Machine Learning Algorithms for Assisting Solvers for Decision Optimization Problems

Morteza Kimiaei*
Fakultät für Mathematik, Universität Wien
Oskar-Morgenstern-Platz 1, A-1090 Wien, Austria
Email: kimiaeim83@univie.ac.at
WWW: http://www.mat.univie.ac.at/~kimiaei

(*Corresponding Author)

Vyacheslav Kungurtsev
Department of Computer Science, Czech Technical University
Karlovo Namesti 13, 121 35 Prague 2, Czech Republic
Email: vyacheslav.kungurtsev@fel.cvut.cz

Abstract. Combinatorial decision problems lie at the intersection of Operations Research (OR) and Artificial Intelligence (AI), encompassing structured optimization tasks such as submodular selection, dynamic programming, planning, and scheduling. These problems exhibit exponential growth in decision complexity, driven by interdependent choices coupled through logical, temporal, and resource constraints. Classical optimization frameworks—including integer programming, submodular optimization, and dynamic programming—offer rigorous theoretical foundations but often struggle to scale in high-dimensional, uncertain, or dynamic environments.

This survey provides a unified overview of classical and learning-augmented approaches for combinatorial decision optimization. It reviews fundamental mathematical models and algorithmic paradigms, spanning submodular minimization and maximization, sequential and stochastic dynamic programming, and temporal and resource-constrained scheduling. The survey further examines how Machine Learning (ML) and Reinforcement Learning (RL) enhance these frameworks by learning heuristic policies, approximating value functions, and guiding branching or rollout decisions. Hybrid ML/RL–optimization methods are highlighted as a means to extend the reach of traditional solvers to adaptive, data-driven, and large-scale decision spaces.

By integrating classical combinatorial structures with learning-based inference, this survey underscores the emergence of ML-assisted decision solvers as a new paradigm for scalable planning and optimization. The resulting hybrid frameworks fuse the interpretability and rigor of optimization theory with the adaptability and generalization power of modern AI, offering a consolidated perspective on the convergence of OR and AI in decision optimization.

Keywords. Global optimization, combinatorial optimization, disjunctive programming, submodular optimization, probabilistic programming, backward induction, machine learning, reinforcement learning, neural networks

Contents

1 Introduction

Decision-making in complex systems often involves selecting optimal actions, sequences, or configurations from an exponentially large set of possibilities. Such problems arise across domains, including operations research (OR), logistics, energy systems, and artificial intelligence (AI), where decisions must satisfy logical, temporal, and resource constraints while optimizing a quantitative objective. In recent years, neural networks (NNs) and graph neural networks (GNNs) have emerged as powerful AI tools to model and solve such decision and optimization tasks, bridging learning and combinatorial reasoning within OR contexts. Despite decades of progress, the combinatorial nature of these problems continues to pose significant computational challenges, especially when uncertainty, interdependence, or sequential structure is present. This work seeks to provide a unified mathematical and algorithmic framework that connects classical combinatorial optimization with learning-based methods such as machine learning (ML) and reinforcement learning (RL). By integrating OR principles with AI and neural computation, we aim to capture both structural constraints and data-driven adaptability. The goal is to enable scalable, adaptive, and interpretable approaches to decision optimization that can reason over both symbolic constraints and data-driven models. In particular, GNN-based formulations offer a natural way to represent dependencies among decision variables through graph structures, enhancing interpretability and efficiency in large-scale decision-making. To this end, we begin by formalizing the general structure of combinatorial decision problems and analyzing their foundational components.

1.1 Combinatorial Decision Problems

Combinatorial decision problems constitute a broad class of discrete optimization tasks where the objective is to determine an optimal subset, ordering, or sequence of decisions subject to problem-specific feasibility constraints. Unlike constraint satisfaction problems (CSPs), which primarily focus on identifying feasible assignments, combinatorial decision problems emphasize the selection of decisions that maximize or minimize a quantitative objective function over a combinatorial space. These problems frequently appear in operations research, logistics, scheduling, and planning, and are characterized by their exponential growth in computational complexity as the decision dimension increases.

Formally, let $\mathcal{X}$ denote a finite or countably large decision space, where each $x \in \mathcal{X}$ represents a discrete configuration, subset, or sequence of decisions. A general combinatorial decision problem can be expressed as

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & x \in \mathcal{F}, \end{aligned} \tag{1}$$

where $f : \mathcal{F} \subseteq \mathcal{X} \to \mathbb{R}$ denotes a cost or reward function, and $\mathcal{F}$ represents the feasible decision set determined by logical, algebraic, or resource constraints. Let $[n] := \{1, \cdots, n\}$ for any positive integer value n. The feasible region $\mathcal{F}$ can often be defined implicitly by a collection of combinatorial relations such as

$$\mathcal{F} := \{x \in \mathcal{X} \mid g_i(x) \leq 0,\ h_j(x) = 0,\ \forall i \in [m], j \in [p]\}, \tag{2}$$

where g_i and h_j encode inequality and equality constraints that describe capacity limits, precedence relations, or resource couplings.

Many canonical problems can be represented as instances of the problem (1):
• **Submodular maximization:** selecting a subset $S \subseteq V$ that maximizes a set function $f(S)$ with the diminishing returns property.
• **dynamic programming and sequencing:** optimizing $f(x_1, x_2, \ldots, x_T)$ over temporal decision sequences subject to state transitions.
• **Planning and scheduling:** determining action sequences or job assignments under temporal and precedence constraints.

These formulations unify discrete optimization, decision-making, and temporal reasoning under a single framework. Classical algorithms—such as greedy procedures, branch-and-bound, or backward induction—can provide exact or approximate solutions, but they often struggle to scale beyond modest dimensions due to the combinatorial explosion of $\mathcal{F}$.

1.2 The Challenge of Sequential and Structured Decisions

A distinguishing feature of combinatorial decision problems is the presence of sequential or interdependent decision variables. In scheduling, planning, and control, each choice constrains future options, introducing temporal or logical dependencies that exponentially expand the effective search space. Let $\mathcal{S}_t$ denote the system state at time step t, and a_t the decision (or action) taken at that step. The decision process can then be described recursively as

$$\mathcal{S}_{t+1} = T(\mathcal{S}_t, a_t),$$

where T is a state-transition operator encoding temporal feasibility, logical consistency, and resource evolution. The cumulative cost over a decision horizon T can be expressed as

$$J(\pi) = \sum_{t=1}^{T} c(\mathcal{S}_t, a_t),$$

where $\pi = \{a_1, a_2, \ldots, a_T\}$ denotes the decision policy. The optimal policy π^* satisfies

$$\pi^* = \operatorname*{argmin}_{\pi \in \Pi} J(\pi),$$

with Π representing the set of admissible policies that satisfy all feasibility and resource constraints.

Such formulations illustrate why classical algorithms—such as dynamic programming and backward induction—often encounter the curse of dimensionality, wherein computational complexity grows exponentially with the number of decision variables or time steps. When uncertainty is introduced—such as stochastic demand, variable processing times, or dynamic resource availability—the complexity further increases, motivating the development of approximate and learning-based solvers capable of generalizing across related decision structures. These solvers leverage function approximation, experience-based policy learning, and structural priors to reduce computational cost while retaining feasible or near-optimal decision quality.

1.3 Role of ML and RL in Decision Optimization

ML and RL have emerged as powerful complements to classical optimization techniques for addressing large-scale structured decision problems. ML methods can learn mappings from problem parameters to approximate solutions or heuristic scores, effectively acting as surrogate models that reduce computational search effort. Examples include neural approximations of value functions, learned branching and variable selection in mixed-integer solvers, and data-driven priority rules in scheduling [40, 50].

RL extends these ideas to sequential optimization. Here, $s \in \mathcal{S}$ denotes the current state of the system, and $a \in \mathcal{A}$ is the action (or decision) taken in that state. The function $c(s, a)$ represents the immediate cost (or negative reward) incurred by applying a in s, while $T(s, a)$ denotes the (possibly stochastic) state-transition operator mapping the current state–action pair to the next state. The expectation $\mathbb{E}[\cdot]$ is taken over stochastic transitions, and $V^*(s)$ is the optimal value function giving the minimum expected discounted cumulative cost from state s onward. By interacting with a simulated or real-world environment, an RL agent learns a policy $\pi(a|s)$ that maximizes cumulative reward, approximating the solution to the Bellman optimality equation [7, 62]

$$V^*(s) = \min_a \ \mathbb{E}\big[c(s, a) + \gamma V^*(T(s, a))\big],$$

with the discount factor $\gamma \in [0, 1)$. This recursive structure directly parallels dynamic programming, but RL enables scalable approximation through sampling, neural function representation, and experience replay [15, 42]. Consequently, RL-based solvers can generalize decision-making patterns across families of problem instances rather than solving each case from scratch.

The integration of ML and RL with combinatorial decision optimization represents a hybrid paradigm: ML models capture statistical structure in decision

spaces, while classical optimization ensures logical consistency and feasibility. This synergy enables adaptive, data-efficient solvers capable of tackling previously intractable problems in logistics, energy scheduling, and multi-agent planning [38, 45, 50].

1.4 Research Questions

This survey is structured around three overarching research questions that guide the integration of ML and RL within classical frameworks for combinatorial decision-making. Together, they define how learning-based paradigms enhance, generalize, and unify traditional approaches in OR, planning, scheduling, and submodular optimization.

Q1. How can ML and RL augment classical decision optimization frameworks? This question explores how predictive and policy-based learning are incorporated into optimization pipelines—including temporal planning, scheduling, and submodular maximization—to improve scalability, adaptability, and generalization beyond deterministic solvers [35, 42, 51].

Q2. What algorithmic and representational mechanisms enable this integration? It investigates hybrid formulations that link symbolic logic, integer programming, and dynamic programming with neural and probabilistic models—such as GNN-based decision graphs, neuro-symbolic policies, and learned surrogate objectives—bridging discrete combinatorial structure with continuous learning dynamics [14, 40, 73].

Q3. What theoretical and practical challenges remain open? This includes open questions on optimality guarantees, convergence under uncertainty, interpretability of learned decision rules, and the trade-off between empirical performance and theoretical soundness in learning-enhanced optimization [30, 37, 58].

These questions collectively frame the survey's exploration of how ML and RL methods extend classical optimization into adaptive, data-driven, and temporally aware decision frameworks.

1.5 Survey Scope and Organization

This paper serves both as a **survey** and a **conceptual synthesis** bridging classical OR, planning, and scheduling with modern learning-based optimization. It integrates foundational models such as temporal and stochastic planning [26, 50, 52], backward induction [7, 56], and submodular optimization [35, 37, 58] with contemporary ML/RL approaches [4, 10, 20, 42].

- Section 1 introduces the motivation and outlines challenges in structured and sequential decision-making, setting the stage for ML/RL-enhanced frameworks.
- Section 2 reviews the theoretical background on ML and RL, highlighting their role in optimization and policy learning.
- Sections 3–4 cover classical and generalized planning and scheduling models, including temporal, stochastic, and event-triggered formulations, and introduce learning-based disjunctive frameworks.
- Section 6 details the backward induction framework and its ML/RL extensions, illustrating forward–backward reasoning, uncertainty handling, and imitation learning.
- Section 7 surveys broader learning-enhanced decision algorithms, which blend optimization with adaptive learning and heuristic discovery.
- Section 8 develops submodular optimization methods—both classical and ML/RL-augmented—showing how learning-driven surrogates and policies scale discrete optimization.
- Section 9 introduces dynamic and temporal decision optimization, combining GNN-based reasoning with RL for adaptive scheduling and resource allocation.
- Section 10 outlines current challenges and future directions, summarizing open problems across submodular, sequential, and temporal optimization. It highlights key research gaps in scalability, interpretability, and data efficiency, and proposes directions for unifying classical optimization with ML and RL frameworks.
- Section 11 concludes with open challenges, theoretical frontiers, and directions for integrating differentiable, neuro-symbolic, and transfer learning approaches in decision optimization.

Throughout, the survey maintains a unified perspective: classical logic- and graph-based optimization provides structural rigor, while ML and RL inject adaptability, scalability, and uncertainty-awareness. The resulting synthesis forms a conceptual bridge between symbolic optimization and data-driven intelligence for temporal and combinatorial decision-making.

1.6 Our Contributions

This survey contributes a unified and systematic perspective on **learning-enhanced combinatorial decision optimization**, integrating the classical

theory of OR with modern ML and RL methodologies. It is positioned at the intersection of Submodular Optimization (`SubModOpt`), Dynamic Programming (`DynP`), Disjunctive Programming (`DisjP`), Disjunctive Scheduling (`DisjS`), Probabilistic Programming (`ProbP`), and Backward Induction (`BI`), establishing conceptual and algorithmic bridges among these domains. Our main contributions are summarized as follows:

- **Unified conceptual framework:** We formalize combinatorial decision problems under a common mathematical structure encompassing submodular, disjunctive, and recursive (dynamic) optimization models. This framework connects classical paradigms such as the Lovász extension [37], Schrijver's polynomial minimization algorithm [58], and Balas'`DisjP` [5] with modern learning-based inference.
- **Integration of ML and RL into structured optimization:** We review how supervised ML and RL augment classical methods by approximating value functions, learning heuristic or policy-based decision rules, and guiding search or branching in high-dimensional decision spaces. Representative advances include GNN–RL frameworks for `DisjS` [38, 50, 73], backward-induction-enhanced RL [4, 20], and submodular RL [35, 51].
- **Algorithmic synthesis:** We systematically present a unified family of algorithmic templates that blend learning and optimization. These methods demonstrate how hybrid solvers can preserve logical structure (e.g., disjunctions or submodularity), while achieving data-driven adaptability [15, 40, 53].
- **Dynamic and temporal decision optimization:** We extend the discussion to dynamic and temporal domains, introducing the recursive `DisjP-DynP` framework that integrates `DisjP` for spatial and resource feasibility with `DynP` for temporal recursion, respectively. This hybrid formulation links `DisjS` graphs with Bellman recursion, enabling adaptive reasoning over time-dependent decision structures. The synthesis generalizes classical scheduling [50] and `DynP` [7] to adaptive, risk-sensitive, and online decision-making via distributional RL [42, 45].
- **Comparative and conceptual analysis:** We provide a comparative taxonomy of classical and learning-enhanced algorithms, highlighting theoretical guarantees (e.g., optimality or submodularity) versus empirical scalability. Benchmark analyses emphasize how hybrid ML/RL–optimization methods achieve near-optimality in real-world decision problems [19, 40].

