Part 3: Beyond Classical SSPs

Dead-Ends, Dual Criteria, and Safety Properties

Recall: Classical SSP Assumptions

The SSP framework from Part 1 rests on key assumptions:

1. **Goal reachability**: There exists at least one **proper policy** — a policy under which the goal is reached with probability 1 from every reachable state
2. **Finite expected cost**: Every proper policy has finite expected cost
3. **Positive costs on non-goal states**: $C(s, a) > 0$ for all $s \notin \mathcal{G}$, ensuring no zero-cost cycles

> ⚠ **But what if these assumptions fail?**
> Many real-world domains contain states from which the goal is simply unreachable — no matter what the agent does.

Dead-End States: Motivation

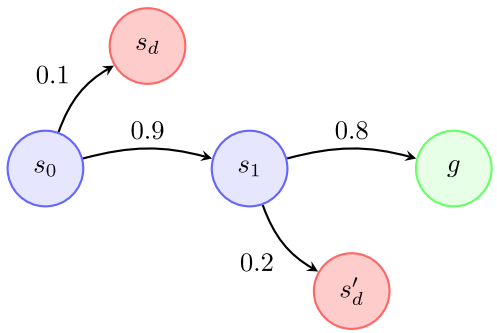
Real domains frequently contain **dead-end states** — states from which the goal is unreachable regardless of the policy.

Triangle Tireworld

A robot drives between locations. Moving can cause a flat tire. If there is no spare at the current location, the robot is stuck — a dead-end.

Navigation with one-way bridges

A bridge can collapse after crossing. If the only path to the goal used that bridge, the agent is trapped.



Even an optimal policy cannot avoid the possibility of reaching s_d or s'_d — the dead-ends are **unavoidable**.

Dead-End Classification

Definition

A state s is a **dead-end** if no policy can reach the goal from s:

$$\max_{\pi} \Pr^{\pi}(\Diamond \mathcal{G} \mid s_0 = s) = 0$$

Dead-ends come in two flavors:

Type	Definition	Example
Avoidable	A dead-end that an optimal policy never visits	Choosing a safe route avoids the trap
Unavoidable	A dead-end that ANY policy may reach with positive probability	Stochastic transitions can force entry

Warning

Why Classical SSP Fails with Dead-Ends

When dead-end states exist, the classical SSP framework breaks down:

1. **No proper policy exists** — no policy can guarantee reaching the goal with probability 1
2. **Infinite values**: $V^*(s) = +\infty$ for dead-end states under the SSP cost criterion, since the agent accumulates cost forever
3. **Non-unique Bellman solutions**: The Bellman equations may admit multiple fixed points (or no finite fixed point at all)
4. **Algorithm divergence**: Classical VI may diverge; LRTDP's convergence guarantees rely on the proper policy assumption

> ✦ **Key insight**
> We need new optimization criteria that remain well-defined even when goal reachability cannot be guaranteed.

Heuristics for SSPs with Dead-Ends

The STRIPS relaxation heuristics h^+_{max} and h^+_{add} fail when dead-ends exist.

⚠ The problem

$h^+_{max}(s) = +\infty$ for dead-end states → infinite values propagate to predecessors → heuristic search **fails** (undiscounted Bellman eq. diverges).

🕮 Solution: Discounted SSP heuristics

Use a **discount factor** $\gamma < 1$ to keep all values finite. Define h^γ_X from h^+_X:

$$h^\gamma_X(s) = c_m(s) \cdot \frac{1 - \gamma^{h^{1,+}_X(s)}}{1 - \gamma}$$

- $c_m(s)$: lower bound on minimum non-zero transition cost from s
- $h^{1,+}_X(s)$: h^+_X computed with all non-zero costs set to 1 (minimum steps to goal; $= +\infty$ for dead-ends)

✦ Theorem

h^γ_{max} is **admissible** for all goal-oriented MDPs — with or without dead-ends. h^γ_{add} is not admissible but more informative. When $\gamma = 1$ and no dead-ends: $h^\gamma_X = h^+_X$.

Optimization Criteria for SSPs with Dead-Ends

MAXPROB: Maximize Goal Probability

MAXPROB criterion

Find a policy that maximizes the probability of reaching the goal:

$$V^*_{\text{maxprob}}(s) = \max_{\pi} \Pr^{\pi}(\Diamond \mathcal{G} \mid s_0 = s)$$

Bellman equation for MAXPROB (with $V(s) = 1$ for $s \in \mathcal{G}$ and $V(s) = 0$ for dead-ends):

$$V(s) = \max_{a \in A(s)} \sum_{s'} T(s' \mid s, a) \cdot V(s')$$

- Solved with **Value Iteration** (maximizing instead of minimizing)
- Ignores costs entirely — only cares about reaching the goal
- A dead-end state s has $V^*_{\text{maxprob}}(s) = 0$
- **Special case of GSSP** where all actions have reward 0 except reaching goal (reward 1)

Limitation

Among policies that achieve the same goal probability, MAXPROB does not distinguish by cost — it might pick an arbitrarily expensive policy.

Kolobov, Mausam, Weld, Geffner. "Heuristic Search for Generalized SSP MDPs." ICAPS 2011.

MinCost-MaxProb: Dual Criterion

MinCost-MaxProb (aka S3P)

Among all policies that maximize goal probability, find the one that minimizes expected cost **only over paths that reach the goal**:

$$\pi^* = \arg\min_{\pi} \mathbb{E}^{\pi}\left[\sum_{t} C(s_t, a_t) \mid \Diamond\mathcal{G}\right] \quad \text{subject to} \quad \overset{\pi}{\Pr}(\Diamond\mathcal{G}) = V^*_{\text{maxprob}}(s_0)$$

Extends MAXPROB by adding cost minimization as a secondary criterion.

Two-phase approach (GPCI algorithm):

1. **Phase 1**: Solve MAXPROB to find $V^*_{\text{maxprob}}(s)$ for all states
2. **Phase 2**: Restrict to policies achieving maximal goal probability; among these, minimize expected cost averaged only over goal-reaching paths

Intuition

"Do your best to reach the goal. If multiple policies are equally likely to succeed, pick the cheapest one."

Teichteil-Königsbuch. "Stochastic Safest and Shortest Path Problems." AAAI 2012.

Live Demo: GPCI (MinCost-MaxProb)

Watch GPCI solve a maze with unavoidable dead-ends

```python
# algorithm: gpci
# speed: 100
# GPCI on a maze with UNAVOIDABLE DEAD-ENDS
# Trap cells (red !): massive cost penalty
# Ice cells (cyan ~): slippery transitions
# Phase 1: MAXPROB to find best goal probability
# Phase 2: Min cost among max-probability policies
from skdecide.hub.solver.gpci.gpci import GPCI
from stochastic_maze import DeadEndMaze
from skdecide.core import Value

maze = DeadEndMaze()
with GPCI(
    domain_factory=lambda: DeadEndMaze(),
    heuristic=lambda d, s: Value(
        cost=abs(d._goal.x - s.x)
            + abs(d._goal.y - s.y)),
    discount=1.0,
    epsilon=0.001,
) as solver:
    solver.solve()
```

❌ Backend not running. Start: cd backend && uv run

Reset | Stop | Speed | Fast | Slow

Ready — click ▶ Run on the code

Generalized SSPs (GSSPs)

GSSP: A General Framework

Relaxes SSP's strict reward restrictions: allows arbitrary real-valued rewards (including 0-reward cycles).

