symmetric, backward search introduces specific complications that require a shift from searching in the *State Space* to searching in the *Goal Space*.

4.1 Complications of Backward Search

Unlike forward search, where we progress through concrete states, backward search faces three main structural challenges.

Complication 1: Dynamic Graph Generation

The search graph cannot be pre-computed. It must be expanded dynamically starting from the goal states. This requires an inverse transition function, denoted as $\gamma^{-1}(s, a)$, to determine predecessors.

Complication 2: Non-Determinism

In forward planning with deterministic actions, applying an action a to state s yields a unique state s'. In the backward direction, this determinism is lost. For example, if an agent is at the `shop` after executing `drive_to_shop`, where was it before? It could have been at `home`, at `work`, or at `school`. Thus, the inverse transition γ^{-1} produces a *set* of possible predecessor states, not a single state.

Complication 3: Goal Sets (Partial States)

We rarely wish to reach a specific, fully defined state. Instead, we want to reach *any* state that satisfies a goal formula, such as $g = \{(on\ A\ B), \neg(on\ C\ D)\}$.

- A goal specification typically defines a set of literals that must hold.
- It ignores irrelevant facts (e.g., the location of block D might not matter).
- Consequently, a "goal" in backward search represents a set of many possible concrete states.

4.2 The Solution: Goal Space Search

To address these complications, backward search is performed in the **Goal Space** rather than the State Space.

- **Nodes**: Sets of ground literals (sub-goals) representing what must be true at that point in the plan.
- **Transitions**: Regression of goals through actions.

Instead of asking "which state allows me to execute a?", we ask "what must be true before a so that a achieves my current goal?".

4.3 Relevance and Regression

4.3.1 Action Relevance

In forward search, we look for *applicable* actions (preconditions met). In backward search, we look for **relevant** actions. An action a is relevant to a goal g if:

1. **Contributes**: It achieves at least one literal in g ($g \cap \text{effects}(a) \neq \emptyset$).
2. **Consistent**: It does not destroy any needed literal in g.
   - Does not delete positive goals ($g \cap \text{effects}^{-}(a) = \emptyset$).
   - Does not add negative goals that contradict positive needs ($g \cap \text{effects}^{+}(a) = \emptyset$).

4.3.2 Regression (γ^{-1})

While forward search uses *Progression* to generate successors, backward search uses **Regression**. If we have a goal g and a relevant action a, the regressed goal g' is the set of conditions that must hold before a is executed to ensure g holds afterwards.

4.4 The Backward Search Algorithm

The search process operates as follows:

1. **Initial Node**: The set of literals specified in the problem goal G.
2. **Branching Rule**: For a node g, generate successors $\gamma^{-1}(g, a)$ for every action a relevant to g.
3. **Solution Criterion**: A node g is a solution if $g \subseteq s_0$ (i.e., all required literals are satisfied by the initial state).
4. **Plan Extraction**: The path from the initial goal node to the solution node gives the plan in reverse order.

Chapter 5

The Partial Order Causal Link Search Space

Previous chapters explored planning strategies that operate within the *State Space*, whether searching forward from the initial state or backward from the goal. These approaches share a fundamental limitation: *they force the planner to commit to a total ordering of actions immediately.*

Forward and backward search heuristics must be extremely sophisticated to understand which actions are necessary and exactly when they should be placed. **Partial Order Causal Link (POCL)** planning shifts the paradigm. Instead of searching through concrete states of the world, we search through the space of *plans*. The core philosophy is the **Principle of Least Commitment**: do not order actions unless absolutely necessary to resolve a conflict.

5.1 Plan Structure in POCL

In POCL, a plan is not a linear sequence of actions but a data structure consisting of four components.

5.1.1 Components of a Partial Order Plan

A plan is defined as a tuple $\pi = \langle \mathcal{A}, \mathcal{O}, \mathcal{L}, \mathcal{B} \rangle$:

- **Actions ($\mathcal{A}$)**: A set of actions selected for the plan. To maintain uniformity, we introduce two dummy actions:
  - `InitAction`: Has no preconditions; its effects represent the initial state s_0.
  - `GoalAction`: Has no effects; its preconditions represent the goal literals G.
- **Ordering Constraints ($\mathcal{O}$)**: A set of constraints of the form $a_i \prec a_j$, meaning action a_i must be executed before a_j. This defines a partial order, not a total sequence.
- **Causal Links ($\mathcal{L}$)**: A set of links denoted as $a_i \xrightarrow{p} a_j$. This signifies that action a_i produces an effect p, which is consumed by action a_j to satisfy one of its preconditions. The link implies a "protection interval": p must remain true between a_i and a_j.
- **Binding Constraints ($\mathcal{B}$)**: Used in lifted planning to handle variables (e.g., $?x \neq ?y$ or $?x = A$).

5.2 The POCL Search Space

Unlike state-space search, there is no "current state" in POCL. We cannot simply look at the world and apply an action. Instead, the search nodes are the plans themselves.

5.2.1 The Initial Node

The search begins with a minimal plan containing only the dummy actions and the fundamental requirement that the plan must start and end:

$$\pi_0 = \langle \{InitAction, GoalAction\}, \{InitAction \prec GoalAction\}, \emptyset, \emptyset \rangle \tag{5.1}$$

Initially, the preconditions of the `GoalAction` are unsatisfied. These unsatisfied preconditions drive the search.

5.2.2 Flaws

The search proceeds by identifying and repairing **flaws** in the plan. A plan is a solution if and only if it has zero flaws. There are two types of flaws:

1. **Open Goals**: A precondition of an action in the plan is not yet supported by a causal link.
2. **Threats**: An action in the plan threatens to delete a proposition protected by an existing causal link.

5.3 Resolving Flaws

The branching rule in the search algorithm involves selecting a flaw and generating a successor plan for each possible method of resolving that flaw.

5.3.1 Resolving Open Goals

If an action a_{need} has a precondition p that is an Open Goal, we must find an action a_{prov} (provider) that has p as an effect.

- **Strategy 1: Reuse.** Choose an action a_{prov} already in $\mathcal{A}$.
- **Strategy 2: New Action.** Instantiate a new action from the domain operators and add it to $\mathcal{A}$.