In summary, this work establishes a coherent theoretical and algorithmic foundation for **ML/RL–augmented decision optimization**, showing how learning principles can extend classical combinatorial methods to dynamic, data-rich, and uncertain environments.

2 Background on ML and RL

ML and RL constitute the algorithmic core of modern data-driven optimization. ML provides statistical and computational methods for inferring predictive or generative models from data, typically by minimizing an expected loss or maximizing likelihood under uncertainty. RL, in contrast, extends learning to sequential decision-making problems, where an agent interacts with an environment to maximize long-term expected reward through exploration and feedback. Together, ML and RL form complementary paradigms: ML captures patterns and uncertainties from data, while RL translates them into adaptive policies and control strategies. This section presents the fundamental definitions, mathematical formulations, and representative algorithms underlying both ML and RL, which later serve as computational building blocks for hybrid frameworks in `ProbP` (Section 4.4), `BI` (Section 6), and `SubModOpt` (Section 8). The presentation progresses from supervised, unsupervised, and probabilistic learning models to sequential decision-making and policy optimization, concluding with recent unified approaches such as forward–backward RL and safe adaptive submodular policies.

Throughout this section, expectations $\mathbb{E}_{(\cdot)}[\cdot]$ are taken with respect to the indicated probability distributions. The notation $\mathrm{KL}(p\|q)$ denotes the Kullback–Leibler divergence between two probability distributions p and q, defined as

$$\mathrm{KL}(p\|q) = \int p(x) \log \frac{p(x)}{q(x)}\, dx,$$

where p typically represents a variational or approximate distribution and q the corresponding target or true distribution (e.g., a posterior).

2.1 ML

ML refers to methods that automatically improve their performance on a given task through experience, without being explicitly programmed [44]. An ML model is a parametric mapping

$$h_\theta : \mathcal{X} \to \mathcal{Y},$$

where $\mathcal{X}$ is the input space, $\mathcal{Y}$ is the output space, and $\theta \in \mathbb{R}^d$ are learnable parameters. The goal is to minimize an expected loss over the data distribution $\mathcal{D}$,

$$\min_\theta \ \mathbb{E}_{(x,y)\sim\mathcal{D}}[\ell(h_\theta(x), y)],$$

where ℓ is a task-specific loss function (e.g., mean squared error, cross-entropy).

Supervised Learning. In supervised learning, the model learns from a labeled dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$, minimizing the empirical loss

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \ell(h_\theta(x_i), y_i).$$

Optimization proceeds iteratively using gradient-based methods to adjust θ in the direction that reduces $L(\theta)$.

Unsupervised Learning. Unsupervised learning seeks to discover hidden structure in unlabeled data $\{x_i\}_{i=1}^N$. Common objectives include maximizing the likelihood $\sum_i \log p_\theta(x_i)$ or minimizing reconstruction loss in autoencoders.

NNs. NNs are layered nonlinear mappings of the form

$$h^{(l+1)} = \sigma(W^{(l)} h^{(l)} + b^{(l)}), \quad l = 0, \ldots, L-1,$$

where $W^{(l)}$ and $b^{(l)}$ are the weight and bias matrices at layer l, σ is an activation function (e.g., `ReLU`, `sigmoid`), and $h^{(0)} = x$. Parameters $\theta = \{W^{(l)}, b^{(l)}\}$ are optimized by stochastic gradient descent (`SGD`) or its adaptive variants such as `Adam` or `RMSProp`. More generally, neural architectures that embed structural priors—such as graph topologies, rule systems, or relational templates—are collectively referred to as template-based models. These appear again in the context of planning and logical reasoning in Section 3.4.

GNNs. For a graph $\mathcal{G} = (V, E)$, each node $v \in V$ has an embedding $h_v^{(k)} \in \mathbb{R}^d$. At message-passing layer k, node representations are updated via

$$m_v^{(k)} = \sum_{u \in \mathcal{N}(v)} \phi_{\text{msg}}(h_u^{(k)}, e_{uv}), \qquad h_v^{(k+1)} = \phi_{\text{upd}}\left(h_v^{(k)}, m_v^{(k)}\right), \tag{3}$$

where ϕ_{msg} and ϕ_{upd} are neural functions and e_{uv} encodes edge features (e.g., precedence or machine type). After K iterations, a global graph embedding is obtained through a permutation-invariant readout $h_{\mathcal{G}} = \phi_{\text{read}}\big(\{h_v^{(K)} \mid v \in V\}\big)$.

These message-passing operators (Eq. 3) later reappear in template-based relational reasoning models for planning (Section 3.4), where they are applied to logical or Datalog structures instead of simple graphs.

Attention Mechanisms. Transformer attention generalizes GNN aggregation by learning global dependencies:

$$\mathrm{attn}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V, \tag{4}$$

where Q, K, V are query, key, and value matrices. Here, the quantity d_k is the dimensionality of the key (and query) vectors used for scaling the dot-product attention. The contextual encoder output is

$$\mathrm{Enc}_\theta(X) = \mathrm{FFN}\big(\mathrm{attn}(Q, K, V)\big), \tag{5}$$

where FFN is a feed-forward network with residual connections. Attention enables reasoning over job precedence and machine interactions in disjunctive graphs.

Deep Submodular Functions (`DSFs`). A Deep Submodular Function [10] is a neural network architecture designed to learn submodular set functions $f : 2^{\mathcal{V}} \to \mathbb{R}$. A `DSF` composes simple submodular building blocks through layers of concave transformations over modular functions. For an input set $S \subseteq \mathcal{V}$ represented by its incidence vector $x \in \{0, 1\}^{|\mathcal{V}|}$, a typical `DSF` has the form

$$f_\theta(S) = \sum_{i=1}^{m} w_i \, \phi_i \left(\sum_{j \in S} a_{ij} \right),$$

where each activation $\phi_i : \mathbb{R} \to \mathbb{R}$ is a concave, nondecreasing function that ensures the submodularity of f_θ, $a_{ij} \geq 0$ and w_i are feature weights, and $\theta = (a, w)$ are trainable parameters. Such networks preserve submodularity and are trained via stochastic gradient descent to approximate a target submodular objective.

Learning Frameworks for Submodular Functions. Beyond `DSFs`, other frameworks [3,6] aim to learn submodular functions directly from labeled subsets $\{(S_i, y_i)\}$ where $y_i = f^*(S_i)$ are oracle or empirical utilities. These methods assume the true function f^* lies within a hypothesis class $\mathcal{F}$ of submodular functions (e.g., mixtures of coverage or facility-location functions). Learning then solves a structured risk minimization problem:

$$\min_{f \in \mathcal{F}} \frac{1}{N} \sum_{i=1}^{N} \ell(f(S_i), y_i) + \lambda \|f\|_{\mathcal{H}},$$

where ℓ is a convex loss (e.g., squared or logistic) and $\|f\|_{\mathcal{H}}$ regularizes the parameters. Here, $\mathcal{H}$ denotes the hypothesis or reproducing-kernel Hilbert space in which the regularization norm $\|f\|_{\mathcal{H}}$ is defined.

Probabilistic Modeling. Probabilistic ML explicitly models uncertainty via distributions $p_\theta(y|x)$ or $p_\theta(x)$. For latent-variable models, the posterior $p_\theta(z|x)$ is approximated by a parametric family $q_\phi(z|x)$ using variational inference, minimizing the Kullback–Leibler divergence:

$$\min_{\theta,\phi} \mathrm{KL}(q_\phi(z|x) \| p_\theta(z|x)).$$

`ProbP`. `ProbP` languages (PPLs) such as `Edward` [64], `Pyro` [55], and `Stan` [55] allow users to specify generative models and inference procedures using general-purpose code. A probabilistic program defines random variables $\Theta \sim p_\theta(\Theta)$ and observations $\mathcal{D} \sim p_\theta(\mathcal{D} \mid \Theta)$, and performs inference on the posterior $p(\Theta \mid \mathcal{D})$. These frameworks automate inference via stochastic gradient estimators, variational inference, or Markov chain Monte Carlo (MCMC), enabling flexible Bayesian learning. `Edward` is a `ProbP` library built on TensorFlow that integrates deep learning and Bayesian inference, supporting scalable variational and Monte Carlo methods. `Pyro` is a deep universal `ProbP` language developed on PyTorch, designed for flexible modeling and stochastic variational inference. `Stan` is a standalone `ProbP` system emphasizing full Bayesian inference via Hamiltonian Monte Carlo and other gradient-based sampling techniques.

Posterior Sampling. In Bayesian learning, model uncertainty is represented by the posterior distribution

$$p(\Theta \mid \mathcal{D}) = \frac{p(\mathcal{D} \mid \Theta) p(\Theta)}{p(\mathcal{D})}.$$

Posterior sampling (Thompson sampling) draws parameter realizations $\Theta_i \sim p(\Theta \mid \mathcal{D})$ to form predictive distributions or scenario sets. This allows data-driven refinement of uncertainty models for optimization under uncertainty and `ProbP`.

Compositional Generative Population Models (CGPMs). A CGPM [57] is a modular probabilistic model representing populations of random variables using a composition of local generative components. Each component defines conditional distributions $p_\psi(X_i \mid Z_i)$ with latent variables Z_i and parameters ψ, and compositions correspond to graph or program structures. CGPMs allow probabilistic programs to express joint models $p_\Psi(X_1, \ldots, X_n)$ that can be queried, conditioned, or marginalized efficiently, supporting structured uncertainty reasoning. The symbols ψ and Ψ collectively represent the parameters of the local and global generative components composing the CGPM.

Amortized Variational Inference. Amortized inference replaces repeated optimization of variational parameters for each observation with a neural network inference model $q_\phi(\Theta \mid \mathcal{D})$, parameterized by ϕ. Given an evidence lower bound (ELBO)

$$\mathcal{L}(\phi, \theta) = \mathbb{E}_{q_\phi(\Theta|\mathcal{D})}\big[\log p_\theta(\mathcal{D} \mid \Theta)\big] - \mathrm{KL}\big(q_\phi(\Theta \mid \mathcal{D}) \,\|\, p(\Theta)\big),$$

amortized inference jointly optimizes (ϕ, θ) to efficiently approximate the posterior $p(\Theta \mid \mathcal{D})$ across data points.

Black-Box Variational Inference (`BBVI`). `BBVI` [67] provides a generic gradient-based framework for optimizing variational objectives without requiring analytic gradients of the model likelihood. The gradient of the ELBO with respect to ϕ is estimated using the score-function estimator:

$$\nabla_\phi \mathcal{L}(\phi) \approx \mathbb{E}_{q_\phi(\Theta)}\big[\nabla_\phi \log q_\phi(\Theta)\big(\log p(\Theta, \mathcal{D}) - \log q_\phi(\Theta)\big)\big].$$

This black-box estimator enables stochastic optimization of inference networks in complex probabilistic programs. Here, Θ denotes the collection of latent or model parameters being inferred, and $\mathcal{D}$ represents the observed dataset. The distribution $q_\phi(\Theta)$ is the variational approximation to the true posterior $p(\Theta \mid \mathcal{D})$, parameterized by ϕ.

Regularization and Generalization. To improve generalization and prevent overfitting, regularization techniques are applied, such as ℓ_2 weight decay, dropout, and early stopping. Generalization refers to the model's ability to perform well on unseen data sampled from the same distribution as the training data.

The `ML` algorithm (=Algorithm 20, introduced in Appendix 12.1). This algorithm outlines the general process behind most machine learning methods that use stochastic gradient-based optimization. It begins by initializing a model and its parameters using available data. Next, the algorithm repeatedly improves the model by comparing its predictions with observed outcomes and adjusting its parameters to reduce the difference—this adjustment is guided by gradients that point toward better solutions. Finally, the process continues until the model's performance stabilizes or a stopping criterion is met, yielding a trained model that captures patterns in the data.

Deep Probabilistic Models. Combining neural networks and probabilistic inference leads to deep generative models such as Variational Autoencoders (VAEs) and Generative Adversarial Networks (GANs), which form the basis for modern `ProbP` systems discussed in Section 4.4.

Imitation Learning and Neural Rationalization. Imitation learning trains a policy $\pi_\theta(a \mid s)$ to mimic expert demonstrations $\mathcal{D}_E = \{(s_i, a_i)\}$ by minimizing a supervised loss

$$\min_\theta \ \mathbb{E}_{(s,a)\sim\mathcal{D}_E}[-\log \pi_\theta(a \mid s)].$$

In game-theoretic extensions, neural rationalization models agents that infer or imitate rational backward reasoning processes, approximating equilibrium behavior through differentiable policy recursion.

2.2 RL

RL formalizes sequential decision making as a Markov Decision Process (MDP) $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma)$, where $\mathcal{S}$ and $\mathcal{A}$ denote state and action spaces, $P(s'|s, a)$ the transition model, $R(s, a)$ the reward, and $\gamma \in [0, 1)$ a discount factor. A policy $\pi_\theta(a|s)$ aims to maximize the expected discounted return

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)\right].$$

The policy-gradient update is

$$\nabla_\theta J = \mathbb{E}_{\pi_\theta}\big[\nabla_\theta \log \pi_\theta(a|s)\, \hat{A}_t\big].$$

Here, $\hat{A}_t$ denotes an estimator of the advantage function, e.g.,

$$\hat{A}_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t).$$

Proximal Policy Optimization (PPO). To stabilize policy-gradient training, PPO employs the clipped surrogate objective

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t\Big[\min\big(\rho_t(\theta)\hat{A}_t, \text{clip}(\rho_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\big)\Big], \tag{6}$$

where $\rho_t(\theta) = \pi_\theta(a_t|s_t)/\pi_{\theta_{\text{old}}}(a_t|s_t)$ and $\epsilon > 0$ controls the clipping range (see Algorithm 22 in Appendix 12.1). Here, the variable θ_{old} stores the previous policy parameters used to compute the importance sampling ratio in each iteration.