Optimizes only among proper policies (policies that reach the goal with probability 1):

$$\pi^* = \arg\max_{\pi \text{ proper}} V^\pi(s_0) \quad \text{where} \quad V^\pi(s) = \mathbb{E}^\pi\left[\sum_t R(s_t, a_t)\right]$$

Key challenge: With 0-reward cycles, the Bellman backup operator has **multiple fixed points**:

- Standard algorithms (VI, LAO*, LRTDP) initialized admissibly can converge to **suboptimal fixed points**
- Even when V^* exists, greedy policies w.r.t. V^* may not be proper (they might loop forever)

GSSP subsumes

SSP, MAXPROB, positive-bounded, negative, and discounted MDPs as special cases of GSSP.

FRET: Find, Revise, Eliminate Traps

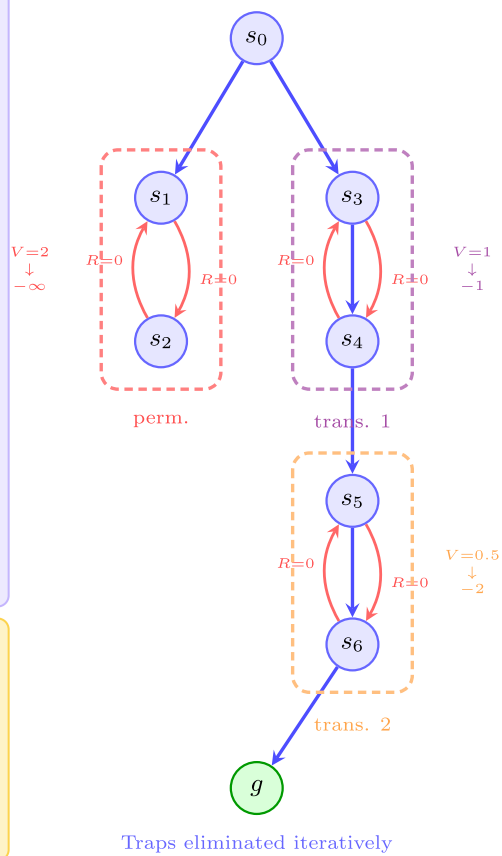
First heuristic search algorithm for GSSPs (SSPs with 0-reward cycles)

{} Algorithm: FRET

- **Repeat** until convergence to V^*:
  1. **FIND & REVISE**: Run heuristic search (LRTDP) with Bellman backups until convergence to $V_i' = \mathcal{B}V_i'$
  2. **ELIMINATE TRAPS**: Identify SCCs in greedy graph $G^{V_i'}$ with no path to goal
     - **Permanent trap** (no exit action): $V(s) = -\infty$ for all $s \in S_C$
     - **Transient trap** (exit exists): $V(s) = \max_{s \in S_C, a \in A_e} Q^{V_i'}(s, a)$
  3. Set $V_{i+1} \leftarrow$ updated values (still admissible: $V_{i+1} \geq V^*$)

✦ Escaping suboptimal fixed points

With 0-reward cycles, Bellman backup has **multiple fixed points**. Standard algorithms (LAO*, LRTDP) initialized admissibly can converge to suboptimal ones. FRET detects **traps** (SCCs with no greedy exit) and forces values down, escaping bad fixed points while preserving admissibility.



Traps eliminated iteratively

FRET: Key Properties and Lemmas

✦ **Lemma 1: Zero-reward intra-trap actions**

If $V = BV$ (Bellman fixed point) and $C = \{S_C, A_C\}$ is a transient trap, then **all actions transitioning within C have reward 0**: $\forall s \in S_C, \forall a : T(s, a, s') > 0 \wedge s' \in S_C \Rightarrow R(s, a) = 0$

Why traps are problematic: Moving within a trap is "free" (0 cost) → multiple suboptimal Bellman fixed points exist.

🕮 **Trap elimination preserves admissibility**

- **Permanent traps**: No policy can reach goal → $V^*(s) = -\infty$ → safe to set $V(s) = -\infty$
- **Transient traps**: Proper policy exists but not greedy → set $V(s)$ to best Q-value of **exit actions** (actions that can leave the trap)

Both operations preserve $V \geq V^*$ (admissibility).

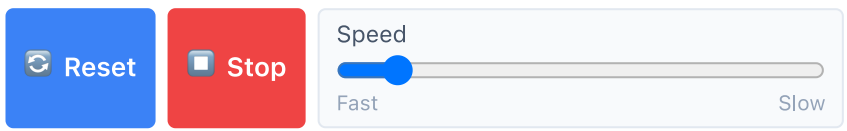
✦ **Theorem (Kolobov et al., 2011)**

Live Demo: FRET

Watch FRET handle a maze with unavoidable dead-ends

```python
# algorithm: fret
# speed: 100
# FRET on a maze with UNAVOIDABLE DEAD-ENDS
# Trap cells (red !): massive cost penalty
# Ice cells (cyan ~): slippery transitions
# FRET eliminates traps iteratively
from skdecide.hub.solver.fret import FRET
from stochastic_maze import DeadEndMaze
from skdecide.core import Value

maze = DeadEndMaze()
with FRET(
    domain_factory=lambda: DeadEndMaze(),
    heuristic=lambda d, s: Value(
      cost=abs(d._goal.x - s.x)
        + abs(d._goal.y - s.y)),
    inner_solver_factory=lambda: ("LRTDP", {}),
    discount=1.0,
    epsilon=0.001,
    dead_end_cost=10000.0,
) as solver:
    solver.solve()
```



Ready — click ▶ Run on the code

The i-dual Approach

i-dual

Formulates SSPs with dead-ends as a **linear program** using the dual of the occupation measure formulation.

Recall the LP formulation of MDPs using **occupation measures** $x(s, a)$:

$$\min \sum_{s,a} C(s, a) \cdot x(s, a) \quad \text{s.t. flow constraints on } x(s, a)$$

The i-dual approach:

- Extends the LP dual to handle dead-ends correctly
- Can encode both **MAXPROB** and **MinCost-MaxProb** criteria
- Connects probabilistic planning to **operations research** (LP duality, network flows)
- Enables the use of mature LP solvers (CPLEX, Gurobi)

iDual for constrained SSPs

The iDual algorithm (Part 2) extends naturally to constrained SSPs: the LP/MILP formulation accommodates probability constraints, PCTL specifications, and unavoidable dead-ends by adding linear constraints on the occupation measures.

Summary: Criteria for SSPs with Dead-Ends

Criterion	Objective	Key Feature
SSP	Min expected cost	Requires proper policy; no 0-reward cycles
GSSP	Max expected reward	Allows arbitrary rewards (including 0-cycles); optimizes only over proper policies
MAXPROB	Max $\Pr(\Diamond \mathcal{G})$	Special case of GSSP; ignores costs
MinCost-MaxProb (S3P)	Min cost given max $\Pr$	Two-phase: maximize probability, then minimize cost

Algorithm	Approach	Criteria Supported
VI / LRTDP (adapted)	Iterative Bellman	SSP, MAXPROB
FRET	Trap elimination + Bellman	**GSSP** (including MAXPROB)
GPCI	Two-phase (maxprob + cost)	**MinCost-MaxProb**
i-dual	Linear programming	MAXPROB, MinCost-MaxProb

✦ **Takeaway**

Safety and PCTL Properties

Beyond Goal Reachability

So far, all criteria focus on **reaching the goal**. But in practice, we often care about the **path** taken:

- **Avoid dangerous states** along the way (no-fly zones, collisions)
- **Maintain invariants** with high probability (battery level, structural integrity)
- **Satisfy temporal constraints** on execution (visit A before B)

Example

Autonomous vehicle: Must reach the destination **AND** maintain $\Pr(\text{collision}) < 0.001$ at every decision step.