Once a_{prov} is identified, we refine the plan by:

1. Adding the causal link $a_{prov} \xrightarrow{p} a_{need}$ to $\mathcal{L}$.
2. Adding the ordering constraint $a_{prov} \prec a_{need}$ to $\mathcal{O}$.
3. If a_{prov} is new, adding its preconditions to the list of Open Goals.

5.3.2 Resolving Threats

A **Threat** occurs when we have a causal link $a_i \xrightarrow{p} a_j$ and a third action a_k (the threat) exists in the plan such that:

1. a_k has an effect $\neg p$ (it deletes the condition needed by a_j).
2. The ordering constraints $\mathcal{O}$ allow a_k to occur between a_i and a_j (i.e., it is consistent that $a_i \prec a_k \prec a_j$).

To resolve a threat, we must force a_k out of the protection interval using ordering constraints:

- **Demotion:** Force the threat to happen before the provider. Add constraint $a_k \prec a_i$.
- **Promotion:** Force the threat to happen after the consumer. Add constraint $a_j \prec a_k$.

If adding either constraint creates a cycle in $\mathcal{O}$, that resolution path is invalid (inconsistent).

5.4 Lifted Planning

In standard POCL, we might instantiate actions with specific objects (e.g., `unstack(A, B)`). This leads to a high branching factor if there are many objects. **Lifted Planning** addresses this by using variables instead of constants (e.g., `unstack(?x, ?y)`) and delaying the binding of these variables.

- **Unification:** When establishing a causal link, we unify the provider's effect with the needed precondition. This generates binding constraints (e.g., $?x = A$).
- **Separation:** When resolving threats, if an action *might* threaten a link depending on variable assignments, we can resolve it by adding an inequality constraint (e.g., $?x \neq A$) instead of enforcing an ordering.

This dramatically reduces the size of the search space by grouping many concrete plans into a single abstract plan node.

Chapter 6

General Search Strategies

The fundamental problem in automated planning is navigating the massive combinatorial explosion of the state space. Having defined the search space (Forward, Backward, POCL) is not enough; we need an efficient strategy to traverse it.
A critical strategic decision is the goal of the search:

- **Optimizing:** We seek the *best* possible plan according to a specific metric.
- **Satisficing:** We seek a plan that meets an acceptability threshold.

Uninformed vs. Informed Search

- **Uninformed (Blind) Search:** Relies solely on the structure of the search space and the cost accumulated so far ($g(n)$). It has no domain-specific knowledge about the goal location.
- **Informed (Heuristic) Search:** Utilizes a heuristic function ($h(n)$) that estimates the cost to reach the goal. This provides direction to the search, drastically pruning the search space.

6.1 Uninformed Search: Dijkstra's Algorithm

Dijkstra's algorithm is the archetype of optimal, uninformed search. It explores the state space by expanding nodes in increasing order of their path cost $g(n)$.

- **Strategy:** Select the node n from the `OPEN` list with the minimal $g(n)$.
- **Completeness:** Yes, if costs are positive.
- **Optimality:** Yes, it guarantees finding the shortest path.
- **Complexity:** $O(|E| + |V| \log |V|)$, where $|V|$ is the number of vertices (states) and $|E|$ is the number of edges (transitions).

Dijkstra's algorithm is practically useless for large planning problems. Even if we assume uniform action costs, Dijkstra's algorithm behaves like a Breadth-First Search, expanding all plans of length k before considering any plan of length $k + 1$. Since the number of valid plans grows exponentially with length, Dijkstra will exhaust memory and time long before reaching a deep goal state.

6.2 Informed Search Strategies

To solve interesting problems, we must guide the search using heuristics. A generic **Best First Search** maintains an `OPEN` list and selects the "best" node according to some evaluation function.

6.2.1 Greedy Best First Search

This strategy selects the node that appears closest to the goal, minimizing the heuristic value $h(n)$.

- **Evaluation Function:** $f(n) = h(n)$.
- **Behavior:** It ignores the cost already paid ($g(n)$). It is "greedy" because it rushes towards the goal.

- **Properties:** It is generally not optimal and not complete (can get stuck in loops or dead ends without loop detection), but it is often very fast.

6.2.2 A* Search

A* combines the cost so far with the estimated cost to go.

- **Evaluation Function:** $f(n) = g(n) + h(n)$.
- **Optimality Condition:** If the heuristic $h(n)$ is **admissible** (i.e., it never overestimates the true cost $h^*(n)$), A* is guaranteed to return an optimal solution.
- **Efficiency:** With an admissible heuristic, A* expands the minimum number of nodes necessary to guarantee optimality. However, if the heuristic is weak (close to 0), A* degenerates into Dijkstra's algorithm.

6.2.3 Weighted A*

To trade optimality for speed, we can bias the search towards the goal:

$$f(n) = g(n) + w \cdot h(n) \tag{6.1}$$

where $w > 1$. This effectively makes the algorithm "more greedy." The solution found will cost at most w times the optimal cost. **Repeated Weighted A*** iteratively runs Weighted A* with decreasing weights (e.g., $w = 5, 3, 2, 1$) to quickly find a solution and then refine it.

6.3 Local Search Strategies

Sometimes, maintaining the full `OPEN` list (all unexpanded nodes) is too memory-intensive. Local search strategies keep only the current node (and immediate neighbors), sacrificing completeness for memory efficiency.

6.3.1 The Problem of Local Optima and Plateaus

Hill climbing relies on the heuristic gradient. It fails in two common scenarios:

1. **Local Optima:** The current node is better than all neighbors, but not the global solution. The algorithm gets stuck.
2. **Plateaus:** All neighbors have the same heuristic value ($h(n_{best}) = h(n)$). The algorithm has no gradient to follow.

Standard Hill Climbing stops at these points (impasses).

6.3.2 Enforced Hill Climbing (EHC)

Used in the FastForward (FF) planner, EHC addresses the plateau problem without full backtracking.

- If no neighbor improves the heuristic, EHC switches to **Breadth-First Search** (BFS) starting from the current node.
- It expands layers until it finds *any* node n' with $h(n') < h(n)$.
- Once found, the search commits to the path to n', discards the BFS tree, and resumes Hill Climbing from n'.