Value Functions and Bellman Operator. The state-value function $V^\pi(s)$ and the action-value function $Q^\pi(s, a)$ quantify the expected discounted return

starting from state s or state–action pair (s, a):

$$V^\pi(s) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \mid s_0 = s\right],$$

$$Q^\pi(s, a) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \mid s_0 = s, a_0 = a\right].$$

They satisfy the Bellman expectation recursions

$$V^\pi(s) = \mathbb{E}_{a\sim\pi(\cdot|s),\, s'\sim P}\big[R(s, a) + \gamma V^\pi(s')\big],$$
$$Q^\pi(s, a) = R(s, a) + \gamma\, \mathbb{E}_{s'\sim P,\, a'\sim\pi(\cdot|s')}\big[Q^\pi(s', a')\big].$$

Equivalently, these relations can be written using the Bellman operator

$$(\mathcal{T}V^\pi)(s) = \mathbb{E}_{a\sim\pi, s'\sim P}\big[R(s, a) + \gamma V^\pi(s')\big], \tag{7}$$

which provides the foundation for `DynP` and value iteration. In these equations, s' and a' denote the next-state and next-action random variables, respectively. The transition kernel $P(s'|s, a)$ defines the probability distribution over next states s' given the current state s and action a, while the policy $\pi(a'|s')$ specifies the distribution over actions taken in the next state.

Neural `BI`. Neural `BI` approximates the recursive Bellman equations in `DynP` using neural networks. Given a finite-horizon problem with value functions $V_t(s)$, the recursion

$$V_t(s) = \max_a \big[R(s, a) + \gamma\, \mathbb{E}_{s'\sim P(\cdot|s,a)} V_{t+1}(s')\big]$$

is approximated by a neural network $V_\phi(s, t)$ with parameters ϕ trained via temporal-difference or regression loss. At each step, a target value $\hat{V}_t(s)$ is computed from the Bellman backup,

$$\hat{V}_t(s) = \max_a \big[R(s, a) + \gamma\, \mathbb{E}_{s'\sim P(\cdot|s,a)} V_\phi(s', t{+}1)\big],$$

and the network parameters are optimized to minimize

$$\mathcal{L}(\phi) = \mathbb{E}\big[(V_\phi(s, t) - \hat{V}_t(s))^2\big].$$

This enables scalable backward reasoning in continuous or high-dimensional state spaces. Here, s' denotes the next state sampled from the transition distribution $P(s' \mid s, a)$, which specifies the probability of reaching state s' after taking action a in the current state s. This neural approximation of `DynP` later serves as the foundation for template-enhanced planning algorithms (see Algorithm 1, below), which generalize such backward reasoning to logical and relational domains.

Contextual Policies and Bandits. In contextual bandits, the agent observes a feature vector (context) $x_t \in \mathbb{R}^d$ and chooses an action $a_t \in \mathcal{A}$ according to a stochastic policy $\pi_\theta(a_t \mid x_t)$. The policy parameters $\theta \in \mathbb{R}^p$ are updated to maximize the expected reward

$$J(\theta) = \mathbb{E}_{x_t, a_t \sim \pi_\theta}\big[r(x_t, a_t)\big].$$

Contextual policies generalize supervised learning by allowing adaptive, data-dependent decision making and are the foundation for learned greedy and adaptive submodular algorithms.

The `AC` algorithm (=Algorithm 21, introduced in Appendix 12.1). This algorithm merges two complementary ideas in reinforcement learning: learning how good actions are and deciding which actions to take. It does so by maintaining two parts—an actor, which represents the decision policy, and a critic, which evaluates how well the actor's recent decisions performed. The critic provides feedback to help the actor improve its choices, and both components are refined together over time as the system gains more experience. This coordination between acting and evaluating allows the algorithm to learn efficient decision strategies in complex, dynamic environments.

The `PPO` algorithm (=Algorithm 22, introduced in Appendix 12.1). This algorithm is designed to make reinforcement learning more stable and reliable when improving decision policies. It starts by collecting experience from interactions with the environment and then measures how the current policy performs compared to past versions. During learning, it carefully limits how much the policy is allowed to change at each update so that improvements are gradual and do not destabilize performance. By repeatedly balancing exploration and controlled adjustment, the algorithm steadily refines the policy until it reaches a stable and effective behavior.

Exploration vs. Exploitation. RL agents must balance exploring new actions and exploiting known rewarding ones. Exploration techniques include ϵ-greedy action sampling, entropy regularization, and optimism bonuses $B(s, a)$ in Upper Confidence Bound (UCB) methods, where

$$Q^+(s, a) = Q(s, a) + \alpha B(s, a),$$

and $\alpha > 0$ controls exploration intensity. Beyond standard policy-gradient methods, recent approaches integrate backward reasoning and value propagation with forward policy execution.

Risk-Sensitive and Distributional RL. Standard RL maximizes the expected return, but in stochastic or safety-critical environments, it is useful to account for variability in outcomes. Distributional RL represents the return

as a random variable $Z(s,a)$ whose distribution captures uncertainty in future rewards. A common risk-sensitive criterion is the Conditional Value-at-Risk (`CVaR`):

$$\texttt{CVaR}_\alpha(Z) = \mathbb{E}[Z \mid Z \leq F_Z^{-1}(\alpha)],$$

where F_Z is the cumulative distribution function of Z and $\alpha \in (0,1]$ is a confidence level controlling risk aversion. The corresponding Bellman operator

$$\mathcal{T}^\pi_{\texttt{CVaR}} Z(s,a) = R(s,a) + \gamma\, \texttt{CVaR}_\alpha\big(Z(s',a')\big)$$

propagates the return distribution under the policy π and discount factor $\gamma \in [0,1)$. Risk-sensitive RL yields robust policies that balance expected performance and downside risk.

Forward–Backward RL (`FBRL`). `FBRL` [20] combines `BI` (value recursion) with forward policy execution. Backward passes propagate optimal value estimates $V_t(s)$ or $Q_t(s,a)$ backward in time, while forward passes sample trajectories (s_t, a_t) using the current policy π_θ. Joint optimization ensures consistency between predicted future values and observed rollouts, improving sample efficiency.

Optimistic Bootstrapping and Backward Induction (`OB2I`). `OB2I` [4] augments `BI` with optimism under uncertainty by adding an exploration bonus $b_t(s,a)$ to the value target:

$$Q_t(s,a) = R(s,a) + \gamma\, \mathbb{E}_{s'}\big[\max_{a'} Q_{t+1}(s',a')\big] + b_t(s,a),$$

where $b_t(s,a) \propto \sqrt{\frac{\log(1/\delta)}{N_t(s,a)}}$ depends on the visitation count $N_t(s,a)$ and confidence δ. This yields provably efficient exploration in `DynP`. Here, s' denotes the next state sampled from the transition distribution $P(s' \mid s,a)$, and a' denotes the action selected in that next state, typically according to the optimal or current policy at time step $t{+}1$.

Model-Based RL and Meta-Learning. When the environment dynamics are unknown, a transition model can be learned as

$$s_{t+1} = f_\psi(s_t, a_t, \Theta_t),$$

where f_ψ is a differentiable function parameterized by $\psi \in \mathbb{R}^{d_\psi}$, and Θ_t denotes exogenous random factors such as disturbances or arrivals. Learning f_ψ enables predictive control and planning using simulated rollouts. Meta-learning further enhances adaptability by learning a prior distribution $q_\phi(\pi)$ over policy parameters π, with $\phi \in \mathbb{R}^{d_\phi}$, so that new tasks can be solved efficiently by fine-tuning this prior. Both model-based and meta-learning paradigms connect RL with adaptive optimization and dynamic system identification.

Safe RL. This method constrains policy exploration to avoid actions that violate safety or performance bounds. A common formalism uses probabilistic logic shields [70], which restrict the feasible action set in state $s \in \mathcal{S}$ to

$$\mathcal{A}_{\text{safe}}(s) = \{a \in \mathcal{A} \mid P(\varphi(s,a)) \geq 1 - \delta\},$$

where $\varphi(s,a)$ is a logical safety predicate, encoding a Boolean or probabilistic condition, expressing whether action a is considered safe in state s, and δ is the allowable risk. Safe RL ensures that the learned policy $\pi_\theta(a \mid s)$ respects these probabilistic constraints during learning and deployment.

Logical and Disjunctive Feasibility in RL. Many decision problems involve mutually exclusive or logical constraints among actions. Let $\mathcal{A}(s_t)$ denote the feasible action set in state s_t, defined by

$$\mathcal{A}(s_t) = \Big\{ a_t \;\Big|\; \bigvee_{l=1}^{L} \big(g_t^l(s_t,a_t) = 0,\ h_t^l(s_t,a_t) \leq 0\big) \Big\},$$

where g_t^l and h_t^l encode equality and inequality constraints for the lth disjunctive condition, and $\bigvee$ denotes logical OR. This formalism connects RL with `DisjP` frameworks, allowing policies to reason over alternative feasible actions such as machine sequencing or resource assignment.

Adaptive Submodular Policy Optimization. Adaptive submodularity [28] extends `SubModOpt` to sequential decision processes where the utility of each action depends on observed outcomes. An adaptive policy π sequentially selects elements $a_t \in \mathcal{A}$ based on past observations to maximize the expected marginal gain:

$$\pi^* = \operatorname*{argmax}_{\pi}\ \mathbb{E}\left[\sum_{t=1}^{T} f(A_t, \omega_t)\right].$$

In this formulation, T denotes the planning or decision horizon, i.e., the total number of time steps or adaptive selections performed by the policy π, ω_t denotes the random outcome observed after taking action a_t, and $f(A_t, \omega_t)$ is a stochastic submodular utility function measuring the realized reward given the current selected set A_t and observation ω_t. Policy-gradient methods can optimize such adaptive policies efficiently while preserving submodularity guarantees.

Connections to Optimization. RL can be viewed as stochastic optimization over expected rewards. Q-learning resembles `DynP`; policy-gradient methods resemble gradient ascent; and actor–critic architectures parallel primal–dual updates in constrained optimization frameworks.

This section establishes the foundational concepts, notation, and algorithmic frameworks of ML and RL, which serve as building blocks for the hybrid approaches explored in later sections on `SubModOpt`, `ProbP`, and `BI`.

3 Classical Planning

Classical planning represents one of the earliest and most rigorously defined models of intelligent decision-making. In contrast to numerical optimization, it operates in a symbolic domain where facts, actions, and goals are expressed through logic and set theory. The planner reasons about which sequences of actions transform the initial world state into one satisfying the desired goal. This section first presents the logical and set-theoretic foundations of classical planning, then illustrates the `STRIPS` representation with canonical examples, and finally shows how these logical constructs can be encoded algebraically to interface with optimization and scheduling frameworks.

Unlike numerical optimization, classical planning is formulated entirely in the language of sets and propositional logic. Its primitive notions are logical atoms, preconditions, and effects rather than continuous variables or objective functions. Integer- or mixed-integer encodings that appear later in this section serve only as faithful translations of these logical relations, not as relaxations or approximations of continuous programs.

3.1 Logical and Set-Theoretic Foundations of Planning

A classical planning problem is defined in the symbolic and logical domain rather than as a numerical optimization task. Following [26, 33, 55], we represent a planning problem as a tuple

$$\mathcal{P} = (P, A, s_0, G),$$

where $P = \{p_1, p_2, \ldots, p_n\}$ is a finite set of propositional variables (facts about the world); $A = \{a_1, a_2, \ldots, a_m\}$ is a finite set of actions, where each action $a \in A$ is defined as

$$a = (\mathrm{pre}(a), \mathrm{add}(a), \mathrm{del}(a)),$$

with $\mathrm{pre}(a) \subseteq P$ denoting the **preconditions** that must hold before a is applied, and $\mathrm{add}(a), \mathrm{del}(a) \subseteq P$ representing its **additive** and **delete** effects; $s_0 \subseteq P$ is the **initial state**, specifying the propositions that are true initially; $G \subseteq P$ is the **goal condition**, specifying the propositions that must hold in the final state.

Each state $s \subseteq P$ represents the set of true propositions and may equivalently be represented by a binary vector $s \in \{0, 1\}^n$. The complete state space is

therefore $S = 2^P$. The deterministic state–transition function

$$\gamma : S \times A \to S, \qquad \gamma(s, a) = (s \setminus \mathrm{del}(a)) \cup \mathrm{add}(a),$$

maps each state–action pair (s, a) to its successor state. A plan is a finite sequence of actions

$$\pi = \langle a_1, a_2, \ldots, a_k \rangle,$$

such that iterative application of γ from s_0 yields a final state s_k satisfying $G \subseteq s_k$. Formally, the semantics of a plan $\pi = \langle a_1, \ldots, a_k \rangle$ can be expressed as a sequence of logical transitions

$$s_{t+1} = \gamma(s_t, a_{t+1}) = (s_t \setminus \mathrm{del}(a_{t+1})) \cup \mathrm{add}(a_{t+1}), \quad t = 0, \ldots, k-1,$$

with the initial condition s_0 and the goal satisfaction condition

$$\models (\pi, \mathcal{P}) \iff G \subseteq s_k.$$

That is, a plan π is valid if and only if, under the transition function γ, the set of propositions in the final state s_k entails all goal propositions in G. Classical planning thus emphasizes logical goal satisfaction rather than numerical optimization; minimizing plan length is usually secondary.

Non-IP Example. To emphasize the symbolic nature of planning before any numeric encoding, consider a robot in a 2×2 grid world. It starts at position $(1, 1)$ and must reach $(2, 2)$. Position atoms represent the state space

$$S = \{(1, 1), (1, 2), (2, 1), (2, 2)\},$$

and the action set $A = \{\mathrm{Up}, \mathrm{Down}, \mathrm{Left}, \mathrm{Right}\}$ is defined by:
- Up: move from (i, j) to $(i - 1, j)$ if $i > 1$,
- Down: move from (i, j) to $(i + 1, j)$ if $i < 2$,
- Left: move from (i, j) to $(i, j - 1)$ if $j > 1$,
- Right: move from (i, j) to $(i, j + 1)$ if $j < 2$.

The initial state is $s_0 = (1, 1)$ and the goal is $G = (2, 2)$. A valid plan is $\pi = \langle \mathrm{Down}, \mathrm{Right} \rangle$, which moves the robot $(1, 1) \to (2, 1) \to (2, 2)$. This plan is verified through the logical transition semantics γ.

Logical Inference Semantics. A planning domain can equivalently be expressed as a system of logical rules governing state transitions. For each action $a \in A$, the following inference rules define when a is applicable and how it changes the world:

$$\frac{\mathrm{pre}(a) \subseteq s_t}{s_{t+1} = (s_t \setminus \mathrm{del}(a)) \cup \mathrm{add}(a)}.$$

That is, if all preconditions of a hold in the current state s_t, then the successor state s_{t+1} is obtained by deleting the propositions in $\mathrm{del}(a)$ and adding those in $\mathrm{add}(a)$. The overall planning problem can be formulated as a satisfiability condition:

$$\exists \pi = \langle a_1, \ldots, a_k \rangle \ \text{ s.t. } \ (\forall t,\ \mathrm{pre}(a_t) \subseteq s_t) \ \wedge \ (G \subseteq s_k).$$

This logical form expresses plan existence as a purely declarative property, independent of any numeric optimization.