UAV mission: Reach target area while ensuring $\Pr(\text{entering no-fly zone}) < 0.01$.

Medical treatment: Administer therapy to cure the patient while keeping $\Pr(\text{adverse reaction}) < 0.05$.

Need

A formalism that combines optimization (minimize cost) with probabilistic constraints on the execution path.

PCTL: Probabilistic Computation Tree Logic

PCTL (Hansson & Jonsson, 1994)

Extends CTL with probabilistic quantifiers: $\mathcal{P}_{\bowtie p}[\phi]$

where $\bowtie \in \{<, \leq, \geq, >\}$, $p \in [0, 1]$, and ϕ is a path formula.

Key path formulas:

Formula	Meaning
$\mathcal{P}_{\geq 0.95}[\Diamond\, \mathcal{G}]$	Reach goal with probability ≥ 0.95
$\mathcal{P}_{\leq 0.01}[\Diamond\, \text{unsafe}]$	Visit an unsafe state with probability ≤ 0.01
$\mathcal{P}_{\geq 0.99}[\text{safe}\, \mathcal{U}\, \mathcal{G}]$	Remain safe until reaching goal, with prob ≥ 0.99
$\mathcal{P}_{\geq 0.9}[\Box\, \text{stable}]$	Always remain in stable states, with prob ≥ 0.9

Theorem

PCTL allows us to **specify** probabilistic safety requirements formally, providing a bridge between the planning and **model checking** communities.

Constrained SSPs

Constrained SSP

Minimize expected cost subject to probabilistic safety constraints:

$$\min_{\pi} \mathbb{E}^{\pi}\left[\sum_{t} C(s_t, a_t)\right] \quad \text{s.t.} \quad \Pr^{\pi}(\Diamond \text{unsafe}) \leq \delta$$

This combines **optimization** (SSP cost minimization) with **verification** (PCTL constraint satisfaction).

Challenges:

- The constraint is **global** — it depends on the entire trajectory, not just individual states
- Standard Bellman equations do not capture probability constraints directly
- The feasible policy space is no longer convex in general
- May require **randomized** (stochastic) policies to achieve optimality

Warning

Adding even a single probability constraint can make the problem significantly harder than unconstrained SSP.

Teichteil-Koenigsbuch, F. "Stochastic Safest and Shortest Path Problems." AAAI 2012.

Solution Approaches for Constrained SSPs

State Augmentation

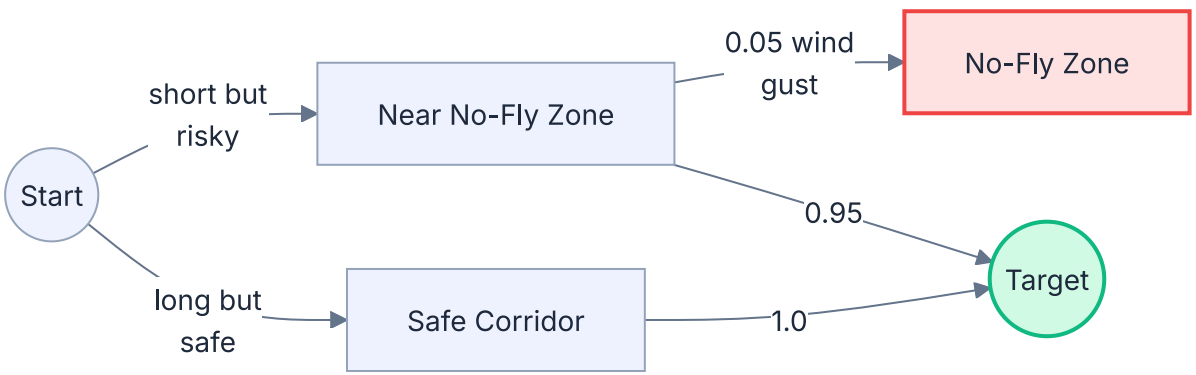
- Augment state with **remaining probability budget**: $s' = (s, p_{\text{remaining}})$
- $p_{\text{remaining}}$ tracks how much violation probability is still "allowed"
- Reduces to an unconstrained SSP on a larger state space

> ✦ **Theorem**
> **Exact** but state space grows significantly: $|\mathcal{S}'| = |\mathcal{S}| \times |\text{budget discretization}|$

Practical Example: UAV Mission Planning

Scenario

A UAV must reach a target area while maintaining $\Pr(\text{entering no-fly zone}) < 0.01$.



Constrained SSP policy behavior:

- **Far from no-fly zones**: take cost-efficient shortcuts (low risk)
- **Near no-fly zones**: switch to safer but longer routes (risk budget running low)
- The policy **adapts** based on the remaining probability budget

Theorem

Connection to Probabilistic Model Checking

Model Checking Tools

- **PRISM**, **Storm**: verify PCTL properties on MDPs
- Can synthesize optimal policies w.r.t. PCTL specifications
- Require **full state space** enumeration
- Very expressive: nested PCTL, reward properties, multi-objective

Planning Community Contribution

- **Heuristic search** + PCTL = scalable constrained planning
- Avoid full state enumeration via forward search (LAO*, LRTDP)
- Domain-specific heuristics prune the search space
- Handle much larger state spaces than model checkers

✦ **Complementary strengths**

Model checking offers expressiveness and formal guarantees; heuristic search planning offers scalability. Bridging these communities is an active and fruitful research direction.

Temporal Logic Specifications for SSPs

Beyond simple goal reachability

LTL for SSPs

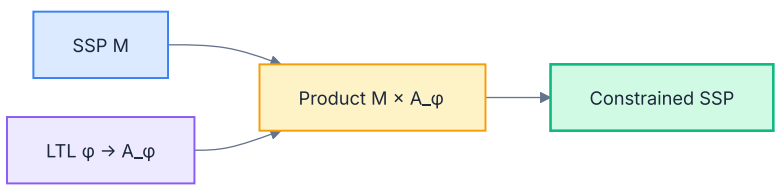
Replace simple goal $\mathcal{G}$ with an **LTL** specification φ:

$$\min_{\pi} \mathbb{E}^{\pi}\left[\sum_{t} C(s_t, a_t)\right] \text{ s.t. } \overset{\pi}{\Pr}(s_0 s_1 \ldots \models \varphi) \geq \theta$$

Example specifications:

- $\Diamond \text{goal}$ — eventually reach goal (standard SSP)
- $\Box \neg \text{unsafe} \wedge \Diamond \text{goal}$ — always avoid unsafe
- $\Diamond \text{checkpoint} \wedge \Diamond \text{goal}$ — visit checkpoint then goal

Solution: product MDP



Product MDP construction

The **product MDP** $\mathcal{M} \times \mathcal{A}_{\phi}$ augments each state (s, q) with the automaton state q tracking LTL progress. Actions inherit transitions from both the MDP and the automaton. This transforms temporal constraints into reachability in the augmented space.