This strategy is aggressive and non-optimal (it commits to a path and never looks back), but extremely effective for satisficing planning in many domains.

Chapter 7

Heuristics: An Overview

The heuristic function estimates the ”distance” or ”cost” from a given search node n to a goal state.

7.1 What to Measure: Cost vs. Length

Defining what the heuristic $h(n)$ actually measures is the first step in designing a planner. We must distinguish between two concepts of ”distance”: plan length and action cost.

Plan Length

In simple domains (e.g., the standard Tower of Hanoi), we often assume that every action has a uniform cost (usually 1). In this case, the ”cost” of a plan is simply the number of steps ($|\pi|$). The heuristic estimates how many actions are needed to reach the goal.

Action Costs

In more realistic scenarios, actions have different costs (e.g., travel time, fuel consumption, risk). A plan with many cheap actions might be preferred over a short plan with expensive actions. In this context, $h(n)$ estimates the sum of the costs of the actions required to reach the goal, not just the number of actions.

7.2 The Perfect Estimate: $h^*(n)$

The theoretical ideal for any heuristic is the **true remaining cost**, denoted as $h^*(n)$ and represents the cost of the optimal plan starting from state n to a goal state. If we had access to $h^*(n)$, planning would be trivial. A simple hill-climbing algorithm that always chooses the neighbor with the minimum h^* would trace the optimal path directly to the goal with no backtracking.
The Paradox of h^*: While desirable, computing $h^*(n)$ perfectly is exactly as hard as solving the planning problem itself. Therefore, we cannot use h^* directly. We must settle for an *estimate* that is: cheaper to compute than solving the problem and accurate enough to provide useful guidance.

7.3 Desired Properties of Heuristics

The effectiveness of a heuristic depends on the search strategy being used. Different strategies impose different requirements on $h(n)$.

If the goal is to find an optimal plan using A*, the heuristic must be **admissible**. A heuristic is admissible if it never overestimates the true cost. Among admissible heuristics, a heuristic h_A is better (more informed) than h_B if $h_A(n) > h_B(n)$ for all n.

If the goal is simply to find *a* solution (satisficing), admissibility is not required. Strategies like Greedy Best First Search or Hill Climbing rely on the **gradient** or relative ordering provided by the heuristic. They need $h(n)$ to correctly indicate which neighbor is ”better” (closer to the goal), even if the absolute numbers are wrong. An inadmissible heuristic that drastically overestimates cost but provides a smooth descent towards the goal is often superior for satisficing planners than an admissible but ”flat” heuristic.

There is a fundamental trade-off in heuristic design: a complex heuristic might provide a very precise estimate (reducing the number of nodes expanded). This indicates the Accuracy of the heuristic. But if computing the heuristic for a single node takes 1 second, and it saves expanding 100 nodes that would have taken 0.01 seconds total, the heuristic is a net loss. This is called Overhead.

A good heuristic must be efficiently computable. The total time ($Time = NodesExpanded \times CostPerNode$) must be minimized. Often, a ”slightly worse” heuristic that is ultra-fast to compute yields a faster overall planner than a ”perfect” heuristic that is slow to calculate.

7.4 Speed vs. Plan Quality

Finally, the choice of heuristic and strategy dictates the output profile of the planner:

- **Find a solution quickly:** Strategies and heuristics should prioritize ”easy” paths (e.g. fewer preconditions, less resource contention), even if they result in longer, costlier plans.
- **Find a good solution:** Strategies must explore deeper and heuristics must account for action costs accurately, accepting a longer runtime to ensure the resulting plan is high-quality.

Chapter 8

A simple Domain Independent Heuristic Function

Uninformed search strategies fail in large state spaces due to the combinatorial explosion. To guide the search effectively, we need **heuristics**. However, designing a specific heuristic for every new problem domain is impractical. We need **Domain Independent Heuristics**—strategies that can be applied to any problem described in a standard language without manual tweaking.

8.1 The Goal Count Heuristic (h_{gc})

The most intuitive domain-independent heuristic is the **Goal Count**. It estimates the distance to the goal by assuming that every unsatisfied goal literal requires exactly one unit of effort (or action) to be achieved.

Given a state s and a goal set g (comprising positive literals g^+ and negative literals g^-), the heuristic value $h_{gc}(s)$ is simply the number of goal literals not currently satisfied in s.

$$h_{gc}(s) = |(g^+ \setminus s) \cup (g^- \cap s)|$$

This heuristic implicitly relies on the **Subgoal Independence Assumption**. It presumes that:

1. Goals can be solved independently.
2. Achieving one goal does not undo another.
3. The cost to achieve a goal is roughly uniform.

In reality, planning is difficult precisely because these assumptions rarely hold.

8.2 Search Behavior and Pathologies

When coupled with a strategy like **Greedy Best First Search**, h_{gc} attempts to minimize the number of remaining goals. While this seems logical, it leads to severe issues in domains with interacting subgoals, such as the Blocks World.

The Local Optima Trap

Greedy search, driven by h_{gc}, will resist taking necessary steps that temporarily break goals (regression) to achieve the final configuration. It will view the necessary dismantling of the tower as a step *backwards*, creating local optima or plateaus where the planner gets stuck or wanders aimlessly.

8.3 Properties of Goal Count

8.3.1 Admissibility

Is h_{gc} admissible? Generally, no.

- Overestimation: if a single action can satisfy multiple goal literals (e.g., an explosion satisfying 'destroyed(A)' and 'destroyed(B)'), h_{gc} counts them as separate costs (e.g., 2), while the real cost is 1. Since $h(n) > h^*(n)$, it is inadmissible.
- Underestimation: conversely, if achieving a single goal literal requires a long sequence of actions (e.g., moving a block from the bottom of a stack), h_{gc} counts it as 1, vastly underestimating the cost.

It can be made admissible by dividing the count by the maximum number of literals an action can satisfy at once, but this usually results in an extremely weak heuristic.

8.3.2 Effectiveness

Empirical results show that h_{gc} is insufficient for complex problems. In domains like "Elevators," planners using h_{gc} generate orders of magnitude more nodes than state-of-the-art planners.