Before introducing algebraic encodings, it is essential to emphasize that the planning semantics are purely logical. A state is a subset of propositions, and an action deterministically maps one such set to another according to its precondition–effect structure. Any algebraic representation, such as a binary integer program, merely encodes these logical dependencies in a form compatible with optimization solvers. It does not relax or modify the symbolic semantics of the original planning problem.

Binary Integer Programming Encoding. While classical planning is fundamentally logical and set-theoretic, it can be encoded as a binary integer program (BIP) for integration with optimization or scheduling systems [5,26]. This encoding serves only as an implementation mechanism—not as a relaxation or redefinition of the logical model. A compact BIP encoding is

$$\begin{aligned}
\min \quad & \sum_{a_i \in A} x_{a_i} \\
\text{s.t.} \quad & x_{a_i} \le y_j, && \forall a_i \in A,\ p_j \in \mathrm{pre}(a_i), \\
& y_j \ge \sum_{a_i \in A} f_{\mathrm{add}}(p_j, a_i)\, x_{a_i}, && \forall p_j \in P, \\
& y_j \ge g_j, && \forall p_j \in G, \\
& x_{a_i}, y_j \in \{0,1\}.
\end{aligned}$$

Here, $x_{a_i} = 1$ if action a_i is chosen and $y_j = 1$ if fact p_j is true in the final state. The parameters $f_{\mathrm{pre}}, f_{\mathrm{add}} : P \times A \to \{0,1\}$ and $g_j \in \{0,1\}$ encode logical relations between actions and propositions. This simplified encoding abstracts away temporal sequencing; a full model would include time-indexed variables $x_{a_i,t}$ indicating execution at time t.

3.2 Block World and STRIPS-Style Representations

In this subsection, we introduce the classical STRIPS-style block world problem [2,26], which illustrates how planning is defined through logical predicates and action schemas. This example demonstrates the set-theoretic semantics of

classical planning and, for completeness, shows how its logical dependencies can be represented in an algebraic encoding for integration with optimization-based solvers. Importantly, the logical model remains the primary definition of the planning task, and the algebraic encoding serves only as a structural translation rather than a relaxation or redefinition of its semantics.

There are two blocks: A and B. In the initial state, it is assumed that block A is on top of block B, the top of A is clear, and block B is clear. The goal is to move block A from B to the table, such that block A is no longer on block B and block B remains clear. In such a problem, propositions are defined as

$$P := \{f_{\text{on}}(A, B), f_{\text{clear}}(A), f_{\text{clear}}(B), f_{\text{onTable}}(A), f_{\text{onTable}}(B), f_{\text{holding}}(A)\},$$

where $f_{\text{on}}(A, B)$ indicates that block A is directly on top of block B; $f_{\text{clear}}(A)$ and $f_{\text{clear}}(B)$ indicate that the tops of blocks A and B are clear, respectively; $f_{\text{onTable}}(A)$ and $f_{\text{onTable}}(B)$ indicate that a block is directly on the table; $f_{\text{holding}}(A)$ means that block A is currently being held by the agent. Actions are defined as follows: $f_{\text{pickUp}}(A)$, removing block A from B with the two preconditions $f_{\text{on}}(A, B)$ and $f_{\text{clear}}(A)$, and the two effects $f_{\text{holding}}(A)$ and $f_{\text{clear}}(B)$; $f_{\text{putDown}}(A)$, placing block A on the table with the precondition $f_{\text{holding}}(A)$ and the two effects $f_{\text{onTable}}(A)$ and $f_{\text{clear}}(A)$; $f_{\text{stack}}(A, B)$, placing block A on top of block B with effects $f_{\text{on}}(A, B)$, $f_{\text{clear}}(A)$, and the two preconditions $f_{\text{clear}}(B)$ and $f_{\text{holding}}(A)$. The initial state is given by

$$s_0 = \{f_{\text{on}}(A, B), f_{\text{clear}}(A), f_{\text{clear}}(B)\},$$

and the goal state is

$$G = \{f_{\text{onTable}}(A)\}.$$

The objective (secondary to goal satisfaction) is to minimize the number of actions taken to reach the goal. The action set is

$$A = \{f_{\text{pickUp}}(A), f_{\text{putDown}}(A), f_{\text{stack}}(A, B)\}.$$

We now introduce an algebraic representation of the block world problem as an illustrative encoding:

$$\begin{aligned}
\min \quad & x_{f_{\text{pickUp}}(A)} + x_{f_{\text{putDown}}(A)} + x_{f_{\text{stack}}(A,B)} \\
\text{s.t.} \quad & \sum_{a_i \in A} x_{a_i} \cdot f_{\text{pre}}(f_{\text{on}}(A, B), a_i) \geq 1, \\
& \sum_{a_i \in A} x_{a_i} \cdot f_{\text{pre}}(f_{\text{clear}}(A), a_i) \geq 1, \\
& \sum_{a_i \in A} x_{a_i} \cdot f_{\text{add}}(f_{\text{clear}}(B), a_i) \geq 1, \\
& \sum_{a_i \in A} x_{a_i} \cdot f_{\text{add}}(f_{\text{onTable}}(A), a_i) \geq 1.
\end{aligned} \tag{8}$$

In this formulation, $x_{a_i} \in \{0,1\}$ denotes whether action a_i is selected in the plan ($x_{a_i} = 1$) or not ($x_{a_i} = 0$); $f_{\text{pre}}(p,a) : P \times A \to \{0,1\}$ is a binary parameter equal to 1 if proposition p is a precondition of action a, and 0 otherwise; $f_{\text{add}}(p,a) : P \times A \to \{0,1\}$ is a binary parameter equal to 1 if proposition p is an additive effect of action a, and 0 otherwise.

The first and second constraints in (8) are **precondition constraints**: $f_{\text{pre}}(f_{\text{on}}(A,B), a_i) = 1$ if a_i requires $f_{\text{on}}(A,B)$ to be true before execution, and $f_{\text{pre}}(f_{\text{clear}}(A), a_i) = 1$ if a_i requires $f_{\text{clear}}(A)$. The first constraint ensures that at least one chosen action requires $f_{\text{on}}(A,B)$, and the second ensures that at least one chosen action requires $f_{\text{clear}}(A)$.

The third and fourth constraints are **additive effect constraints**: $f_{\text{add}}(f_{\text{clear}}(B), a_i) = 1$ if a_i produces $f_{\text{clear}}(B)$, and $f_{\text{add}}(f_{\text{onTable}}(A), a_i) = 1$ if a_i produces $f_{\text{onTable}}(A)$. Thus, the third constraint ensures that $f_{\text{clear}}(B)$ becomes true, and the fourth ensures that the goal condition $f_{\text{onTable}}(A)$ is satisfied. Overall, this encoding maintains a one-to-one correspondence with the logical planning semantics, serving as a compact algebraic bridge between propositional planning and mixed-integer optimization. In this case, the optimal plan is

$$\pi = \langle f_{\text{pickUp}}(A), f_{\text{putDown}}(A) \rangle,$$

which achieves the goal condition $f_{\text{onTable}}(A)$. The plan is visualized in Figure 1.

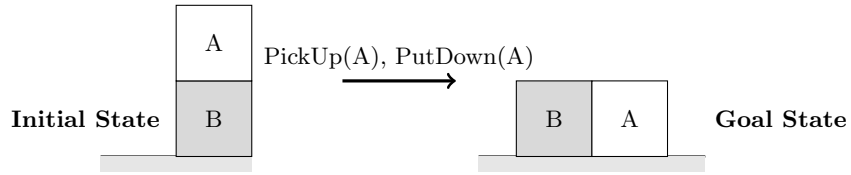


Figure 1: Block world problem: moving block A from on top of B to the table.

3.3 Logistics and Transportation Planning

In this subsection, we introduce a classical `STRIPS`-style logistics and transportation planning problem [26]. This domain models objects, locations, and movement actions using logical predicates and state transitions. It demonstrates how transportation planning can be expressed entirely in propositional logic and set-theoretic terms, and also how the same structure may be encoded algebraically for integration with mixed-integer optimization frameworks. As before, the logical semantics remain the foundation of the planning problem, while the algebraic form serves as a compact encoding, not a relaxation or redefinition.

We consider two locations, L_1 and L_2, and two movable objects, A and B. Initially, A is at L_1 and B is at L_2. The goal is to move both A and B to L_1. The set of propositions is

$$P := \{f_{\text{at}}(A, L_1), f_{\text{at}}(A, L_2), f_{\text{at}}(B, L_1), f_{\text{at}}(B, L_2)\},$$

where each predicate $f_{\text{at}}(X, L_i)$ indicates that object X is located at location L_i in the current state. The action set is defined as: $f_{\text{move}}(A, L_1, L_2)$, moving object A from L_1 to L_2 with the precondition $f_{\text{at}}(A, L_1)$ and the effect $f_{\text{at}}(A, L_2)$; $f_{\text{move}}(A, L_2, L_1)$, moving object A from L_2 to L_1 with the precondition $f_{\text{at}}(A, L_2)$ and the effect $f_{\text{at}}(A, L_1)$; $f_{\text{move}}(B, L_2, L_1)$, moving object B from L_2 to L_1 with the precondition $f_{\text{at}}(B, L_2)$ and the effect $f_{\text{at}}(B, L_1)$. The initial state and goal state are defined as

$$s_0 := \{f_{\text{at}}(A, L_1), f_{\text{at}}(B, L_2)\}, \qquad G := \{f_{\text{at}}(A, L_1), f_{\text{at}}(B, L_1)\}.$$

The action set is

$$A = \{f_{\text{move}}(A, L_1, L_2), f_{\text{move}}(A, L_2, L_1), f_{\text{move}}(B, L_2, L_1)\}.$$

The objective is to minimize the number of actions required to reach the goal state. Under the logical semantics, a valid plan is

$$\pi = \langle f_{\text{move}}(B, L_2, L_1)\rangle,$$

which moves object B to L_1 while A already satisfies its goal condition.

We now formulate an algebraic encoding of the same logical relationships:

$$\begin{aligned}
\min \quad & x_{f_{\text{move}}(A,L_1,L_2)} + x_{f_{\text{move}}(A,L_2,L_1)} + x_{f_{\text{move}}(B,L_2,L_1)} \\
\text{s.t.} \quad & \sum_{a_i \in A} x_{a_i} \cdot f_{\text{pre}}(f_{\text{at}}(A, L_1), a_i) \geq 1, \\
& \sum_{a_i \in A} x_{a_i} \cdot f_{\text{pre}}(f_{\text{at}}(B, L_2), a_i) \geq 1, \\
& \sum_{a_i \in A} x_{a_i} \cdot f_{\text{add}}(f_{\text{at}}(A, L_1), a_i) \geq 1, \\
& \sum_{a_i \in A} x_{a_i} \cdot f_{\text{add}}(f_{\text{at}}(B, L_1), a_i) \geq 1.
\end{aligned} \tag{9}$$

In this formulation, $x_{a_i} \in \{0, 1\}$ is a binary decision variable equal to 1 if action a_i is selected in the plan, and 0 otherwise; $f_{\text{pre}}(p, a) : P \times A \to \{0, 1\}$ is a binary parameter function equal to 1 if proposition p is a precondition of action a, and 0 otherwise; $f_{\text{add}}(p, a) : P \times A \to \{0, 1\}$ is a binary parameter function equal to 1 if proposition p is an additive effect of action a, and 0 otherwise.

The first two constraints in (9) represent **precondition constraints**: $f_{\text{pre}}(f_{\text{at}}(A, L_1), a_i) = 1$ if a_i requires A to be at L_1 before execution, and

$f_{\text{pre}}(f_{\text{at}}(B, L_2), a_i) = 1$ if a_i requires B to be at L_2. These ensure that valid preconditions must hold before executing corresponding actions.

The third and fourth constraints are **additive effect constraints**: $f_{\text{add}}(f_{\text{at}}(A, L_1), a_i) = 1$ if a_i produces the fact that A is at L_1, and $f_{\text{add}}(f_{\text{at}}(B, L_1), a_i) = 1$ if a_i produces the fact that B is at L_1. These ensure that the plan's outcome satisfies both goal conditions.

This algebraic encoding again corresponds directly to the logical semantics of the planning problem. It preserves all precondition–effect dependencies while abstracting away temporal sequencing. Hence, it serves as a bridge between symbolic reasoning and mixed-integer optimization without altering the underlying logic of the planning domain.

The optimal plan is $\pi = \langle f_{\text{move}}(B, L_2, L_1) \rangle$, since A is already located at L_1 in the initial state. The resulting transition from s_0 to G is illustrated in Figure 2.

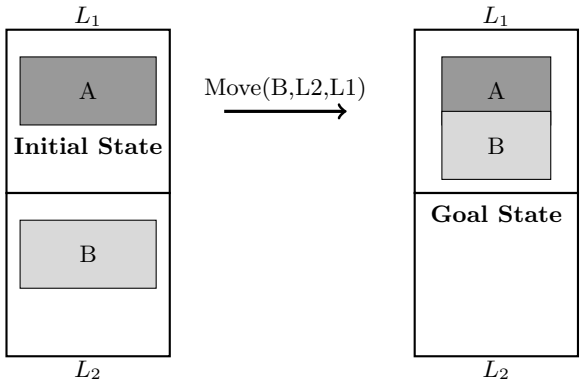


Figure 2: Logistics problem: moving object B from L_2 to L_1 while A is already at L_1.

3.4 ML/RL for Classical Planning

Modern research increasingly integrates ML/RL with classical planning by embedding symbolic reasoning within differentiable models. This hybridization allows neural agents to reason over logical structures such as `STRIPS` operators or lifted relational domains while maintaining the optimization and generalization capabilities of ML/RL. Two complementary paradigms dominate this line of work: (i) template-enhanced learning with background knowledge, and (ii) graph-based lifted heuristic learning. Both instantiate the general principle of Template-Enhanced Modeling (TEM), wherein the learning process is constrained by a symbolic or relational template defining the model's inductive bias.