Part 3: Summary

Key References for Part 3

Generalized SSPs and algorithms:

- Kolobov, A., Mausam, Weld, D., & Geffner, H. "Heuristic Search for Generalized Stochastic Shortest Path MDPs." *ICAPS*, 2011. (GSSP, FRET, MAXPROB)
- Teichteil-Koenigsbuch, F. "Stochastic Safest and Shortest Path Problems." *AAAI*, 2012. (S3P, MinCost-MaxProb, GPCI)
- Trevizan, F., Thiebaux, S., & Haslum, P. "Occupation Measure Heuristics for Probabilistic Planning." *AAAI*, 2017. (i-dual)

Heuristics for dead-ends:

- Teichteil-Koenigsbuch, F., Vidal, T., & Infantes, G. "Extending Classical Planning Heuristics to Probabilistic Planning with Dead-Ends." *AAAI*, 2011.

Safety and constrained planning:

- Hansson, H. & Jonsson, B. "A Logic for Reasoning about Time and Reliability." *Formal Aspects of Computing*, 1994.
- Sprauel, J., Kolobov, A., & Teichteil-Koenigsbuch, F. "Saturated Path-Constrained MDP." *AAAI*, 2014.
- Baier, C. & Katoen, J.-P. *Principles of Model Checking*. MIT Press, 2008.

Part 4: Modeling Languages & the IPPC

From domain descriptions to competition benchmarks

PPDDL: Extending PDDL with Probability

> ⓘ **Note**
>
> **Recall**: Classical PDDL separates **domain** (types, predicates, deterministic actions) from **problem** (objects, init, goal). Every action produces exactly one successor state — no stochastic outcomes.

PPDDL — Probabilistic PDDL — extends classical PDDL with three key additions:

1. **Probabilistic effects**: actions can have multiple outcomes, each with a specified probability
2. **Reward fluents**: numerical rewards or costs can be accumulated during execution
3. **Goal-based optimization**: maximize expected total reward (or equivalently minimize expected cost)

PPDDL was introduced by Younes & Littman (2004) and served as the standard language for the IPPC from 2004 to 2008.

PPDDL Syntax Overview

Domain file — types, predicates, actions:

```
(define (domain blocksworld)
  (:requirements :probabilistic-effects)
  (:types block)
  (:predicates
    (on ?b1 - block ?b2 - block)
    (ontable ?b - block)
    (clear ?b - block)
    (holding ?b - block)
    (handempty))
  (:action pick-up ...)
  (:action put-down ...)
  (:action stack ...)
  (:action unstack ...))
```

The `:probabilistic-effects` requirement enables PPDDL extensions.

Problem file — objects, init, goal, reward:

```
(define (problem blocksworld-4)
  (:domain blocksworld)
  (:objects b1 b2 b3 b4 - block)
  (:init (handempty) (clear b1) (clear b4)
         (ontable b2) (ontable b3)
         (on b1 b2) (on b4 b3))
  (:goal (and (on b1 b2) (on b2 b3) (on b3 b4)))
  (:goal-reward 100)
  (:metric maximize (reward)))
```

Key additions over standard PDDL:

- `(:goal-reward 100)` — reward for reaching the goal
- `(:metric maximize (reward))` — optimization objective

PPDDL: Probabilistic Effects Syntax

The core PPDDL extension — probabilistic outcomes for actions:

‹› Syntax

```
(probabilistic p1 (effect1) p2 (effect2) ...)
```

Each p_i is a probability, and each effect is a standard PDDL effect (conjunctions of add/delete literals).

Key rules:

- Probabilities must satisfy $\sum_i p_i \leq 1$
- If $\sum_i p_i < 1$, the remaining probability mass means **no effect** (null outcome)
- Nested `probabilistic` blocks are **not** allowed — the distribution must be flat

⚠ Constraint

PPDDL: Probabilistic Effects — Example

A block pick-up action that may fail (fumble):

```
(:action pick-up
  :parameters (?b - block)
  :precondition (and (clear ?b) (ontable ?b) (handempty))
  :effect (probabilistic
    0.9 (and (holding ?b)
             (not (ontable ?b))
             (not (clear ?b))
             (not (handempty)))
    0.1 (and)))   ; fumble — nothing happens
```

This defines a simple stochastic action:

- With probability **0.9**: successfully pick up the block
- With probability **0.1**: fumble, no state change

PPDDL Idiom

PPDDL: Reward Fluents

PPDDL provides a built-in `(reward)` fluent:

```
; In action effects:
(decrease (reward) 1)   ; cost
(increase (reward) 10)  ; reward

; In problem file:
(:goal-reward 100)
(:metric maximize (reward))
```

Objective

$$\max_\pi \mathbb{E}\left[\sum_{t=0}^{T} r_t + R_{\text{goal}} \cdot \mathbf{1}[\text{goal reached}]\right]$$

Example — navigate with outcome-dependent costs:

```
(:action navigate
  :parameters (?from ?to - location)
  :precondition (and (robot-at ?from)
                     (path ?from ?to))
  :effect (probabilistic
    0.8  (and (robot-at ?to)
              (not (robot-at ?from))
              (decrease (reward) 1))
    0.15 (and (decrease (reward) 1))
    0.05 (and (robot-at ?to)
              (not (robot-at ?from))
              (damaged)
              (decrease (reward) 5))))
```

Success (cost 1) | Stuck (cost 1) | Damaged arrival (cost 5)

Example Domain: Triangle Tireworld

A classic IPPC benchmark illustrating dead-ends in probabilistic planning.

PPDDL Semantics: Grounding to an MDP

➜ **PPDDL → Factored MDP**

A PPDDL domain + problem defines a **factored MDP** $\mathcal{M} = (S, A, T, C, G, s_0)$:

- **States** S: consistent truth assignments to ground predicates (n Boolean predicates $\Rightarrow$ up to 2^n states)
- **Actions** $A(s)$: ground action instances whose preconditions hold in s
- **Transitions** $T(s'|s, a)$: derived from probabilistic effects of a
- **Cost/Reward** $C(s, a)$: from `(increase/decrease (reward) ...)` in effects
- **Goal** $G \subseteq S$: states satisfying the `(:goal ...)` condition; **Initial state** s_0: from `(:init ...)`

⚠ **State space explosion**

E.g. Blocksworld with 10 blocks has 131 ground predicates $\Rightarrow$ up to 2^{131} theoretical states (many inconsistent, but reachable space still very large).

Solver strategies:

- **Full enumeration**: build the explicit MDP (only feasible for small problems)
- **Incremental exploration**: generate states on demand (LAO*, LRTDP from Part 2)

STRIPS Relaxation Heuristics for SSPs

Computing admissible heuristics directly from the PPDDL representation

The min-min / h^det approach

Solve the **all-outcomes determinization** optimally: each probabilistic effect becomes a separate deterministic action. The optimal plan cost $h^{det}(s)$ is **admissible** but searches in the original state space — expensive.