8.4 Conclusion

The Goal Count heuristic serves as a fundamental baseline. It demonstrates how structure (literals) can be used to guide search. However, its failure to account for action interactions and chain reactions makes it too weak for practical planning. Modern heuristics must look beyond the immediate set of missing goals and analyze the causal structure of the problem (e.g., Relaxed Planning Graphs) to provide useful guidance.

Chapter 9

Domain-Configurable Planning: Hierarchical Task Networks

9.1 The Paradigm Shift: From Goals to Tasks

Until now, our approach to planning has been fundamentally passive regarding domain knowledge. We defined the ”what” (objects, predicates, initial state, goal) and left the ”how” entirely to the planner’s search algorithm. This approach ignores a brutal truth: in many real-world scenarios, we already know the general procedure to solve a problem. Ignoring this knowledge is inefficient.
s **Domain-Configurable Planning** bridges the gap between Domain-Independent planning (flexible but slow) and Domain-Specific planning (fast but rigid). It allows the domain designer to inject ”recipes” or procedural knowledge into the system. The fundamental shift here is moving from achieving a **Goal** to performing a **Task**. A task implies a process, not just a destination.

9.2 Core Components of HTN

Hierarchical Task Network (HTN) planning is the standard framework for this approach. It relies on a strict hierarchy of operations.

The central entity is the Task. We distinguish between two types.
Primitive Tasks correspond directly to atomic actions executable by the agent (operators).
Non-Primitive Tasks (or Compound Tasks) represent high-level activities that cannot be executed directly. They must be broken down.

9.2.1 Methods and Decomposition

The bridge between high-level abstractions and executable actions is the **Method**. A method specifies *how* to decompose a non-primitive task into a set of sub-tasks. A method consists of:

1. The task it decomposes.
2. Preconditions that must hold for the method to be applicable.
3. A network of sub-tasks.

This structure forms a **Simple Task Network (STN)**. The planning process becomes a cycle of decomposing non-primitive tasks using available methods until only primitive tasks (actions) remain.

9.3 Total-Order Simple Task Networks

The simplest form of HTN planning assumes that the sub-tasks within a method are totally ordered. This means a method provides a rigid sequence of steps.

9.3.1 The TOFD Algorithm

The search strategy used here is **Total Order Forward Decomposition (TOFD)**. It operates similarly to a forward state-space search but in the space of task lists. The state of the search is defined by:

- The current plan (sequence of primitive actions found so far).
- The current world state (updated as primitive actions are added).
- The list of remaining tasks to be processed.

The algorithm proceeds by examining the first task in the list. If it is primitive and applicable, it is moved to the plan, and the state is updated. If it is non-primitive, the planner chooses a matching method (this is a choice point, hence a source of backtracking), expands the task into its sub-tasks, and prepends them to the task list. The process repeats until the task list is empty.

9.3.2 Recursion and Expressivity

HTN allows for recursive methods. Consider moving a stack of blocks. We can define a method `move-stack` that:

1. Moves the topmost block (primitive or simple task).
2. Recursively calls `move-stack` on the remainder of the pile.

This recursive structure requires careful modeling, specifically the inclusion of a **base case** to terminate the recursion (e.g., a method that does nothing when the pile is empty), otherwise, the planner will loop infinitely.

9.4 Limitations of Total Order and the Need for Partial Order

While Total-Order HTN is efficient, it suffers from the "interleaving problem." If a method specifies a strict sequence, the planner cannot weave sub-tasks from different high-level tasks together.

Consider a robot that needs to `fetch` two objects, A and B, located in the same room, and bring them back.

- If `fetch` is defined as "Go to location, Pick up, Return", a total-order planner executing `fetch(A)` then `fetch(B)` will generate a plan: Go-PickA-Return, Go-PickB-Return. This is grossly inefficient.
- An intelligent agent should Go, PickA, PickB, then Return.

To achieve this, we must relax the ordering constraints. In **Partial-Order HTN**, methods specify sub-tasks with partial ordering constraints (or no constraints) rather than a rigid sequence. This allows the planner to interleave sub-tasks from different high-level activities to optimize the plan or satisfy complex constraints, significantly increasing flexibility at the cost of a more complex search algorithm (Partial Order Forward Decomposition).

9.5 Modeling Strategies

Effective HTN planning depends heavily on the quality of the domain model. A common modeling strategy is to defer choices. Instead of creating different methods for "truck already at location" vs "truck needs to drive", one can model a generic `deliver` method that calls a sub-task `achieve-truck-at-location`. This sub-task can then have two methods: one that does nothing (if the truck is there) and one that drives the truck. This modularity reduces the combinatorial explosion of methods and makes the domain easier to maintain.

Chapter 10

Delete Relaxation and the Relaxation Principle

In classical planning, achieving a goal often involves executing actions that temporarily undo previously achieved conditions. For example, to move a vehicle from A to D via B and C, one must consume fuel (negating the `have-fuel` condition) and then refuel (re-achieving it).

This necessity to "destroy" facts to make progress makes planning difficult. If we assume that conditions, once achieved, remain true forever, the problem becomes significantly easier. This intuition forms the basis of **Relaxation Heuristics**. By simplifying the problem constraints, we can calculate heuristic estimates ($h(s)$) to guide the search in the original, complex problem.

10.1 The General Relaxation Principle

The fundamental idea of relaxation is to transform a difficult problem P into a simpler problem P' such that optimal solutions to P' provide admissible estimates for P.

10.1.1 Formal Definition

Given a planning problem $P = \langle \Sigma, s_0, S_g \rangle$, a problem P' is a **relaxation** of P if and only if the set of solutions for P is a subset of the solutions for P':

$$\text{Solutions}(P) \subseteq \text{Solutions}(P')$$

If this condition holds, the cost of the optimal solution to the relaxed problem, denoted as $h^*(P')$, is an **admissible heuristic** for the original problem. Since every solution to P is also a valid solution to P', the cheapest solution in P' cannot be more expensive than the cheapest solution in P:

$$h^*(s)_{P'} \leq h^*(s)_P$$

10.1.2 Types of Relaxation

Relaxation does not strictly imply removing information; it implies loosening constraints. Common methods include:

- **Adding Edges:** Adding new transitions to the state space (e.g., allowing tiles in an 8-puzzle to jump over each other).
- **Ignoring State Variables:** Removing specific facts from preconditions and effects.
- **Adding Goal States:** Expanding the set of valid goal states.