Learning with Background-Knowledge Templates. Chen et al. [17] introduce a neuro-symbolic framework in which a classical planning domain is represented as a differentiable Datalog program containing a set of rules

$$\text{Condition} \;\rightarrow\; \text{Action},$$

encoding a background-knowledge (BK) policy $\sigma(s) \subseteq \mathcal{A}$ that returns admissible actions for state $s \in \mathcal{S}$. Each rule becomes a differentiable relational template through a Lifted Relational Neural Network (LRNN) parameterized by weights $\Theta = \{H_r, B^r_\lambda\}$. Given a Datalog program F, its differentiable evaluation on a state s proceeds by relational message passing:

$$\hat{h}_r = \phi_1\left(\sum_{\lambda \in \text{body}(r)} B^r_\lambda \, \hat{v}(\lambda)\right), \qquad h_\Theta(s) = \phi_2\left(\sum_{r \in F} H_r \, \text{agg}\big(\{\hat{h}_r\}\big)\right),$$

where ϕ_1, ϕ_2 are activation functions (e.g., `ReLU`), and agg is a permutation-invariant aggregation (e.g., sum or mean). The resulting $h_\Theta(s)$ represents either a policy score $\pi_\Theta(a|s)$ or a value estimate used for plan evaluation. Training optimizes Θ to minimize plan-quality loss or to maximize expected reward, yielding policies that improve plan efficiency while preserving symbolic interpretability.

Graph Encodings and Neural Heuristics. A complementary development by Horčik et al. [29] applies GNNs to lifted planning representations. A lifted planning state s is encoded as a typed graph $G(s) = (V, E)$, where vertices represent objects or ground atoms, and edges represent predicate relations. Three canonical state-encoding templates are considered:

- *Object encoding:* vertices are domain objects; edges connect objects that co-occur in predicates (Gaifman graph).
- *Atom encoding:* each ground atom is a vertex linked through shared arguments.
- *Object–atom encoding:* a bipartite graph combining the above two structures.

A GNN with parameters θ performs message passing over $G(s)$ to compute a heuristic $h_\theta(s)$ estimating the remaining plan cost. The parameters θ are trained from optimal plans using the ranking loss $L^*(\pi, \theta)$:

$$L^*(\pi, \theta) = \sum_i \sum_{t \in O_i \setminus \{s_i\}} \log\big(1 + \exp[g(s_i) - g(t) + h_\theta(s_i) - h_\theta(t)]\big),$$

where $g(s)$ is the accumulated cost-to-state, π is the optimal plan sequence, and O_i is the A* open list at iteration i. This loss penalizes any suboptimal

ranking of states during search, enabling the learned heuristic to mimic optimal A* expansions. Empirically, the object-encoding template achieved the best trade-off between expressiveness and computational cost, nearly matching the performance of the classical `LAMA` planner while remaining orders of magnitude faster.

Template-Enhanced Modeling (TEM). Both LRNN and GNN approaches instantiate Template-Enhanced Modeling, in which learning occurs within a symbolic or relational template rather than from unstructured data. In LRNNs, the template is a Datalog program defining the logical skeleton of the policy; in GNN heuristics, it is a graph schema defining relational message-passing structure. Training adjusts only the numeric parameters (Θ, θ), while the template encodes domain knowledge and structural priors. This separation yields: **Lifted generalization** across problem sizes due to variable-independent representation; **Sample efficiency** from reusing symbolic priors as inductive bias; **Interpretability** via human-readable templates; and **Compatibility** with RL formulations through policy-gradient or actor–critic updates.

Integration with RL and Search. From an RL standpoint, the learned policy $\pi_\Theta(a|s)$ or heuristic $h_\theta(s)$ can be incorporated into a reinforcement loop that alternates between value propagation and policy improvement. The LRNN parameters Θ are updated via policy gradients using trajectories (s_t, a_t, r_t, s_{t+1}) with advantage estimates $\hat{A}_t$, whereas the GNN parameters θ can be optimized through gradient descent on $L^*(\pi, \theta)$. Both yield differentiable surrogates of the Bellman operator under a fixed logical template, blending symbolic `DynP` with stochastic gradient optimization. The resulting unified procedure is summarized by the `TEMPL` algorithm, below, unifying symbolic `DynP` and neural learning. The background-knowledge template F defines the structural prior (analogous to the Bellman operator in `SPD`), while the neural parameters (Θ, θ) adapt to optimize plan quality or cumulative reward. This hybrid approach generalizes `DynP` to structured, differentiable reasoning, bridging the gap between classical planning and modern ML/RL. Algorithm 1 summarizes the overall learning–planning procedure. In ($S0_1$), the planner initializes the symbolic domain, logical or graph-based template F, and neural parameters (Θ, θ) that will be optimized. ($S1_1$) performs template-based message passing, propagating information through the relational structure of F to produce embeddings or value estimates for each state $s \in \mathcal{S}$. These representations drive ($S2_1$), where the algorithm updates the neural parameters either by policy-gradient reinforcement learning (for LRNN policies) or by minimizing the ranking loss $L^*(\pi, \theta)$ (for GNN heuristics). ($S3_1$) then executes or simulates plans under the current model, collecting new trajectories or rollouts for supervision and feedback. Finally, ($S4_1$) evaluates convergence based on expected improvement or loss stabilization, repeating the process until the learned policy or heuristic converges to a high-quality planner. Together, these stages implement a closed

learning–reasoning loop that integrates symbolic structure with gradient-based optimization. When search is used instead of direct policy execution in ($S3_1$), `TEMPL` typically adopts an A* planner guided by the learned heuristic $h_\theta(s)$. A* expands states in order of $f(s) = g(s) + h_\theta(s)$, combining actual and predicted costs to efficiently find a plan that minimizes expected path length or reward loss. The resulting plan $\pi = (a_1, \ldots, a_T)$ is then used to update the parameters in ($S2_1$).

4 Planning Generalizations

Classical planning, as introduced in the previous section, assumes instantaneous actions, deterministic outcomes, and purely logical representations of state transitions. However, many real-world decision problems involve temporal durations, resource sharing, uncertainty, and event-driven dynamics that go beyond static logical models. Planning generalizations extend the classical framework to capture these richer aspects by incorporating time, probability, and conditional logic while preserving the underlying symbolic semantics of preconditions, effects, and goals. In this extended setting, algebraic or mixed-integer formulations are used not as relaxations of logical disjunctions, but as operational encodings that preserve the logical structure of the problem. The following subsections present four major extensions—temporal and metric planning, stochastic and robust planning, event-triggered and conditional planning, and `ProbP`—each progressively integrating elements of optimization, uncertainty, and learning within the logical foundations of planning.

4.1 Temporal and Metric Planning

In the context of OR, the goal of the temporal planning problem [26] is to find optimal or feasible sequences of actions under time constraints and resource limitations. This can be achieved by integrating scheduling, optimization, and decision-making into one framework to handle real-world problems (e.g., logistics, workforce management, and project scheduling).

A temporal planning problem [26] can formally be considered as a tuple

$$\langle \mathcal{S}, A, I, G, T \rangle,$$

where $\mathcal{S}$ is the set of states, A is the set of actions with durations, I is the initial state, G is the goal state conditions, and T represents the temporal constraints. Before the mathematical formulation of such a problem, we discuss its key ingredients:

- **Action Timing.** Specific start times and durations are allocated to actions.
- **Temporal Constraints.** Actions are subject to temporal precedence (e.g.,

Algorithm 1 Template–Enhanced ML/RL Planning (`TEMPL`) Algorithm

Initialization (S0$_1$) Specify planning domain $\mathcal{D} = (\mathcal{S}, \mathcal{A}, P, R, \gamma)$ and background-knowledge template F (Datalog program or graph schema). Initialize neural parameters Θ (for LRNN) or θ (for GNN heuristic), learning rates $\eta, \eta_v > 0$, and convergence threshold $\epsilon > 0$.

repeat

Template-based message passing (S1$_1$) For each state $s \in \mathcal{S}$, compute relational embeddings using the logical/graphical template:

$$\hat{h}_r = \phi_1\left(\sum_{\lambda \in \text{body}(r)} B_\lambda^r \, \hat{v}(\lambda)\right), \quad h_\Theta(s) = \phi_2\left(\sum_{r \in F} H_r \, \text{agg}\big(\{\hat{h}_r\}\big)\right).$$

Here, ϕ_1, ϕ_2 are activation functions and agg denotes permutation-invariant aggregation (e.g., `sum` or `mean`).

Policy or heuristic update (S2$_1$)

- **Policy learning:** update Θ using policy-gradient objective

  $$\Theta \leftarrow \Theta + \eta \, \nabla_\Theta \log \pi_\Theta(a_t|s_t) \, \hat{A}_t,$$

  where $\hat{A}_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)$.

- **Heuristic learning:** update θ by minimizing the ranking loss $L^*(\pi, \theta)$ defined above using gradient descent:

  $$\theta \leftarrow \theta - \eta_v \nabla_\theta L^*(\pi, \theta).$$

Plan or trajectory generation (S3$_1$) Execute the current policy π_Θ or run search (e.g., A*) guided by $h_\theta(s)$ to obtain trajectories $\tau = \{(s_t, a_t, r_t, s_{t+1})\}_{t=1}^T$ and corresponding plans $\pi = (a_1, \ldots, a_T)$.

Convergence or termination (S4$_1$) If the expected improvement $\Delta J < \epsilon$ or validation loss stabilizes, stop; otherwise set parameters $\Theta \leftarrow \Theta_{\text{new}}$ or $\theta \leftarrow \theta_{\text{new}}$ and continue.

until convergence or maximum iterations.

Output Return optimized parameters (Θ^*, θ^*) and the corresponding policy π_{Θ^*} or heuristic $h_{\theta^*}(s)$ capable of generating high-quality plans under the relational template F.

action A must finish before action B starts) and deadlines or time windows (e.g., action A must occur between t_{start} and t_{end}).
• **Resource Constraints.** Resources (e.g., machines, workers) are limited and shared among tasks. Resources must never be allocated more than what can be used.
• **Types of objective functions.** Minimizing total completion time, minimizing lateness or tardiness, and maximizing resource utilization or profit are the three types of objective functions in temporal planning problems.

We formulate a typical temporal planning problem in OR as the following MILP problem

$$\begin{array}{lll} \min & C_{\max} & \\ \text{s.t.} & x_j \geq C_i, & \text{for all } (i,j) \in \texttt{PR}, \\ & C_i = x_i + d_i, & \text{for all } i, \\ & t_{\text{start}} \leq x_i \leq t_{\text{end}}, & \text{for all } i, \\ & \sum_{i \in A} r_i \, f_{\text{usage}}(t) \leq R, & \text{for all } t, \\ & C_{\max} \geq C_i, & \text{for all } i. \end{array} \tag{10}$$

Here, x_i denotes the start time of action i, and $C_i = x_i + d_i$ its completion time, where d_i is the given duration. The precedence relation $(i,j) \in \texttt{PR}$ indicates that action i must finish before action j starts. The parameters t_{start} and t_{end} define the earliest and latest allowable start times, respectively. Each action i requires r_i units of a shared resource with total capacity R. The function $f_{\text{usage}}(t)$ is an indicator that equals 1 if task i is active at time t (i.e., $x_i \leq t < C_i$), and 0 otherwise.

The first constraint enforces precedence relationships, ensuring that a successor action j cannot start before its predecessor i completes. The equality $C_i = x_i + d_i$ defines the completion time of each action. The third constraint enforces temporal window feasibility, requiring that each action start between its earliest and latest permissible start times t_{start} and t_{end}. The fourth constraint represents resource limitations, ensuring that the total resource usage $\sum_i r_i f_{\text{usage}}(t)$ does not exceed the available capacity R at any time t. The final constraint defines the makespan $C_{\max}$ as the latest completion time among all actions. Illustrative scheduling and energy-planning examples are relocated to Appendix 12.2 for reference.

4.2 Stochastic and Robust Planning

In many real-world domains, planning must account for uncertainties in action outcomes, resource availability, or environmental conditions. **Stochastic planning** generalizes classical and temporal planning by allowing probabilistic state transitions and random rewards. The objective is to compute a policy that

minimizes the expected cost (or equivalently, maximizes expected reward) over a stochastic process such as an MDP [31, 52].

Formally, a stochastic planning problem is defined by the tuple

$$\mathcal{P}_{\text{stoch}} = \langle \mathcal{S}, A, P, R, \gamma, \rho_0 \rangle,$$

where $\mathcal{S}$ is the finite set of states, A is the finite set of actions, $P(s'|s,a)$ denotes the transition probability from state s to s' when action a is executed, $R(s,a)$ is the immediate reward function, $\gamma \in [0,1)$ is the discount factor, and ρ_0 is the initial state distribution. A stationary policy $\pi : \mathcal{S} \to A$ maps each state to an action and induces a stochastic process $\{S_t\}_{t=0}^{\infty}$. The value of a policy π is the expected discounted reward

$$V^{\pi}(\rho_0) = \mathbb{E}_{\pi, P, \rho_0}\left[\sum_{t=0}^{\infty} \gamma^t R(S_t, \pi(S_t))\right].$$

The optimal policy $\pi^{\star}$ satisfies

$$V^{\star}(s) = \max_{a \in A}\left[R(s,a) + \gamma \sum_{s'} P(s'|s,a) V^{\star}(s')\right].$$

Illustrative stochastic grid-world planning example is relocated to Appendix 12.2 for reference.

Stochastic Optimization Formulation. In operations research, **stochastic planning** problems are often cast as **multistage stochastic programs**, where decisions evolve sequentially over time and adapt to uncertainty as it is revealed. A general formulation is

$$\min_{x_0 \in X_0} \mathbb{E}_{\xi_1} \min_{x_1(\xi_1) \in X_1(\xi_1)} \cdots \mathbb{E}_{\xi_T} \min_{x_T(\xi_{1:T}) \in X_T(\xi_{1:T})} \left[\sum_{t=0}^{T} f_t(x_t, \xi_t)\right], \tag{11}$$

where $T \in \mathbb{N}$ denotes the finite **planning horizon**, i.e., the number of decision stages in the stochastic program. At each stage $t = 0, 1, \ldots, T$, uncertainty ξ_t is observed and a decision x_t is chosen adaptively. $x_t \in \mathbb{R}^{n_t}$ denotes the decision vector at stage t. Each decision x_t is chosen after observing the random outcome ξ_t, and can depend on all previously observed realizations $\xi_{1:t} = (\xi_1, \ldots, \xi_t)$; $\xi_t \in \Xi_t$ is a random vector representing the uncertainty (e.g., demand, price, weather) revealed at stage t. The random variables $\xi_1, \ldots, \xi_T$ follow a joint probability distribution $\mathcal{D}$ on the sample space $\Xi_1 \times \cdots \times \Xi_T$; $f_t(x_t, \xi_t)$ is the cost (or negative reward) incurred at stage t when decision x_t is made and random event ξ_t is realized. Typical examples include production costs, delay penalties, or resource usage; $X_t(\xi_{1:t})$ denotes the feasible decision set at stage

t, which may depend on all previously realized uncertainties. This set encodes resource, logical, or coupling constraints such as capacity limits or inventory balance equations. The outer expectation $\mathbb{E}_{\xi_t}$ is taken with respect to the distribution of ξ_t, capturing the expected cost over random outcomes. The nested minimizations represent adaptive decisions: each x_t is chosen after observing $\xi_{1:t}$, so that decisions can respond to uncertainty as it unfolds.