{} STRIPS relaxation: reasoning about atoms

Instead of searching in the state space, forget delete effects and track the cost of achieving individual **atoms**:

$$g_s(\omega) \leftarrow \min_{a \in \mathcal{A},\ \exists i,\ \omega \in \mathrm{add}_i(a)} \{g_s(\omega),\ C(s,a) + g_s(\mathrm{prec}(a))\}$$

Aggregate over goal atoms: $h(s) = \bigoplus_{\omega \in \mathcal{G}} g_s(\omega)$

- $\bigoplus = \max \rightarrow h^+_{max}$: **admissible** for SSPs (Bonet & Geffner, 2001; Teichteil-Königsbuch, Vidal & Infantes, AAAI 2011)
- $\bigoplus = \sum \rightarrow h^+_{add}$: **not admissible** but more informative in practice
- Polynomial in the number of atoms — much cheaper than h^{det}

PPDDL Limitations

Limitation	Description
No continuous variables	All predicates are Boolean — no real-valued fluents
No concurrent actions	Only one action per time step
No exogenous events	Environment changes only through agent actions
No partial observability	Agent always observes the full state
Limited reward	Only the simple `(reward)` fluent mechanism

⚠ Impact

Competition domains were restricted to Boolean, single-agent, fully-observable problems. Many real-world scenarios (continuous battery levels, multi-agent coordination, weather, noisy sensors) could **not be encoded**. Solvers optimized for PPDDL did not always generalize.

✦ Motivation for RDDL

Determinization: Leveraging Classical Planners

Using PDDL planners to solve PPDDL problems

The Big Idea: Determinization

Why solve a hard problem when you can solve an easier one?

Core insight

Instead of solving the full MDP (expensive), simplify it to a **classical planning problem** (cheap), solve that, execute the plan, and re-solve if things go wrong.

The recipe:

1. Take your PPDDL problem
2. Remove the probability annotations — create a PDDL version
3. Solve efficiently with a classical planner (FF, Fast Downward, ...)
4. Execute the plan in the real (stochastic) environment
5. If an unexpected outcome occurs, **replan** from the new state

Key advantage

All-Outcomes Determinization — Concept

Replace each probabilistic action with multiple deterministic actions

Definition

Given a probabilistic action a with outcomes $o_1, \ldots, o_k$ occurring with probabilities $p_1, \ldots, p_k$, create k **deterministic actions** $a_{o_1}, \ldots, a_{o_k}$, one per outcome.

Example: Action `move-right` in a stochastic grid world

Outcome	Probability	Deterministic action
Move right (intended)	0.8	`move-right_success`
Slip up	0.1	`move-right_slip-up`
Slip down	0.1	`move-right_slip-down`

The classical planner can now **choose** which outcome to assume — it will naturally pick the most convenient outcomes, yielding an **optimistic plan**.

All-Outcomes Determinization — Cost Assignment

How to assign costs to deterministic actions?

Option 1: Original cost

$$C(s, a_{o_i}) = C(s, a)$$

All deterministic versions share the original cost. Simple but ignores outcome likelihoods.

Option 2: Probability-weighted cost

$$C(s, a_{o_i}) = C(s, a) - \log(p_i)$$

Penalizes unlikely outcomes — the planner prefers plans using probable outcomes.

✦ **Theorem**

The probability-weighted cost has a nice interpretation: minimizing total cost corresponds to maximizing the **probability of plan success** (when $C(s, a) = 0$), since $-\sum \log p_i = -\log \prod p_i$.

Most-Likely-Outcome Determinization

The simplest possible determinization

Definition

For each probabilistic action a, keep **only the single most probable outcome** $o^* = \arg\max_{o_i} p_i$ and discard all others.

Advantages: simpler, smaller PDDL problem, faster to solve

Limitations

Very fragile — completely ignores unlikely outcomes. Fails badly when unlikely outcomes are **catastrophic** (dead-ends). The most likely outcome may have low probability (e.g., 0.3).

FF-Replan

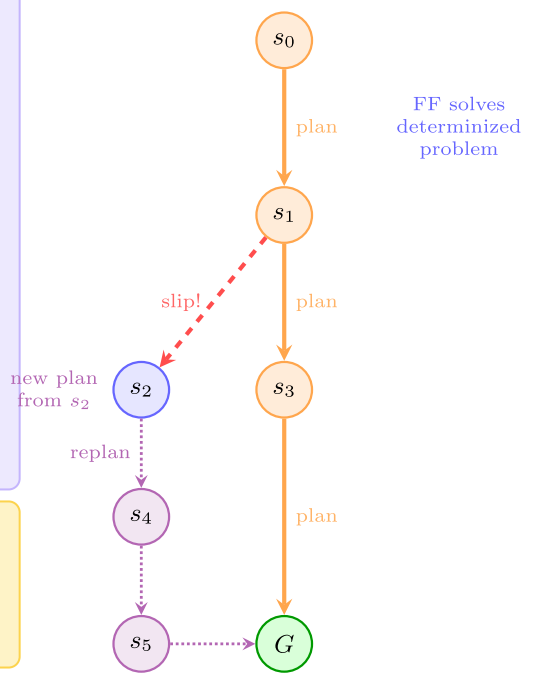
The surprisingly effective baseline

{} FF-Replan
(Yoon, Fern, Givan, 2007)

- **Initialize:** $s \leftarrow s_0$
- **Repeat** until $s \in G$:
  - Compute $\text{plan} \leftarrow \text{FF}(\text{determinize}(\mathcal{M}), s)$ using classical planner on determinized problem
  - **Execute plan:** for each action a in plan:
    - Execute a, observe $s' \sim T(\cdot \mid s, a)$
    - **If** $s' \neq$ expected outcome: discard plan, break to replan from s'
    - $s \leftarrow s'$

✦ Key property

Remarkably effective despite its simplicity! Won/placed highly at **IPPC 2004** and remained competitive in subsequent PPDDL-era competitions.



Yoon, Fern, Givan. FF-Replan: A Baseline for Probabilistic Planning. ICAPS 2007.

FF-Replan — Analysis

Why does such a simple approach work so well?

Why it works:

- Many PPDDL domains are "mostly deterministic" — the most likely outcome dominates
- Classical planners are very fast — replanning is cheap
- Replanning adapts to the actual state, providing implicit feedback

Weaknesses:

- No lookahead over uncertain outcomes — purely reactive
- May enter loops: plan, fail, replan same way, fail again
- Performs poorly with many equally likely outcomes or catastrophic dead-ends