10.2 Delete Relaxation

The most prominent form of relaxation in STRIPS planning is **Delete Relaxation**.

10.2.1 Mechanism

In a STRIPS domain, actions have positive effects (add lists) and negative effects (delete lists). Delete relaxation creates a relaxed problem P^+ by ignoring all negative effects.

- **Original Action:** $Effects(a) = Add(a) \cup Del(a)$
- **Relaxed Action:** $Effects(a)^+ = Add(a)$

In the relaxed state space, facts essentially accumulate. Once a literal becomes true, it remains true forever. This property is monotonic: applying an action never decreases the set of true facts.

10.2.2 The Relaxed State Space

It is a common misconception that the relaxed state space is smaller than the original. In reality, it is often fundamentally different and can contain more transitions.

- **Example (Blocks World):** In the original problem, unstacking A from B deletes `on(A,B)`. In the relaxed problem, `on(A,B)` remains true. This allows the planner to reuse the fact `on(A,B)` later, effectively duplicating resources or skipping preconditions that would normally require restoration.

10.3 The Optimal Delete Relaxation Heuristic (h^+)

The optimal heuristic derived from delete relaxation is denoted as h^+.

$$h^+(s) = \text{Cost of the optimal plan for } P^+ \text{ starting at } s$$

10.3.1 Accuracy

h^+ is generally very accurate. In many benchmark domains (e.g., Logistics, Gripper, Miconic), the ratio between the relaxed plan length and the true plan length is close to 1. In the Blocks World, it is proven that $h^+(n) \geq \frac{1}{4}h^*(n)$.

10.3.2 Computational Complexity

Despite the simplification, computing h^+ is computationally expensive.

- While solving the relaxed problem is easier than the original (polynomial time to find *a* solution), finding the **optimal** solution to the relaxed problem is **NP-equivalent**.
- This complexity arises because optimal relaxed planning maps to the *Set Cover* problem.
- Consequently, h^+ is rarely used directly in practice. Instead, planners use approximations of h^+ (such as h_{max} or h_{add}) or heuristics based on the Relaxed Planning Graph (like h_{FF}).

10.3.3 Admissibility Warning

To guarantee admissibility, one must calculate the **optimal** cost in the relaxed space. Simply finding *any* plan for the relaxed problem (which might be long and inefficient) does not provide an admissible heuristic, as that specific suboptimal relaxed plan could be more expensive than the optimal original plan.

Chapter 11

Relaxed Planning Graph

While the optimal relaxed heuristic h^+ is informative, computing it is NP-hard. We need a way to estimate h^+ efficiently.

The **Relaxed Planning Graph (RPG)** is the data structure that allows us to do exactly this. It provides a polynomial-time method to estimate the cost of the relaxed plan, famously used in the FF (Fast-Forward) planner.

11.1 Building the Relaxed Planning Graph

The RPG is a layered graph structure that alternates between **Proposition Levels** (facts) and **Action Levels** (operators). It represents the reachability of facts over time in the relaxed problem.

11.1.1 Structure

- Proposition Level 0 (P_0): Contains all atoms true in the current state s.
- Action Level i (A_i): Contains all actions whose preconditions are present in P_{i-1}.
- Proposition Level i (P_i): Contains all facts in P_{i-1} plus all positive effects of actions in A_i.

Crucially, since this is a *relaxed* graph, facts are never deleted. They accumulate. To model persistence explicitly, we introduce **No-Op (Maintenance) Actions**: for every fact p, a special action $noop\_p$ has precondition p and effect p. This ensures that if p is true at level i, it is also available at level $i+1$.

11.1.2 Construction Algorithm

The graph is built iteratively: 1. Start with $P_0 = s$. 2. Identify all applicable actions given P_k (including No-Ops). 3. Add their positive effects to form P_{k+1}. 4. Repeat until all goal literals G are present in the last proposition level (or the graph stabilizes without reaching the goal, implying the goal is unreachable).

11.2 Extracting a Heuristic (h_{FF})

Building the graph tells us *if* the goal is reachable, but the graph itself is not a plan. To get a heuristic value, we must extract a solution from the graph.

11.2.1 Backward Solution Extraction

We search backward from the last level m to level 0:

1. **Goal Set:** Start with the set of goals G at level m.
2. **Select Actions:** For each goal $g \in G$, choose an action a at level m that achieves g.
    - Preference is usually given to No-Ops (cost 0) or actions that achieve multiple pending goals simultaneously.

3. **Regression:** The new goal set for level $m-1$ becomes the union of the preconditions of all selected actions.

4. Repeat until level 0 is reached.

The heuristic value $h_{FF}(s)$ is the number of actions in this extracted plan (excluding No-Ops). Alternatively, if actions have costs, it is the sum of the costs.

11.2.2 Properties of h_{FF}

- **Not Admissible:** The extraction process is greedy (it picks the first available action that satisfies a goal). It does not guarantee finding the *shortest* relaxed plan. Therefore, $h_{FF}(s) \geq h^+(s)$, making it inadmissible.

- **Highly Informative:** Despite being inadmissible, h_{FF} is extremely effective for satisficing planning (finding *any* solution quickly) because it captures interaction between sub-goals better than simple counters.

11.3 Graphplan vs. RPG

It is important to distinguish the RPG from the original **Graphplan** algorithm:

- **Graphplan:** Builds a full planning graph including *mutex* (mutual exclusion) constraints to handle negative interactions. It searches for a valid plan in this complex structure. It is exact but slower.

- **RPG (in FF):** Builds a *relaxed* graph (ignores delete effects and mutexes). It is an approximation used solely to compute a heuristic value quickly.

Chapter 12

Pattern Databases

12.1 The Strategic Pivot: Pre-computation

In the previous chapters, you focused on heuristics calculated dynamically during the search, like h_{FF} derived from the Relaxed Planning Graph. While effective, these heuristics suffer from a redundancy problem: they re-solve similar relaxations over and over again for every visited node.