The structure of problem (11) captures the essence of sequential decision-making under uncertainty. If all random variables ξ_t are known in advance, (11) reduces to a deterministic multistage planning problem. If uncertainty is revealed only once at $t = 0$, the formulation reduces to a two-stage stochastic program:

$$\min_{x_0 \in X_0} \Big\{ f_0(x_0) + \mathbb{E}_{\xi_1}\big[Q(x_0, \xi_1)\big] \Big\}, \quad Q(x_0, \xi_1) := \min_{x_1 \in X_1(x_0, \xi_1)} f_1(x_1, \xi_1),$$

where x_0 represents first-stage (here-and-now) decisions, and x_1 denotes recourse (wait-and-see) decisions made after observing ξ_1.

In energy systems, for example, x_t may represent generation levels or storage dispatch at time t, while ξ_t denotes stochastic wind output or demand. The objective is to minimize the expected total cost of operation while maintaining feasibility across all possible realizations of uncertainty.

Stochastic planning is applied in **inventory control**, **energy systems**, and **robot navigation**. In renewable scheduling, stochastic forecasts of wind or solar generation are incorporated to minimize expected operating cost while ensuring reliability.

The `SPD` Algorithm (=Algorithm 2). The `SPD` algorithm (Stochastic Planning via `DynP`) is a classical framework for solving MDP-based stochastic planning problems by iterative value updates. This algorithm iteratively improves the value function and policy through successive applications of the Bellman operator. It starts by initializing an arbitrary value function $V_0(s)$ for all states $s \in \mathcal{S}$ in ($S0_2$). In each iteration, it performs a **Bellman update** in ($S1_2$), computing the expected return of each action by combining immediate reward $R(s, a)$ and the discounted future value $\gamma \sum_{s'} P(s'|s, a)V_k(s')$. Next, in the **policy improvement** step ($S2_2$), it selects for each state the action a that maximizes this expected return, thereby generating an improved policy π_{k+1}. The algorithm then checks convergence in ($S3_2$) by measuring the maximum change between successive value functions; if this difference is below a tolerance ϵ, the algorithm terminates. Finally, the optimal policy $\pi^\star$ and value function $V^\star$ are returned. Thus, `SPD` alternates between evaluation and improvement, converging to the optimal solution of the stochastic planning problem under the Bellman fixed-point condition.

Algorithm 2 The Stochastic Planning DynP (SPD) Algorithm

Initialization ($S0_2$) Initialize $V_0(s)$ arbitrarily for all $s \in \mathcal{S}$. Set iteration counter $k = 0$.

repeat

Bellman update ($S1_2$) For each state $s \in \mathcal{S}$, compute

$$V_{k+1}(s) = \max_{a \in A} \big[R(s,a) + \gamma \sum_{s'} P(s'|s,a) V_k(s')\big].$$

Policy improvement ($S2_2$) Derive the improved policy

$$\pi_{k+1}(s) = \operatorname*{argmax}_{a \in A} \big[R(s,a) + \gamma \sum_{s'} P(s'|s,a) V_{k+1}(s')\big].$$

Convergence check ($S3_2$) If $\max_s |V_{k+1}(s) - V_k(s)| < \epsilon$, stop; else set $k \leftarrow k+1$.

until convergence or maximum iterations.

Output Return $\pi^\star(s) = \pi_k(s)$ as the optimal policy and $V^\star(s) = V_k(s)$ as the value function.

4.3 Event-Triggered and Conditional Planning

Event-triggered and conditional planning extends deterministic and temporal planning by explicitly modeling contingencies—decisions that depend on external events, observations, or conditions revealed during execution. Unlike fixed action sequences, an **event-triggered plan** encodes conditional branches such as if event E occurs, execute action A; otherwise execute action B.

Formally, a conditional planning problem is defined as

$$\mathcal{P}_{\text{cond}} = \langle \mathcal{S}, A, E, P, \Phi, G \rangle,$$

where $\mathcal{S}$ is the set of states, A the controllable actions, E the set of exogenous events, P defines probabilistic or logical event models, Φ is a conditional policy mapping event histories $e_{0:t}$ to actions, and G represents goal conditions. A conditional policy Φ aims to minimize

$$J(\Phi) = \mathbb{E}\left[\sum_{t=0}^{T} c(S_t, \Phi(e_{0:t}), E_t)\right], \quad S_{t+1} = f(S_t, \Phi(e_{0:t}), E_t).$$

Illustrative conditional planning example for autonomous flight is relocated to Appendix 12.2 for reference.

Mathematical Formulation. Event-triggered control can be expressed as the following stochastic optimization problem:

$$\begin{aligned} \min \quad & \mathbb{E}\left[\sum_{t=0}^{T} c_t(x_t, u_t)\right] \\ \text{s.t.} \quad & x_{t+1} = f(x_t, u_t, \omega_t), \quad t = 0, \dots, T-1, \\ & u_t = \Phi(x_t, \omega_{0:t}), \\ & \Phi \in \mathcal{F}_{\text{logic}}. \end{aligned} \tag{12}$$

Here, $x_t \in \mathbb{R}^n$ denotes the **system state** at time t (e.g., position, energy level, queue length), which evolves dynamically over time; $u_t \in \mathbb{R}^m$ is the **control action** or decision taken at stage t; in an event-triggered setting, u_t is executed only when an event condition is satisfied; $\omega_t \in \Omega$ represents an **exogenous event** or disturbance (e.g., sensor measurement, environmental condition, communication signal) that may trigger control actions. The sequence $\omega_{0:t} = (\omega_0, \dots, \omega_t)$ denotes the history of observed events up to time t; $\psi : \mathbb{R}^n \times \Omega \to \{\texttt{true}, \texttt{false}\}$ is the **trigger condition** (or **event predicate**) that determines whether an event is activated based on the current state and the observed event signal. If $\psi(x_t, \omega_t) = \texttt{true}$, the corresponding control action $u^{(1)}$ is executed; otherwise, the default action $u^{(0)}$ is applied. This predicate encodes the logical rule that governs event detection in the `ETP` framework;

$f : \mathbb{R}^n \times \mathbb{R}^m \times \Omega \to \mathbb{R}^n$ is the **state transition function** that models the system dynamics. It determines the next state x_{t+1} as a function of the current state x_t, control u_t, and event ω_t; $c_t(x_t, u_t)$ is the **stage cost function** at time t, which quantifies the immediate penalty or energy expenditure incurred by applying control u_t in state x_t. The overall objective $\mathbb{E}[\sum_{t=0}^{T} c_t(x_t, u_t)]$ minimizes the expected cumulative cost over the planning horizon T; $\Phi : \mathbb{R}^n \times \Omega^{t+1} \to \mathbb{R}^m$ denotes the **event-triggered policy**, which maps the current state and event history $\omega_{0:t}$ to a control action u_t. This policy determines how the system reacts when events occur; and $\mathcal{F}_{\text{logic}}$ is the **set of admissible logic-based policies**, typically constrained by temporal logic, Boolean rules, or disjunctive (if–then) conditions. For example, $\Phi \in \mathcal{F}_{\text{logic}}$ may enforce that certain actions are triggered only when safety or threshold conditions are satisfied. The formulation (12) therefore captures both continuous dynamics (via f) and discrete event logic (via Φ and $\mathcal{F}_{\text{logic}}$), combining them in a unified stochastic optimization framework. The expectation operator $\mathbb{E}$ is taken over all possible event realizations $\omega_{0:T}$ according to their underlying probability distribution.

Event-triggered planning arises in **autonomous systems**, **manufacturing**, and **networked control**. In robotic assembly, sensor events trigger path adjustments; in cyber-physical systems, messages are transmitted only when significant deviations occur; and in scheduling, conditional triggers handle machine failures.

The `ETP` Algorithm (=Algorithm 3). The `ETP` (Event-Triggered Planning) algorithm integrates discrete-event logic and optimization via a rule-based update mechanism. This algorithm operates by continuously monitoring external event signals and triggering control actions only when specific logical conditions are satisfied. It begins with the initialization of the system state x_0, event set E, and rule-based policy Φ in ($S0_3$). At each time step, an event ω_t is observed, and the algorithm evaluates the trigger condition $\psi(x_t, \omega_t)$ to determine whether an event has occurred in ($S1_3$). If the condition is true, the control action $u^{(1)}$ is executed; otherwise, the default action $u^{(0)}$ is applied in ($S2_3$). The system state is then propagated forward through the dynamic model $x_{t+1} = f(x_t, u_t, \omega_t)$ in ($S3_3$). As the algorithm evolves, event patterns and system responses are analyzed to adapt the logical policy Φ, enabling learning-based refinement of triggering thresholds or decision rules in ($S4_3$). The process continues until the horizon T is reached, after which the resulting event-triggered policy $\Phi^\star$ and the corresponding trajectory $\{x_t, u_t\}_{t=0}^{T}$ are returned. In summary, `ETP` dynamically balances responsiveness and efficiency by activating control only when necessary events occur, thereby reducing computational and communication costs while maintaining system performance.

Algorithm 3 The Event-Triggered Planning (ETP) Algorithm

Initialization (S0$_3$) Initialize system state x_0, event set E, policy rules Φ, and planning horizon T.

for $t = 0$ to $T - 1$ **do**

Event detection (S1$_3$) Observe event signal ω_t; check if trigger condition $\psi(x_t, \omega_t) = \texttt{true}$.

Action selection (S2$_3$)

$$u_t = \begin{cases} u^{(1)} & \text{if } \psi(x_t, \omega_t) = \texttt{true}, \\ u^{(0)} & \text{otherwise.} \end{cases}$$

State update (S3$_3$) Propagate state $x_{t+1} = f(x_t, u_t, \omega_t)$.

Policy adaptation (S4$_3$) If recurrent event patterns are detected, update rule parameters in Φ via data-driven learning.

end for

Output Return the event-triggered policy $\Phi^\star$ and resulting trajectory $\{x_t, u_t\}_{t=0}^{T}$.

4.4 Probabilistic Programming (ProbP) and ML/RL-Enhanced Methods

In this subsection, we introduce the foundations of ProbP and its integration with **optimization under uncertainty**. We first review classical approaches based on **probabilistic integer nonlinear programming (INLP)**, where randomness in the problem data is modeled through probabilistic (chance) constraints, and expectations in the objective function. To approximate such problems, we present a **Monte Carlo Disjunctive (MCD)** framework that combines scenario sampling with disjunctive logic representations of uncertainty and integrality. This formulation provides a unifying view of probabilistic optimization problems and establishes the algorithmic foundation for handling randomness through sampling-based relaxations. Building on this classical setting, we then discuss how ML and RL techniques can enhance the MCD approach—by improving inference accuracy, constraint satisfaction, and learning-based decision updates—thus bridging traditional ProbP with modern data-driven and adaptive optimization paradigms.

4.4.1 Classical Methods for ProbP

ProbP with disjunctive logic has been applied in various domains such as OR, e.g., see [24, 47, 68]. We first define the set of simple bounds

$$\mathbf{X} := \{x \in \mathbb{R}^n \mid \underline{x} \leq x \leq \overline{x}\} \quad \text{with } \underline{x}, \overline{x} \in \mathbb{R}^n \ (\underline{x} < \overline{x}) \tag{13}$$

on variables $x \in \mathbb{R}^n$ (called the **box**). We here define a probabilistic problem as

$$\begin{array}{ll} \min & \mathbb{E}[f(x, \Theta)] \\ \text{s.t.} & x \in C_{\text{in}}^{\text{pr}}, \end{array} \tag{14}$$

where the probabilistic integer nonlinear feasible set is

$$C_{\text{in}}^{\text{pr}} := \{x \in \mathbf{X} \mid g(x, \Theta) = 0, \ \ h(x, \Theta) \leq 0, \ \ x_i \in s_i \mathbb{Z}, \ \ i \in [n]\},$$

with $\mathbf{X}$ given by (13), Θ a random parameter vector, and $\mathbb{E}$ the expectation operator. The components of the vectors

$$g(x, \Theta) := \big(g_1, \ldots, g_m\big), \qquad h(x, \Theta) := \big(h_1, \ldots, h_p\big)$$

are (possibly non-convex) constraint functions $g_k : C_{\text{in}} \to \mathbb{R}$ for $k \in [m]$ and $h_j : C_{\text{in}} \to \mathbb{R}$ for $j \in [p]$.

To handle uncertainty, probabilistic (chance) constraints are imposed:

$$P\big(g(x, \Theta) = 0\big) \geq 1 - \alpha_g, \qquad P\big(h(x, \Theta) \leq 0\big) \geq 1 - \alpha_h,$$

where α_g and α_h denote acceptable risk levels of constraint violations.

Monte Carlo Relaxation. The expectation in the objective can be approximated by the sample average approximation (SAA):

$$\mathbb{E}[f(x,\Theta)] \;\approx\; \frac{1}{K}\sum_{i=1}^{K} f(x,\Theta_i), \quad \Theta_i \sim \mathcal{D}_\Theta,$$

where $\mathcal{D}_\Theta$ is the distribution of Θ. Similarly, the probabilistic constraints are relaxed using disjunctive representations over sampled scenarios

$$\bigvee_{l\in\Gamma_g} g(x,\Theta_l) = 0, \qquad \bigvee_{l\in\Gamma_h} h(x,\Theta_l) \le 0, \tag{15}$$

where Γ_g and Γ_h are scenario sets generated by sampling.