Insight

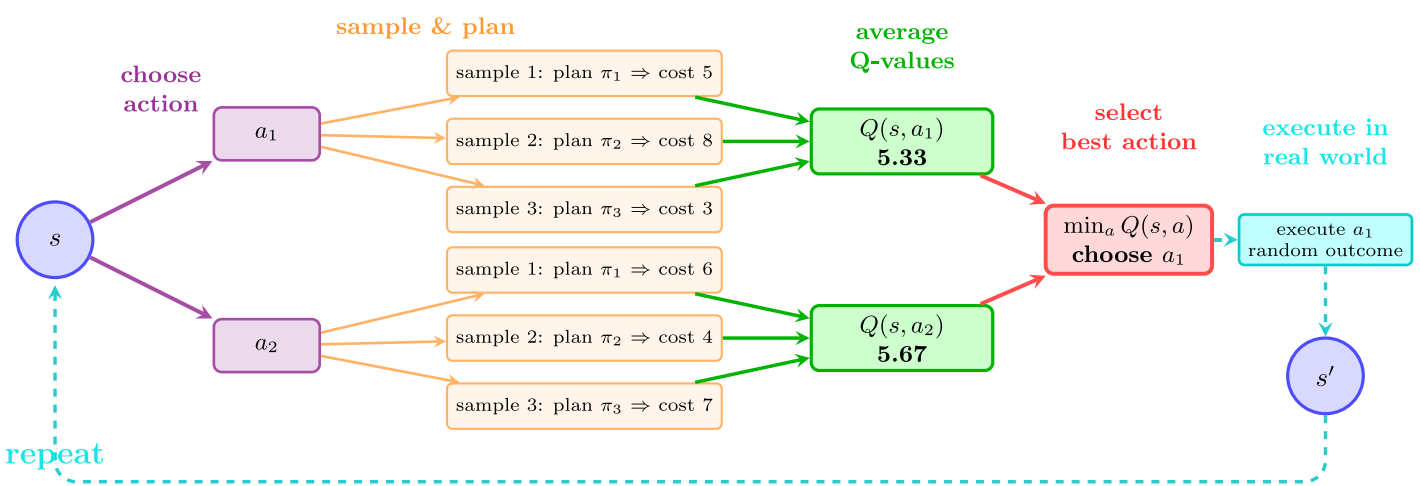
Hindsight Optimization — Concept

Looking into possible futures to make better decisions

Hindsight optimization

Evaluate an action by imagining multiple possible futures where all future randomness is revealed (**hindsight** = full information about future outcomes).

Intuition: Sample deterministic futures, compute optimal plans for each, and average the costs to evaluate actions.



Yoon, Fern, Givan, Kambhampati. Probabilistic Planning via Determinization in Hindsight. ICAPS 2008.

Hindsight Optimization — Algorithm

FF-Hindsight: combining FF with scenario sampling

{} Hindsight Optimization

For action selection at state s:

- **For each** candidate action $a \in A(s)$:
  - **Sample** M deterministic scenarios (complete future randomness)
  - **Solve** each determinized problem from (s, a) using FF $\rightarrow$ get costs $c_1, \ldots, c_M$
  - **Average:** $\hat{Q}(s, a) = \frac{1}{M} \sum_{m=1}^{M} c_m$
- **Select** best action: $a^* = \arg\min_a \hat{Q}(s, a)$

Properties:

- $\hat{Q}(s, a)$ provides an **approximation** of $Q^*(s, a)$
- Much better action selection than naive FF-Replan
- Cost: $|A| \times M$ classical planning calls per step

RFF — Robust FF

Building robust policies from classical plans

RFF (Robust FF)

Combines determinization with **robust policy construction**: instead of a single plan, builds a partial policy covering likely execution trajectories.

Key ideas:

- Generate a classical plan from the current state
- Identify states reachable under **all possible outcomes** along the plan
- Extend the policy to cover these states via additional planning calls
- Repeat until sufficient coverage

RFF vs FF-Replan

RFF constructs something between a single plan (FF-Replan) and a full policy (VI): a **partial policy** covering the most likely execution paths. It handles dead-ends more gracefully and reduces replanning frequency.

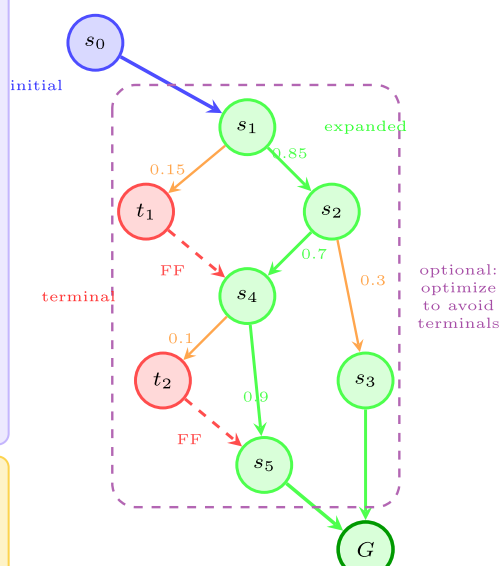
Teichteil-Koenigsbuch, Infantes, Kuter. RFF: A Robust, FF-Based MDP Planning Algorithm. AAAI 2008.

RFF — Robust FF

Incremental plan aggregation for robust policy construction

{} Robust FF (RFF)

- **Initialize:** $\pi \leftarrow \emptyset, T \leftarrow \{s_0\}$
- **Repeat** while $\Omega(s_0, \pi) > \rho$:
  - **For each** $s \in T$ with $\omega(s_0, \pi, s) > \rho$:
    - Solve determinized problem from s to $G_{\text{FF}} \subseteq G \cup S_\pi$ using FF
    - **Aggregate** plan into π, update terminal states T
  - **Estimate** $\Omega(s_0, \pi)$ via Monte-Carlo (N samples)
- **Return** π when $\Omega(s_0, \pi) \leq \rho$



✦ Key parameter

ρ = max acceptable probability of replanning. Controls the trade-off between offline planning time and online robustness. Winner of **IPPC 2008** fully-observable track.

Live Demo: SSP-Replan, SSP-Hindsight, SSP-PlanMerger

```
# algorithm: __solver__

# The solver is selected via the dropdown below
# and automatically passed to the backend by the code
print("▶ See maze visualization →")
```

❌ Backend not running. Start: cd backend && uv run python server.py

Reset | Stop | Solver: SSP-Replan (FF-Replan) | Speed: Fast Slow

Ready — click ▶ Run on the code

Controls:

- **Solver dropdown**: Choose between SSP-Replan (FF-Replan), SSP-Hindsight (FF-Hindsight), or SSP-PlanMerger (RFF)
- **Speed slider**: Adjust visualization speed
- Watch the **orange "R" markers** show where replanning occurs (SSP-Replan)

Live Demo: Determinization & LRTDP on Triangle Tireworld

Determinization: Summary

Strengths, limitations, and impact on the PPDDL era

Strengths

- Leverages fast, mature PDDL planners
- Scales to large state spaces
- Simple to implement
- Effective in "mostly deterministic" domains
- Can be enhanced (hindsight, RFF)

Limitations

- Suboptimal — no probabilistic reasoning
- Replanning overhead at execution time
- Struggles with frequent unlikely outcomes
- Dead-ends can cause failures

✦ **Historical impact**

The success of FF-Replan and its variants fundamentally shaped the PPDDL-era competitions, leading to harder benchmarks and motivating the transition to RDDL.

RDDL: Relational Dynamic Influence Diagram Language

A more expressive modeling framework

RDDL: A More Expressive Language

Why RDDL? PPDDL is limited to Boolean variables, single-agent, fully observable problems. Real-world domains need continuous variables, concurrent actions, exogenous events, and partial observability — motivating RDDL (Sanner, 2010).

A different philosophy: PPDDL is *action-centered* (effects per action); RDDL is *variable-centered* — each state variable has a **Conditional Probability Function (CPF)** defining its next-state distribution, rooted in dynamic Bayesian networks.