Pattern Databases (PDBs) represent a strategic shift. Instead of computing the heuristic on the fly, we solve a simplified version of the problem (a sub-problem) optimally offline, store the results in a lookup table (the database), and then simply query this table during the actual search. We are effectively trading memory (to store the database) for speed (instant heuristic retrieval).

The core intuition is simple: if you want to solve a Rubik's cube, you might first learn to solve just the corners perfectly. That knowledge is a "pattern" you can apply regardless of the state of the other pieces.

12.2 Formalizing Abstraction

To implement this, we cannot simply "ignore" parts of the problem arbitrarily; we must do so systematically to guarantee admissibility. This process is called Abstraction.

Given a planning problem with a set of state variables, a Pattern is a subset of these variables that we decide to track. We systematically ignore everything else. This creates an Abstract State Space. For example, in the Blocks World with blocks A, B, C, and D, we might decide to only care about block A. Our pattern P contains only facts involving A (e.g., `on(A, ?)`, `clear(A)`). This induces a mapping ϕ from the original state space S to the abstract state space S'.

$$\phi : S \to S' \tag{12.1}$$

Every concrete state s maps to an abstract state s' which is simply the intersection of s with the pattern P. The crucial property is that the abstract state space is exponentially smaller than the original one, making it solvable.

12.3 Relaxation through Abstraction

Is a PDB heuristic admissible? Yes, because abstraction is a form of Relaxation. When we abstract a problem, we are essentially removing preconditions and effects related to the variables we ignore.

- If an operator o changes a variable inside the pattern, it remains an operator in the abstract space.
- If o only changes variables outside the pattern, it becomes a "self-loop" or identity transition in the abstract space (often ignored).

Any valid plan in the original state space corresponds to a valid path in the abstract state space. Therefore, the optimal cost in the abstract space $h^*(s')$ is a lower bound on the optimal cost in the original space $h^*(s)$.

$$h_{PDB}(s) = h^*_{Abstract}(\phi(s)) \leq h^*(s) \tag{12.2}$$

12.4 Construction: The Backward Search

You do not build a PDB by running forward searches. That would be inefficient. The database is constructed once via a Backward Breadth-First Search (or Dijkstra) starting from the abstract goal states.

1. Identify all abstract states that satisfy the goal conditions relevant to the pattern. These have a heuristic value of 0.
2. Perform a backward search in the abstract space to find the optimal distance from every reachable abstract state to the set of abstract goal states.
3. Store these distances in a table indexed by the abstract state.

During the actual search, when you encounter state s, you calculate its abstract index $\phi(s)$, look it up in the table, and get the value immediately.

12.5 The Additivity Trap and Partitioning

A single pattern is rarely informative enough. If you only look at block A, the heuristic will tell you nothing about the mess block B is in. To get a strong heuristic, we need multiple patterns (e.g., one for A, one for B, etc.).

However, you cannot simply sum the heuristic values of different patterns. If pattern P_1 requires moving block C to free A, and pattern P_2 requires moving block C to free B, summing them would count the cost of moving C twice. This violates admissibility.

To sum PDBs admissibly, we must ensure Additivity. This leads to the concept of Disjoint Pattern Databases or Cost Partitioning. The strict rule for additive PDBs is:

- The patterns must be disjoint (no shared variables).
- Every operator in the domain must affect variables in at most one pattern.

If these conditions are met, the sub-problems are independent regarding the cost, and their heuristic values can be summed:

$$h(s) = \sum_i h_{PDB_i}(s) \tag{12.3}$$

This is significantly more powerful than taking the maximum ($\max_i h_{PDB_i}(s)$), which is the standard way to combine admissible heuristics that overlap.

12.6 Performance and Limitations

PDBs are not a silver bullet.

- Memory Bottleneck: The size of the database grows exponentially with the size of the pattern. If you include too many variables, you run out of RAM before you even start planning.
- The Granularity Problem: If patterns are too small, the heuristic is weak (low accuracy). If they are too large, they are uncomputable.
- Pattern Selection: Deciding *which* variables to group into a pattern is a hard optimization problem in itself. Automated pattern selection is a key area of research (e.g., using genetic algorithms or hill climbing to find good patterns).

In conclusion, PDBs move the complexity from the online search phase to the offline pre-computation phase. They are ideal when you need to solve many instances of the same domain or when the state space structure allows for clean partitioning into independent sub-problems.

Chapter 13

Planning in Non-Deterministic Domains via Model Checking

Classical planning relies on restrictive assumptions: the world is finite, time is implicit, the initial state is fully known, and most importantly, actions are **deterministic**. In classical planning, if an agent is in state s and executes action a, the outcome is a unique, predictable state s'.

However, real-world domains are rarely this cooperative. We face: uncertainty in the Initial State, non-deterministic actions, the environment may change independently of the agent and partial observability.

In such domains, a linear sequence of actions is insufficient because the agent cannot predict the exact trajectory of the world. Instead of a sequence, we must synthesize a **Plan as a Controller** (or Policy). This is a function that maps observations (or states) to actions, allowing the agent to react to feedback from the environment.

13.1 Planning as Model Checking (PMC)

Model Checking is a verification technique used to determine if a finite-state model of a system satisfies a given logical specification (usually in Temporal Logic).

$$\mathcal{M} \models \varphi$$

Where $\mathcal{M}$ is the system model and φ is the property.

Planning as Model Checking inverts this paradigm. Instead of verifying a system, we want to *synthesize* a controller that forces the system to satisfy a goal.

- **Domain ($\mathcal{D}$):** A Non-Deterministic Finite State Machine (FSM).
- **Goal ($\mathcal{G}$):** A Temporal Logic formula (e.g., ”Eventually reach state G”).
- **Plan (Π):** A state-action table or automaton that restricts the behavior of $\mathcal{D}$ such that $\mathcal{G}$ is satisfied.

13.2 Symbolic Representation

Classical explicit-state search (like A*) fails in non-deterministic domains because the state space branches on outcomes, not just actions, leading to a double exponential explosion. To handle this, PMC uses **Symbolic Representation**.