The integer constraints $x_i \in s_i\mathbb{Z}$ for $i \in [n]$ can be expressed disjunctively as

$$x_i \in \bigcup_{k\,:\,s_ik\in\mathbf{X}_i} \{s_ik\}, \qquad i \in [n]. \tag{16}$$

Thus, the probabilistic INLP problem (14) can be written as the disjunctive SAA problem

$$\begin{aligned} \min \quad & \frac{1}{K}\sum_{i=1}^{K} f(x,\Theta_i) \\ \text{s.t.} \quad & \bigvee_{l\in\Gamma_g} g(x,\Theta_l) = 0, \\ & \bigvee_{l\in\Gamma_h} h(x,\Theta_l) \le 0, \\ & x_i \in \bigcup_{k:\; s_ik\in\mathbf{X}_i} \{s_ik\}, \quad i \in [n]. \end{aligned} \tag{17}$$

The `MCD` algorithm (=Algorithm 4). This algorithm is a generic Monte Carlo Disjunctive framework that approximates probabilistic INLP problems by scenario sampling. It proceeds through four main stages:

(S0$_4$) Initialization. Random scenarios $\Gamma_f = \{\Theta_1,\ldots,\Theta_K\}$ are generated from the distribution of Θ. In the example above, $K = 3$ and $\Gamma_f = \{2,5,9\}$. An initial feasible x is chosen from $\{0,1,\ldots,10\}$.

(S1$_4$) Objective evaluation. The expected objective is approximated by the sample average

$$f_{\mathrm{MC}}(x) = \frac{1}{K}\sum_{i=1}^{K} f(x,\Theta_i). \tag{18}$$

For example, with $x = 5$, $f_{\mathrm{MC}}(5) = (3+0+4)/3 = 7/3$.

(S2$_4$) Constraint checking. Probabilistic constraints are relaxed using disjunctions as in (15). In this facility-location example, there are no nonlinear constraints, so the disjunction is trivially satisfied.

(S3$_4$) Integer feasibility enforcement. The algorithm ensures $x \in \{0, 1, \ldots, 10\}$, i.e., x must belong to the disjunctive set

$$x \in \bigcup_{k=0}^{10} \{k\}.$$

(S4$_4$) Updating best solution. If $f_{\text{MC}}(x) < f_{\text{best}}$, the algorithm updates the best known solution. For the example, starting from $x = 2$ ($f_{\text{MC}} = 10/3$), the algorithm updates to $x = 5$ with $f_{\text{MC}} = 7/3$.

Termination. The process is repeated until all sampled scenarios are evaluated or a convergence criterion is satisfied. In the example, the algorithm correctly identifies $x^* = 5$ as the optimal solution.

Algorithm 4 A Generic Monte Carlo Disjunctive (`MCD`) Framework for Probabilistic INLP

Initialization (S0$_4$) Generate K random scenarios $\Gamma_f = \{\Theta_1, \ldots, \Theta_K\}$ from distribution $\mathcal{D}_\Theta$. Initialize feasible set $x \in \mathbf{X}$, and best solution $(x_{\text{best}}, f_{\text{best}})$ with $f_{\text{best}} = +\infty$.

repeat

Objective evaluation (sample average) (S1$_4$) Compute the Monte Carlo approximation of the expected objective $f_{\text{MC}}(x)$ by (18).

Constraint checking (disjunctive form) (S2$_4$) Check sampled probabilistic constraints by (15). If violated, add corresponding disjunctive cuts to eliminate infeasible scenarios.

Integer feasibility enforcement (S3$_4$) Enforce discrete restrictions for each $i \in [n]$ by (16).

Updating the best solution (S4$_4$) If $f_{\text{MC}}(x) < f_{\text{best}}$ and x is feasible, update $x_{\text{best}} := x$ and $f_{\text{best}} := f_{\text{MC}}(x)$.

until stopping criterion is met (e.g., maximum iterations, convergence tolerance).

Illustrative conditional planning example of Probabilistic INLP is relocated to Appendix 12.2 for reference.

4.4.2 ML/RL-enhancements for ProbP

ML and RL offer significant opportunities to enhance the ProbP framework introduced in Section 4.4. While the Monte Carlo Disjunctive (MCD) algorithm provides a principled approximation for solving probabilistic INLP problems, recent advances in probabilistic ML and RL show how inference and optimization can be improved in terms of scalability, adaptability, and safety.

Probabilistic inference with ML. Step ($S1_4$) of Algorithm 4 computes the sample average approximation $f_{\text{MC}}(x)$ by (18), where $x \in \mathbb{Z}^n$ is the decision vector, $\Theta_i \sim \mathcal{D}_\Theta$ are K i.i.d. samples from the distribution $\mathcal{D}_\Theta$ of the random parameter vector Θ, and $f : \mathbb{Z}^n \times \mathbb{R}^d \to \mathbb{R}$ is the cost function. Deep ProbP frameworks such as Edward [64] or programmable inference languages [39] introduce adaptive surrogates $\hat{f}(x, \phi)$, parameterized by $\phi \in \mathbb{R}^m$, to approximate the expectation $\mathbb{E}[f(x, \Theta)]$. Amortized inference solves

$$\min_{\phi} \ \mathbb{E}_{\Theta \sim \mathcal{D}_\Theta}\Big[\big(f(x,\Theta) - \hat{f}(x,\phi)\big)^2\Big],$$

which reduces variance in the Monte Carlo estimates of $f_{\text{MC}}(x)$.

Structured constraint handling. The chance constraint in (14),

$$P(h(x,\Theta) \le 0) \ge 1 - \alpha_h,$$

with violation tolerance $\alpha_h \in [0,1]$, can be strengthened by modeling the joint distribution of $\Theta \in \mathbb{R}^d$ using composable generative population models (CGPMs) [57]. Instead of enforcing feasibility through disjunctive constraints, one can optimize

$$\min_{x} \ \mathbb{E}_{\Theta \sim p(\Theta)}[\max\{0, h(x,\Theta)\}]\,,$$

where $h : \mathbb{Z}^n \times \mathbb{R}^d \to \mathbb{R}^p$ is the vector of inequality constraints and $p(\Theta)$ is a learned distribution. This reduces the number of disjunctive cuts needed to satisfy the probabilistic constraints.

Safe and constrained RL. The risk thresholds $\alpha_g, \alpha_h \in [0,1]$ in (14) parallel safety constraints in RL. Probabilistic logic shields [70] and LTL-based exploration [11] restrict the feasible action set in state $s \in \mathcal{S}$ to

$$\mathcal{A}_{\text{safe}}(s) = \big\{a \in \mathcal{A} \ \mid \ P(\varphi(s,a)) \ge 1 - \delta\big\},$$

where $\mathcal{A}$ is the full action set, $\varphi(s,a)$ is a temporal-logic safety condition, and $\delta \in [0,1]$ is a risk tolerance parameter. In the MCD setting, feasible solutions x are analogues of safe actions a, and the scenario sets Γ_g, Γ_h correspond to sampled safety conditions.

Policy search as optimization. The update step ($S4_4$) selects the best feasible candidate solution. In RL, parametric policies $\pi_\theta : \mathcal{S} \to \Delta(\mathcal{A})$ with parameters $\theta \in \mathbb{R}^p$ are optimized by gradient ascent on the expected return:

$$\nabla_\theta J(\theta) = \nabla_\theta \, \mathbb{E}_{\pi_\theta}\big[R(x, \Theta)\big],$$

where $R(x, \Theta) = -f(x, \Theta)$ defines the reward function. Black-box variational inference [67] optimizes

$$\max_\theta \; \mathbb{E}_{q_\theta(x)}[f(x, \Theta)],$$

with q_θ a variational distribution over x. This mirrors the `MCD` update step, but enables scalable gradient-based optimization.

Uncertainty representation in AI. Finally, scenario generation in ($S0_4$) draws $\Theta_i \sim \mathcal{D}_\Theta$, where $\mathcal{D}_\Theta$ is a prior distribution on Θ. Probabilistic ML methods [25] refine this by learning a posterior distribution $p(\Theta \mid \mathcal{D})$ from observed data $\mathcal{D}$. Scenarios are then sampled as

$$\Theta_i \sim p(\Theta \mid \mathcal{D}),$$

yielding more faithful uncertainty representation and thereby improving both objective estimation and constraint satisfaction in the `MCD` loop.

In summary, integrating ML and RL into the `MCD` framework strengthens each stage of `MCD` (=Algorithm 4): adaptive inference for objective evaluation in ($S1_4$), composable probabilistic modeling for constraint enforcement in ($S2_4$), RL-style safety guarantees for feasible sets in ($S2_4$/$S3_4$), and black-box policy search for solution updates in ($S4_4$). This synergy extends probabilistic INLP beyond static Monte Carlo relaxations, providing adaptive, safe, and scalable methods for optimization under uncertainty.

Table 1 summarizes how recent advances in ML and RL strengthen each stage of Algorithm 4. Probabilistic ML enriches uncertainty modeling in ($S0_4$), while adaptive inference frameworks such as Edward and programmable inference accelerate objective evaluation in ($S1_4$). Structured abstractions (`CGPM`s) and safe RL mechanisms (probabilistic shields, LTL-based exploration) reinforce constraint satisfaction in ($S2_4$)/($S3_4$). Finally, solution updates in ($S4_4$) align with policy search methods from RL, enabling scalable optimization under uncertainty. Together, these enhancements extend the static Monte Carlo approach into an adaptive, safe, and composable framework.

5 Scheduling and Disjunctive Planning

Scheduling generalizes classical planning to continuous time and resource-allocation domains. Whereas classical planning reasons over logical precon-

Table 1: Enhancements of the MCD framework via ML and RL techniques.

MCD Step	Objective	ML/RL Enhancement
($S0_4$): Scenario generation	Sampling Θ from $\mathcal{D}_\Theta$	Probabilistic ML improves uncertainty representation and sample fidelity [25]
($S1_4$): Objective evaluation	Monte Carlo estimate of $\mathbb{E}[f(x, \Theta)]$	Deep ProbP and programmable inference reduce variance via amortization and decomposition [39, 64]
($S2_4$)–($S3_4$): Constraint enforcement	Disjunctive relaxations for chance constraints	CGPMs provide flexible compositional models of Θ; safe RL with logic shields and LTL constraints ensures probabilistic feasibility [11, 57, 70]
($S4_4$): Solution update	Select/update feasible candidate x	Black-box policy search treats updates as policy gradient steps in probabilistic programs [67]

ditions and effects, scheduling integrates these logical relations with temporal and capacity constraints, producing optimization problems defined over time intervals. Disjunctive programming, introduced by Balas [5], provides the formal basis for representing the mutual exclusivity of operations sharing a resource. In this sense, scheduling remains grounded in logical structure—expressed through precedence and disjunctive relations—while its algebraic or MILP formulations serve only as computational encodings of these logical constraints. The following subsections develop classical scheduling models, their disjunctive representations, and modern learning-based generalizations.

Scheduling is a fundamental problem in operations research, where the objective is to allocate a set of jobs to a set of resources over time in order to optimize a performance measure. Formally, a scheduling problem is defined by a set of jobs $J := [n]$, each with processing requirements, release times, and deadlines, and a set of resources $R := [m]$, each with limited capacity. A feasible schedule specifies the start and completion time of every job on some resource, subject to:

- **Precedence constraints:** Certain jobs must finish before others can start.
- **Resource constraints:** Each resource can handle at most one job at a time (or has a limited capacity shared among jobs).
- **Timing constraints:** Each job must be executed within its release time and deadline window.

The objective function may vary depending on the application, e.g., minimizing makespan (the maximum completion time), minimizing total tardiness, or minimizing resource cost:

$$
\begin{aligned}
\min \quad & C_{\max} + \sum_{j=1}^{m} f_j(y_j) \\
\text{s.t.} \quad & t_j \geq c_i + s_{ij}, && \text{for all } (i,j) \in \text{precedence pairs}, \\
& \sum_{i=1}^{n} x_{ij} \leq 1, && \text{for all } j \in [m], \\
& c_i \leq d_i, && \text{for all } i \in [n], \\
& C_{\max} \geq c_i, && \text{for all } i \in [n], \\
& c_i = t_i + p_i, && \text{for all } i \in [n], \\
& y_j \geq x_{ij}, && \text{for all } i \in [n], j \in [m], \\
& \sum_{j=1}^{m} x_{ij} = 1, && \text{for all } i \in [n], \\
& x_{ij}, y_j \in \{0,1\}, \quad t_i, c_i \geq 0.
\end{aligned}
\tag{19}
$$

Here, x_{ij} is a binary variable equal to 1 if job i is assigned to resource (or machine) j and 0 otherwise. The start time of job i is t_i, and its completion time is $c_i = t_i + p_i$, where p_i denotes the processing time. The variable $C_{\max}$ represents the makespan, i.e., the latest completion time among all jobs.

The parameter s_{ij} denotes the setup or transition time required between the completion of job i and the start of job j. Each resource j has a binary activation variable y_j that equals 1 if it is used by any job. The activation cost is modeled by a function $f_j(y_j)$, often linear or convex (e.g., $f_j(y_j) = c_j y_j$). The constraint $y_j \geq x_{ij}$ ensures that a resource is considered active if at least one job is assigned to it. The model enforces (i) precedence constraints between jobs, (ii) exclusive assignment of jobs to resources, (iii) satisfaction of release and deadline windows, and (iv) non-negativity of start and completion times. The objective minimizes the total makespan plus the aggregate resource usage cost. Additional scheduling examples, including job-shop, workforce, and flow-shop formulations, are compiled in Appendix 12.3.

5.1 Classical Disjunctive Scheduling (`DisjS`) Formulation

Static `DisjS` defines precedence and resource constraints with precedence arcs C, disjunctive arcs D, and makespan minimization $C_{\max}$. It establishes the formal base of disjunctive optimization. This formulation is utilized in Algorithm 6 (S0_6), Algorithm 7 (S0_7), and Algorithm 8 (S0_8) for graph initialization and constraint encoding.

`DisjP` provides a mathematical framework for representing scheduling and sequencing problems in which two operations competing for the same resource cannot overlap in time. Introduced by Balas [5], a disjunctive graph $\mathcal{G} = (V, C \cup D)$ encodes a set of operations V, conjunctive arcs C representing precedence constraints, and disjunctive arcs D representing resource conflicts that must be ordered one way or the other.

Classical Formulation. Each operation $i \in V$ has a start time $x_i \in \mathbb{R}_+$ and processing duration $p_i > 0$. For any two operations (i, j) requiring the same machine, exactly one of the following disjunctions must hold:

$$x_i + p_i \leq x_j \quad \text{or} \quad x_j + p_j \leq x_i,$$

ensuring that no two jobs overlap on a shared resource. Precedence relations $(i, j) \in C$ impose additional constraints $x_i + p_i \leq x_j$. The scheduling objective is often to minimize the makespan

$$C_{\max} = \max_{i \in V}(x_i + p_i),$$

subject to all conjunctive and disjunctive constraints.