Key features:

- **Fluent types**: `state-fluent`, `action-fluent`, `observ-fluent`, `interm-fluent`, `non-fluent` — each can be `bool`, `int`, or `real`
- **CPFs**: each variable's evolution is specified as an expression mixing state, actions, and probability distributions (e.g., Poisson, Normal)
- **Concurrent actions**, **exogenous events**, **partial observability**, and **complex reward expressions** — all natively supported

More on RDDL

RDDL Key Concepts

Fluent types, CPFs, and reward structure

Fluent types — variables that define the domain:

Type	Role	Example
`non-fluent`	Constants	`ROAD(x,y)`, `MAX-FUEL`
`state-fluent`	Evolving state	`location(v)`, `fuel-level`
`action-fluent`	Agent decisions	`move(v,v')`, `refuel`
`interm-fluent`	Derived quantities	`total-demand`, `risk-score`
`observ-fluent`	Observations (POMDPs)	`noisy-position`

Conditional Probability Functions (CPFs) — each state variable's evolution:

RDDL vs PPDDL

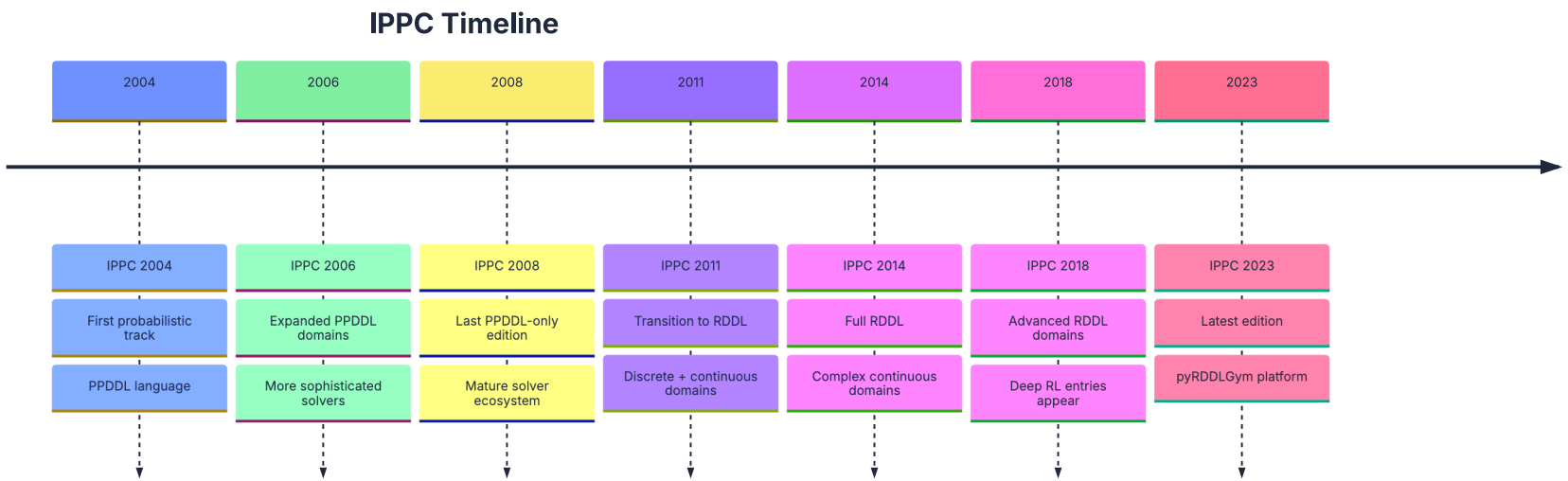
Feature	PPDDL	RDDL
Variable types	Boolean predicates	Bool, int, real fluents
State dynamics	Probabilistic effects per action	CPFs per variable
Concurrent actions	No	Yes
Exogenous events	No	Yes (via CPFs)
Continuous variables	No	Yes
Partial observability	No	Yes (`observ-fluent`)
Reward specification	`(reward)` fluent	Arbitrary expression
Used in IPPC	2004, 2006, 2008	2011, 2014, 2018, 2023

The International Probabilistic Planning Competition (IPPC)

Driving progress in probabilistic planning

IPPC: History & Timeline

The IPPC is part of the International Planning Competition (IPC) at ICAPS:



IPPC Eras: Two Phases of Development

PPDDL Era (2004–2008)

- **Boolean** state variables only
- **Single-agent**, fully observable
- Domains: Blocksworld, Tireworld, Zenotravel, Elevator, ...
- Solvers: adapted from **classical planning**
- Key insight: classical planning heuristics transfer well

RDDL Era (2011–2023)

- **Mixed** variable types (bool, int, real)
- **Concurrent** actions, exogenous events
- Domains: Traffic, Reservoir, Wildfire, Navigation, ...
- Solvers: **Monte Carlo** methods, gradient-based, learning
- Key insight: simulation-based approaches scale better

The transition from PPDDL to RDDL also marked a shift in **solver paradigms**: from classical-planning-based approaches to simulation-based and learning-based approaches.

PPDDL Era: Key Solvers (2004–2008)

FF-Replan (Yoon, Fern & Givan, 2007):

- **Determinize** the problem (pick most likely outcome), solve with FF (classical planner)
- If execution diverges from plan, **replan** from current state
- Remarkably simple — yet extremely competitive!

FF-Hindsight (Yoon, Fern, Givan & Kambhampati, 2008):

- Sample multiple deterministic "futures" (**hindsight optimization**)
- Evaluate actions by their average performance across futures
- Better action selection than single determinization

RFF (Teichteil-Koenigsbuch, Infantes & Kuter, 2008):

- **Robust** FF-based planner
- Generates policies (not just plans) using iterative replanning

PPDDL Era: More Key Solvers

GPT — General Problem Taker (Bonet & Geffner):

- Based on **LAO*** and **LRTDP** (the algorithms from Part 2!)
- Used admissible heuristics derived from classical planning relaxations
- Demonstrated that heuristic search scales to large MDPs

mGPT — mini-GPT (Bonet & Geffner):

- Improved GPT with **better heuristics** and more efficient implementation
- Combined multiple heuristic functions
- Strong performance across diverse IPPC domains

> **Key lesson from the PPDDL era**: classical planning techniques — especially heuristics like h^{add}, $h^{\max}$, and the FF heuristic — could be **effectively adapted** for probabilistic planning, providing strong

RDDL Era: Key Solvers (2011–2023)

PROST (Keller & Eyerich, 2012):

- **UCT-based** planner (Monte Carlo Tree Search)
- Dominated IPPC 2011 — the first RDDL competition
- Combined intelligent state-space exploration with simulation-based evaluation
- We will study UCT in detail in Part 5

Symbolic Perseus (Zamani et al.):

- Symbolic **POMDP** solver for partially observable RDDL domains
- Used algebraic decision diagrams (ADDs) for compact representation

JaxPlan / GurobiPlan (Gimelfarb et al., 2024):

- **JaxPlan**: compiles RDDL to JAX, uses **gradient-based** optimization
- **GurobiPlan**: formulates planning as **MILP** (Mixed-Integer Linear Program)

Lessons from the IPPC

- **Simple baselines can be surprisingly strong** – FF-Replan (determinize and replan) was competitive for years
- **Heuristic quality matters enormously** – the best PPDDL-era solvers had the best classical planning heuristics, yielding orders-of-magnitude speedups
- **Online planning works well** – UCT-based planners like PROST excelled; interleaving planning and execution is often more practical than computing full policies
- **Theory-practice gap persists** – theoretically optimal algorithms (exact VI) rarely scale; practical solvers sacrifice guarantees for scalability