13.2.1 Binary Decision Diagrams (BDDs)

States and transitions are not stored explicitly. Instead, they are encoded as Boolean formulas and stored using **Binary Decision Diagrams (BDDs)**.

- A set of states S is represented by a boolean formula $\xi(S)$.
- Transitions are represented by a relation $\xi(T(s, a, s'))$.

- Set operations (Union, Intersection, Difference) correspond to logical operations (OR, AND, AND NOT) on BDDs.

This allows the planner to manipulate massive sets of states (e.g., 10^{20} states) efficiently in a single computational step.

13.3 Planning Algorithms for Reachability Goals

When the domain is non-deterministic, the concept of "solution" splits into different grades of strength. We aim to reach a set of goal states G.

13.3.1 1. Weak Solutions (Possible Plans)

A plan is a **Weak Solution** if there exists *at least one* execution trace that reaches the goal. This is optimistic planning: "If I am lucky, this will work."

- **Technique:** Similar to standard search but finding any path in the non-deterministic graph.

13.3.2 2. Strong Solutions (Guaranteed Plans)

A plan is a **Strong Solution** if *all* possible execution traces are guaranteed to reach the goal, regardless of non-deterministic outcomes. This is loop-free execution.

- **Algorithm (StrongPlan):** This is a backward fixed-point computation.
- We compute the **Strong Pre-Image**: The set of states from which an action a is guaranteed to lead *only* into the current accumulated target set.
- $SA_{new} = \{(s, a) \mid \forall s'.T(s, a, s') \implies s' \in \text{Target}\}$
- We iterate backward from the goal G until we cover the initial state I.

13.3.3 3. Strong Cyclic Solutions (Trial and Error)

In many scenarios, strong solutions do not exist (e.g., trying to pick up a slippery block; you might fail infinitely many times in theory). A **Strong Cyclic Solution** guarantees reaching the goal under the assumption of **fairness**: if a non-deterministic outcome is possible, it will eventually happen if retried enough times.

- The plan may contain loops (trial and error).
- **Algorithm:** A global nested fixpoint computation. It iteratively prunes states that cannot reach the goal and actions that might lead to "bad" cycles (cycles that cannot exit to the goal).

13.4 Partial Observability

Real robots rarely know the exact state of the world. They only receive **Observations**.

- **Belief State:** The set of all physical states the agent *might* be in, given the history of actions and observations.
- **Search Space:** The planner searches in the space of Belief States (AND-OR search).
  - **OR node:** Choice of action by the agent.
  - **AND node:** Non-deterministic branching based on possible observations from the environment.

13.4.1 Conformant Planning

A special case of partial observability is **Null Observability**. The agent receives no feedback at all. A **Conformant Plan** is a sequence of actions that guarantees the goal is reached regardless of the initial state or action outcomes. This is achieved by finding a sequence that coerces the belief state into a subset of the goal states.

Chapter 14

Temporal Planning

Classical planning relies on a comfortable lie: that actions are atomic, instantaneous transitions ($\gamma : S \times A \to S$). In the real world (robotics, logistics, space operations), actions take time, they overlap, and they must be synchronized. **Temporal Planning** addresses the questions: "What to do?", "Which parameters?", and critically, "When?".

The core challenges introduced by time are:

- **Durative Actions:** Actions have a start time and an end time.
- **Concurrency:** Actions may execute in parallel. This is not just optimization; it is often a requirement (e.g., holding a door open while passing through).
- **External Events:** Windows of opportunity (e.g., satellite visibility) open and close independently of the agent.

14.1 Temporal Reasoning Frameworks

Before we can plan, we must be able to reason about time constraints. We treat time points or intervals as variables in a constraint network.

14.1.1 Interval Algebra (Allen's Algebra)

Defined by James Allen, this algebra reasons about the qualitative relations between time intervals. There are 13 basic relations (e.g., *before, meets, overlaps, starts, during, finishes*, and their inverses, plus *equal*). While expressive, full reasoning in Allen's algebra is computationally heavy.

14.1.2 Point Algebra

Often, it is sufficient to reason about Time Points (the start or end of an action). The relations are simple: $<, =, >, \leq, \geq, \neq$.

14.1.3 Simple Temporal Networks (STN)

This is the standard mathematical structure used in temporal planning. An STN is a graph where:

- **Nodes** are time points (t_i).
- **Edges** represent constraints on the distance between points: $a \leq t_j - t_i \leq b$.

Checking the consistency of an STN (i.e., does a valid schedule exist?) is equivalent to finding if there are no **negative cycles** in the corresponding distance graph. This can be solved efficiently using the Bellman-Ford or Floyd-Warshall algorithms ($O(N^3)$).

14.2 Planning Approaches

We distinguish two main philosophies in temporal planning: Time-Oriented (Plan Space) and State-Oriented.

14.2.1 Time-Oriented Approaches (Plan Space)

This approach extends POCL (Partial Order Causal Link) planning. Instead of simple literals, we manage **Temporally Qualified Expressions (TQEs)**.

- A TQE has the form $p(args)@[t_{start}, t_{end})$.
- The planner searches for a plan by resolving "flaws" (open goals, threats) by adding causal links and temporal constraints to a database.
- **TPS (Temporal Planning System):** An algorithm that refines a partial plan until all temporal and logical constraints are met.

14.2.2 State-Oriented Approaches

This is the dominant approach in modern planners (like TFD, OPTIC). It adapts forward state-space search to handle time. The "state" must now include temporal information.

The PDDL 2.1 Model

Actions are no longer atomic. A **Durative Action** has:

- **Duration:** Fixed or variable.
- **Conditions:** `at start`, `at end`, or `over all` (invariants).
- **Effects:** `at start` or `at end`.

Snap Actions

To handle concurrency using standard search algorithms, we decompose a durative action A into two instantaneous "snap" actions:

- $start(A)$: Apply start effects, check start conditions.
- $end(A)$: Apply end effects, check end conditions.

The planner must ensure that every $start(A)$ is eventually followed by an $end(A)$, respecting the duration constraints.

Decision Epochs (TFD Approach)

In planners like Temporal Fast Downward (TFD), the state contains a timestamp t and an **agenda** of currently executing actions.

- Transitions can either start a new action or **advance time** to the next event in the agenda (the end of an active action).
- Limitation: This approach is theoretically incomplete because it cannot start actions at arbitrary times, only at the "decision epochs" (start/end of other actions).

Symbolic STN in States (OPTIC Approach)

To overcome the incompleteness of decision epochs, advanced planners (like OPTIC or POPF) embed a **Simple Temporal Network within each state**.

- The state is a tuple $\langle \text{Facts}, \text{STN}, \text{Agenda} \rangle$.
- Time advancement is implicit. The STN tracks the constraints between all start and end points added so far.
- A state is valid only if the internal STN is consistent (no negative cycles).

14.3 Planning as SMT

An alternative to heuristic search is encoding the temporal planning problem as a logical formula in **Satisfiability Modulo Theories (SMT)**.

- We use variables for action occurrences and real-valued variables for their timestamps.
- Theories like Linear Real Arithmetic (LRA) handle the temporal constraints.
- If the SMT solver finds a model, that model is a valid schedule.

Chapter 15

PlanSys2 - Robotic Planning System

15.1 From Theory to Systems

Until now, we have treated planning as an isolated algorithmic problem: given I and G, find π. In robotics, this is insufficient. We need a **Planning System**: a software infrastructure capable not just of generating plans, but of managing knowledge, executing actions, monitoring progress, and handling failures in a dynamic environment.

Before PlanSys2, several systems attempted to bridge this gap:

- **CORTEX:** A cognitive architecture, tightly coupled and lacking temporal planning support.
- **SkiROS2:** Powerful, based on Behavior Trees and Ontologies, but complex.
- **ROSPlan:** The standard for ROS1. It introduced the modular pipeline (Knowledge Base $\rightarrow$ Problem Interface $\rightarrow$ Planner $\rightarrow$ Dispatch). However, it suffered from the limitations of ROS1 (no real-time, single master) and lacked a clean API for multi-robot coordination.

15.2 PlanSys2: The ROS2 Evolution

PlanSys2 is the successor to ROSPlan, re-engineered for **ROS2**. It is designed for efficiency, reliability, and security. Key architectural features include:

1. **ROS2 & DDS:** Leveraging the Data Distribution Service for communication allows for zero-configuration networking and real-time capabilities.
2. **Lifecycle Nodes (Managed Nodes):** Every component in PlanSys2 has a strictly defined state machine (Unconfigured, Inactive, Active, Finalized). This ensures deterministic startup and shutdown behavior, crucial for safety-critical robotics.
3. **Modular Architecture:** Components communicate via clearly defined ROS2 interfaces, allowing easy substitution of solvers or executors.

15.3 System Architecture

PlanSys2 is composed of four primary nodes that replicate the cognitive cycle:

15.3.1 1. Domain Expert

Responsible for the static knowledge. It parses the PDDL Domain file (types, predicates, actions). In PlanSys2, the domain can be distributed: different nodes can contribute different parts of the PDDL model, which are merged at runtime.

15.3.2 2. Problem Expert

Responsible for the dynamic knowledge (the state). It manages:

- **Instances:** The objects in the world.
- **Predicates/Functions:** The current state of the world (s_0).
- **Goals:** The desired state (G).

It provides services to add, remove, or update these elements dynamically (e.g., via perception updates).

15.3.3 3. Planner

This node wraps the PDDL solver. It takes the domain from the Domain Expert and the problem from the Problem Expert to generate a plan.

- **Solvers:** It supports POPF (default) and TFD (Temporal Fast Downward). Support for OPTIC is planned.
- **Namespaces:** It can handle multiple planning instances simultaneously (e.g., for multi-robot simulation) by using distinct namespaces.

15.3.4 4. Executor

The most complex component. It is responsible for taking the symbolic plan and executing it in the real world.

- **Action Auctioning:** When an action needs to be executed (e.g., `move`), the Executor broadcasts a request. Available Action Performers (ROS2 nodes implementing the action) "bid" to execute it. This natively supports multi-robot fleets where multiple robots can perform the same type of action.
- **Plan Optimization:** It analyzes the plan structure to parallelize actions where possible, even if the original PDDL plan was sequential (provided no causal links are violated).

15.4 The Core Innovation: From Plan to Behavior Tree

A raw PDDL plan is fragile. PlanSys2 converts the PDDL plan into a **Behavior Tree (BT)** before execution. This allows for reactive control and robust monitoring.

15.4.1 The Conversion Process

1. **Dependency Graph:** The Executor first builds a directed graph of the plan.
   - If Action A produces an effect needed by Action B, a causal link $A \to B$ is established.
   - Temporal constraints are respected.
2. **Flow Extraction:** The graph is analyzed to identify independent "execution flows".
3. **BT Construction:**
   - **Sequences ($\to$):** Used when strict causal dependencies exist.
   - **Parallel Nodes ($=$):** Used when independent flows can run concurrently.

Each durative PDDL action is expanded into a specific BT subtree that explicitly checks:

- `at start` requirements before starting.
- `over all` invariants while running.
- `at end` effects upon completion.

15.5 Using PlanSys2

15.5.1 The Terminal

PlanSys2 provides a powerful CLI (`plansys2_terminal`) for interacting with the system during development:

- `get domain/problem`: Inspect the current model.
- `set instance/predicate/goal`: Modify the state or goal manually.
- `run`: Triggers the planner and immediately starts execution.

15.5.2 Developing Action Performers (C++)

To make the robot actually *do* something, you must implement Action Performers. PlanSys2 provides the `ActionExecutorClient` class. A developer must implement the `do_work()` method, which is called cyclically (e.g., at 4Hz):

- **get_arguments():** Retrieve PDDL parameters (e.g., `robot1`, `kitchen`).
- **send_feedback():** Report progress (0.0 to 1.0) to the Executor.
- **finish():** Report success or failure when the action concludes.

This structure enforces a clean separation between the *deliberative layer* (PlanSys2) and the *reactive layer* (ROS2 nodes), allowing for robust and scalable robotic autonomy.