Part 4 Summary

Modeling Languages

- **PPDDL**: extends PDDL with probabilistic effects and rewards
  - Defines factored MDPs via action schemas
  - Boolean variables, single-agent, fully observable
  - Used in IPPC 2004–2008
- **RDDL**: variable-centered modeling via CPFs
  - Bool/int/real fluents, concurrent actions
  - Exogenous events, partial observability
  - Used in IPPC 2011–2023

The IPPC

- **PPDDL era**: classical planning adaptations
  - FF-Replan, FF-Hindsight, RFF, GPT/mGPT
  - Heuristic search dominates
- **RDDL era**: Monte Carlo + learning methods
  - PROST (UCT), JaxPlan, Deep RL
  - Simulation-based approaches scale better
- **Key insight**: simple, well-implemented approaches often outperform complex theoretical solutions

Part 5: Simulator-Based Planning & Applications

From declarative models to black-box simulators

Simulator-Based Planning & scikit-decide

From declarative models to black-box simulators

Beyond Declarative Models

Real-world systems as simulators

Declarative models (PPDDL, RDDL):

- Describe dynamics symbolically with logical predicates and rules
- Structured, analyzable, optimizable
- Excellent for benchmarks and competitions

But many real-world domains are defined by code:

- Flight dynamics and atmospheric models
- Traffic simulators
- Industrial process controllers
- Supply chain simulations
- Robotic physics engines

The Simulator-Based Planning Paradigm

Planning with black-box transition functions

Simulator-based planning

The planner interacts with the domain through a black-box interface:

$$s' \sim \text{simulate}(s, a)$$

No explicit transition probabilities $T(s'|s, a)$ — only the ability to **sample** next states.

What the planner needs from the simulator:

- `get_initial_state()` $\rightarrow s_0$
- `get_applicable_actions(s)` $\rightarrow A(s)$
- `sample_next_state(s, a)` $\rightarrow s' \sim T(\cdot|s, a)$
- `get_cost(s, a, s')` $\rightarrow c$

scikit-decide — Overview

A unified framework for decision-making under uncertainty

> 📖 **scikit-decide**
> An open-source Python library developed by **Airbus AI Research** for decision-making under uncertainty.
> Key design principle: **separate domain definition from solver implementation**.

Architecture:

scikit-decide — Domain Definition

Defining an MDP as a Python class

```python
from skdecide import DeterministicPlanningDomain
from skdecide.hub.domain import MDP

class MyDomain(MDP):
    def _get_initial_state(self):
        return State(x=0, y=0)

    def _get_applicable_actions(self, state):
        return [Action.UP, Action.DOWN,
                Action.LEFT, Action.RIGHT]

    def _get_next_state_distribution(self, state, action):
        # Return distribution over next states
        ...

    def _get_transition_value(self, state, action, next_state):
        return Value(cost=1.0)

    def _is_terminal(self, state):
        return state == self.goal
```

Example

Flight Planning Fundamentals

How wind affects flight paths

Probabilistic Flight Planning — The Problem

A real-world application of simulator-based planning

Objective: Plan a flight path from origin to destination under uncertainty.

Sources of uncertainty:

- **Wind**: speed and direction vary stochastically, with spatial and temporal correlation
- **Weather cells**: storms and turbulence that appear, move, and dissipate
- **Fuel consumption**: depends on wind conditions (headwind vs tailwind)
- **Air traffic**: routing constraints that change dynamically

⚠ Why declarative models fall short

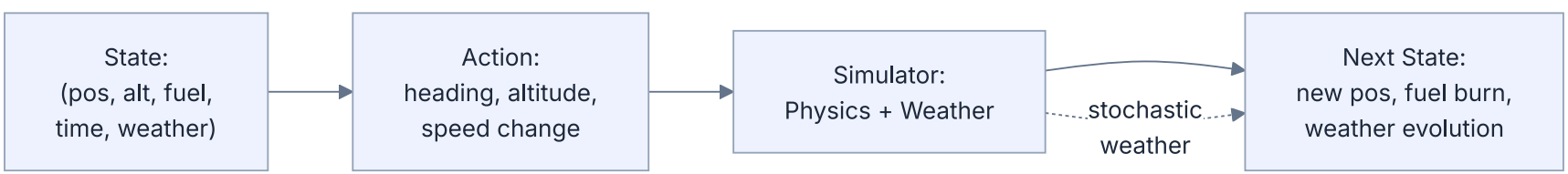
- Complex atmospheric physics cannot be captured in PPDDL or RDDL predicates
- Continuous state space: position (x, y), altitude z, fuel f, time t
- Spatially correlated stochastic weather evolution
- Need to interface with existing meteorological models and databases

Flight Planning — Domain Structure

Modeling the flight planning MDP

Flight Planning MDP

- **State**: (x, y, z, f, t, w) — position, altitude, fuel level, time, weather state
- **Actions**: heading changes $\Delta\theta$, altitude changes Δz, speed adjustments Δv
- **Transition**: physics simulation + stochastic weather evolution
- **Cost**: fuel consumption + time + safety penalty for weather proximity
- **Goal**: reach destination airport with sufficient fuel reserves



Cost function: $C(s, a, s') = \alpha \cdot \text{fuel}(s, a, s') + \beta \cdot \Delta t + \gamma \cdot \text{weather\_penalty}(s')$

Flight Planning — Solution Approach

Applying scikit-decide solvers

Step 1: Define the domain as a scikit-decide MDP

```
class FlightDomain(MDP):
    def _get_next_state_distribution(self, state, action):
        # Use atmospheric simulator to compute next state
        # Wind model + physics + stochastic weather update
        return weather_simulator.step(state, action)
```

Step 2: Choose and apply a solver

```
from skdecide.hub.solver.mcts import UCT

domain = FlightDomain(origin="LFBO", destination="LFPG")
solver = UCT(domain, num_iterations=1000, exploration_constant=0.5)
solver.solve()

# Execute online
state = domain.get_initial_state()
while not domain.is_terminal(state):
```

Live Demo: Flight Planning with LRTDP

Paris → New York with stochastic winds

The Broader Picture

Modern probabilistic planning is a spectrum

Course Wrap-Up

Key takeaways, further reading, and next steps

Key Takeaways

What we covered in 3.5 hours

1. Mathematical Foundations

- MDPs and SSPs provide the formal framework for planning under uncertainty
- The Bellman equation is the cornerstone: $V^*(s) = \min_a \left[C(s,a) + \sum_{s'} T(s'|s,a)\, V^*(s')\right]$

2. Rich Algorithm Landscape

- Dynamic Programming: VI, PI (exact, offline)
- Heuristic Search: LAO*, LRTDP (focused, offline)
- Determinization: FF-Replan, Hindsight, RFF (leveraging classical planning)
- Online & Monte Carlo: UCT, PROST, POMCP, Goal-HSVI (covered in Part 2)

3. Modeling Languages & Competitions

- PPDDL and RDDL standardize domain modeling for the community
- IPPC competitions drive algorithmic innovation

4. Extended Models & Real-World Applications

- Dead-ends, MAXPROB, FRET, and safety constraints for robust SSPs
- Extended SSP models: short-sighted SSPs, iDual, temporal logic specifications (Part 5)

Thank You!

Questions?

Florent Teichteil-Koenigsbuch

ICAPS 2026 Summer School — Probabilistic Planning

Course materials: github.com/fteicht/icaps26-prob-planning

Slides built with Slidev