Graph Interpretation. In the static job–shop setting, $\mathcal{G}$ is fixed once all jobs and machines are known. Conjunctive arcs form directed acyclic subgraphs for precedence relations, while disjunctive arcs between operations on the same machine are oriented during search to produce feasible schedules. Heuristic or exact algorithms (e.g., branch–bound, tabu search) determine an orientation minimizing $C_{\max}$.

5.2 Scheduling as Sequential Decision-Making

This paragraph reformulates scheduling as an MDP with disjunctive graphs $\mathcal{G}_t$, state transitions, reward $r_t = C_{\max}^{t-1} - C_{\max}^{t}$, and RL policy $\pi_\phi(a_t|s_t)$ optimized via `PPO`. This concept is implemented in Algorithm 6 (S1_6–S3_6), Algorithm 8 (S2_8, S3_8), and Algorithm 5 (S1_5, S3_5).

In practical manufacturing and service systems, the job set and machine availability vary over time due to random arrivals, breakdowns, or changing priorities. The resulting dynamic disjunctive graph is denoted

$$\mathcal{G}_t = (V_t, C_t \cup D_t),$$

where the vertex set V_t and constraint sets C_t, D_t evolve with time t. Each update modifies the feasible region and may introduce new jobs, remove completed operations, or alter machine availability, requiring adaptive rescheduling.

Processing times are $p_i, p_j > 0$, start times x_i, x_j, and the current makespan $C^t_{\max}$ defines the reward

$$r_t = C^{t-1}_{\max} - C^t_{\max},$$

which measures the improvement achieved by the latest scheduling decision. An equivalent definition often used in minimization settings is $r_t = -\Delta C_{\max}$, which simply reverses the sign convention to express makespan reduction as a negative cost rather than a positive reward. Both conventions are equivalent under the policy-gradient formulation.

This dynamic formulation naturally casts scheduling as a sequential decision-making process. At each time step t, the system observes the current graph state $\mathcal{G}_t$, selects an action a_t (such as dispatching an operation or orienting a disjunctive edge), and transitions to a new graph $\mathcal{G}_{t+1}$ representing the updated production state. The decision policy $\pi_\phi(a_t|s_t)$ is trained—typically via `PPO` or another actor–critic method introduced in Section 5.3—to maximize the expected cumulative reward $\sum_t \gamma^t r_t$, thereby minimizing the long-run makespan. This formulation unifies `DisjP`, `DynP`, and RL: disjunctive constraints specify feasible local actions, while RL provides temporal adaptation and generalization under stochastic environments.

5.3 GNN–RL Framework for `DisjS`

Building upon the general ML, RL, and neural-architecture foundations introduced in Section 2, this subsection specializes those formulations for `DisjS`. GNNs encode the structural dependencies among operations and shared resources, while RL agents, trained via `PPO`, optimize adaptive sequencing policies under logical disjunction constraints. Together, these components form the learning-enhanced `DisjP` framework for dynamic and risk-sensitive scheduling. The focus is on how graph-based neural representations and policy-gradient methods are integrated to model precedence, resource sharing, and logical disjunctions, thereby establishing the computational bridge between the theoretical ML/RL foundations and their realization in practical scheduling algorithms.

Logical Disjunction in `DisjP`. The fundamental logical disjunction governing the structure of disjunctive programs is expressed abstractly as

$$(g^1(x) = 0,\ h^1(x) \le 0) \vee (g^2(x) = 0,\ h^2(x) \le 0) \vee \cdots \vee (g^L(x) = 0,\ h^L(x) \le 0), \tag{20}$$

representing L alternative regimes of feasibility. This general disjunction forms the logical core of mixed-integer `DisjP` models and underlies all subsequent examples and algorithms.

Mixed-Integer `DisjP` Formulation. The general logical disjunction in (20) leads to the canonical mixed-integer disjunctive nonlinear program (MIDNP):

$$\begin{array}{ll} \min & f(x) \\ \text{s.t.} & x \in C_{\texttt{DisjP}}, \end{array} \tag{21}$$

where $f : C_{\texttt{DisjP}} \subseteq \mathbb{R}^n \to \mathbb{R}$ may be nonlinear or nonconvex, and $C_{\texttt{DisjP}}$ denotes the disjunctive feasible region. This abstract formulation unifies a wide range of combinatorial, scheduling, and planning problems under a single optimization framework.

The feasible region of (21) is given by the disjunctive set

$$C_{\texttt{DisjP}} := \left\{ x \in \mathcal{X} \mid \bigvee_{l=1}^{L} \big(g^l(x) = 0,\ h^l(x) \le 0\big),\ x_i \in s_i\mathbb{Z} \text{ for } i \in I \right\}, \tag{22}$$

where $\mathcal{X} \subseteq \mathbb{R}^n$ is a bounded domain, $I \subseteq [n]$ is the set of integer variables, and each pair (g^l, h^l) defines the equality and inequality constraints associated with disjunct $l \in [L]$. This general form serves as the mathematical foundation for all `DisjP` models and their dynamic and learning-based extensions.

Risk-Sensitive and Differentiable Logic. Distributional RL models the return as a random variable $Z^\pi(s,a)$ and propagates its distribution by $\mathcal{T}^\pi Z(s,a) = R(s,a) + \gamma Z^\pi(s',a')$. A risk-sensitive objective uses the Conditional Value-at-Risk:

$$\texttt{CVaR}_\alpha(Z) = \mathbb{E}[Z \mid Z \le F_Z^{-1}(\alpha)], \tag{23}$$

which measures expected loss in the worst α fraction of cases. For neural-symbolic reasoning, logical disjunctions are represented by differentiable operators:

$$f_{\text{disj}}(x) = 1 - \prod_i (1 - m_i x_i), \qquad m_i = \sigma(c w_i), \tag{24}$$

where m_i is a learned weight and $c > 0$ controls smoothness. These operators enable continuous optimization over logical constraints. Equations (3)–(24) constitute the core mathematical framework reused throughout the learning-based scheduling and disjunctive-planning models.

We here introduce GNN-based RL models for representing operations and machines as graph nodes with embeddings $h_v^{(t)}$ and optimizing scheduling via `PPO`. It is directly realized in ($S0_6$)–($S3_6$) of Algorithm 6 and reused in ($S1_8$) of Algorithm 8. Recent work integrates GNN and RL methods into this framework. GNNs encode the evolving disjunctive graph $\mathcal{G}_t$ through node embeddings $h_v^{(t)}$ that capture machine and job states, while RL agents learn dispatching policies $\pi_\phi(a_t|s_t)$ that select the next feasible operation or orientation to reduce

makespan. The GNN and PPO formulations follow those introduced in Section 2. These hybrid methods enable size-agnostic and real-time decision-making under stochastic environments.

This background establishes the classical and dynamic foundations of DisjS on which subsequent sections build, including dynamic GNN–RL integration, risk-sensitive RL, and differentiable neural logic for disjunctive reasoning.

Risk-Sensitive and Robust Scheduling. We here introduce risk-sensitive RL using the CVaR metric and distributional Bellman operators to manage uncertainty in scheduling. These ideas are applied in ($S0_5$)–($S3_5$) of Algorithm 5 and referenced in ($S1_9$) of Algorithm 9 for risk-aware recursion. Risk-sensitive formulations employ the Bellman operator and CVaR metric from Section 5.3. The algorithmic structure for distributional RL remains identical. Algorithm 5 outlines how distributional RL introduces risk sensitivity into DisjS. In ($S0_5$), the policy parameters ϕ, the return distribution $Z^\pi(s,a)$, and the confidence level α for the CVaR measure are initialized, together with precedence and disjunctive constraints. In ($S1_5$), multiple simulated trajectories are generated under the current policy π_ϕ, and corresponding returns $Z = \sum_t \gamma^t r_t$ are computed to capture stochastic variability in rewards. During ($S2_5$), the distributional Bellman operator $\mathcal{T}^\pi$ updates the estimate of the return distribution by propagating uncertainty through successive state–action pairs. Finally, ($S3_5$) computes the CVaR metric and updates the policy parameters to minimize expected downside risk, thereby biasing the learning process toward safer scheduling strategies. Overall, Algorithm 5 enables risk-aware optimization over disjunctive constraints, ensuring robust performance under stochastic operating conditions.

RL-Based Scheduling Policy. This paragraph outlines the unified RL-enhanced DisjS paradigm using GNN, attention, and dynamic RL architectures. It defines key notation for π_ϕ, V_ϕ, and $\mathcal{G}_t$. This formulation underpins Algorithm 6 ($S0_6$–$S3_6$), Algorithm 7 ($S1_7$–$S3_7$), Algorithm 8 ($S1_8$–$S3_8$), and Algorithm 5 ($S1_5$–$S3_5$).

While classical DisjP provides a rigorous convexification-based framework for logical and combinatorial optimization, recent advances in ML and RL introduce data-driven strategies that improve both representation and policy optimization. Disjunctive formulations often result in large-scale mixed-integer programs that are computationally demanding. ML and RL replace explicit combinatorial enumeration with differentiable approximation and adaptive policy learning, as detailed below and summarized in Table 2.

Algorithm 5 Distributional RL framework for risk-aware `DisjS`

Initialization

($S0_5$) Initialize policy parameters ϕ, return distribution $Z^\pi(s,a)$, and confidence level α for `CVaR`. Define precedence and disjunctive constraints.

repeat

Sampling and return estimation

($S1_5$) Simulate multiple trajectories under policy π_ϕ. Compute sampled returns $Z = \sum_t \gamma^t r_t$.

Distributional Bellman update

($S2_5$) Apply operator $\mathcal{T}^\pi Z(s,a) = R(s,a) + \gamma Z(s',a')$ to estimate the return distribution.

Risk-sensitive optimization

($S3_5$) Compute $\texttt{CVaR}_\alpha(Z) = \mathbb{E}[Z | Z \leq F_Z^{-1}(\alpha)]$. Update policy parameters ϕ to minimize negative `CVaR` (risk-averse objective).

until policy converges to stable risk-sensitive schedule

Table 2: Representative ML/RL approaches enhancing `DisjP`.

Domain	Model & Key Formulation	Reference
Job-shop scheduling	GNN with `PPO`; message passing (3), policy update (6)	[48]
Large-scale JSSP	Transformer-based Deep RL; attention dependency (4)	[18]
Dynamic JSSP	Size-agnostic GNN; stochastic disjunction (20)	[36]
Chemical scheduling	Distributional RL; `CVaR` risk objective (23)	[45]
Neuro-symbolic logic	Differentiable ILP with logic operators (24)	[14]

Throughout this subsection, $\mathcal{G} := (V, C \cup D)$ denotes the disjunctive graph, where V is the set of operations, C the conjunctive precedence edges, and D the disjunctive edges representing shared resources. The makespan is denoted by $C_{\max}$, and the reward signal is defined as $r_t = C_{\max}^{t-1} - C_{\max}^{t}$, which is positive when the makespan decreases. Equivalently, one may define $r_t = -\Delta C_{\max}$ to maintain a minimal interpretation. Embedding dimensions are written as d for node features, d_h for hidden embeddings, and d_k for attention-key dimensions. The policy is represented by $\pi_\phi(a_t|s_t)$ (actor parameters ϕ), the critic/value network by $V_\phi(s_t)$ where applicable, and θ denotes neural-encoder or GNN weights used for representation learning. When reused across models, distinct losses are indicated by subscripts, e.g., L_{PPO} for policy-gradient training and

L_{logic} for differentiable neural logic learning. Time-dependent disjunctive graphs are denoted by $\mathcal{G}_t$ and evolve as new decisions or environment changes occur.

All learning-based disjunctive algorithms presented in this section share a unified iterative structure consisting of (i) graph encoding, (ii) policy sampling, (iii) reward evaluation, and (iv) parameter update. Each subsequent algorithm specializes this loop with different representations (GNN, attention, dynamic adaptation, or risk sensitivity).

In scheduling problems, the disjunctive structure is encoded as a directed graph $\mathcal{G} = (V, C \cup D)$. Each node $v \in V$ carries a feature vector x_v, and GNN message passing and policy optimization use the unified formulations from Section 5.3. The stochastic policy $\pi_\phi(a_t|s_t)$ is trained by `PPO` to optimize the expected makespan reduction.

Building upon this foundation, Algorithm 6 provides a step-by-step overview of how RL enhances `DisjS` through graph representation and policy optimization. In ($S0_6$), the disjunctive graph $\mathcal{G} = (V, C \cup D)$ and model parameters (θ, ϕ) are initialized, establishing the structural and decision-making components of the system. In ($S1_6$), the GNN performs message passing across conjunctive and disjunctive edges to compute node embeddings that encode operation and machine dependencies. ($S2_6$) maps the current graph state s_t to a scheduling action a_t sampled from the stochastic policy $\pi_\phi(a_t|s_t)$, with an immediate reward measuring the reduction in makespan. Finally, ($S3_6$) updates the parameters through the `PPO` objective $L(\phi)$, improving the expected scheduling performance while maintaining stable policy updates. Overall, Algorithm 6 learns a neural policy that implicitly resolves disjunctive alternatives—selecting operation orderings—without explicit combinatorial enumeration.

Graph-Based Neural Representation. This paragraph describes Transformer-based attention $\text{attn}(Q, K, V)$ for capturing global dependencies in disjunctive graphs. It is employed in Algorithm 7 ($S1_7$–$S3_7$). The attention mechanism for global dependency modeling follows the definition in Section 5.3. The Transformer encoder captures relational dependencies among operations, and the `PPO`-based training minimizes expected makespan.

Algorithm 7 summarizes how attention-based deep RL captures long-range dependencies in disjunctive graph scheduling. In ($S0_7$), the disjunctive graph $\mathcal{G} = (V, C \cup D)$ is constructed, `node2vec` embeddings x_i are generated, and the Transformer parameters θ are initialized. In ($S1_7$), multihead attention layers compute contextual dependencies among operations through $\text{attn}(Q, K, V)$, producing encoded representations Z that reflect both conjunctive and disjunctive relations. During ($S2_7$), the decoder sequentially generates a schedule by sampling $\pi_t \sim \pi_\theta(\pi_t|\pi_{1:t-1}, X)$, guided by the attention-weighted embeddings. Finally, ($S3_7$) updates the model via policy-gradient optimization to minimize the expected makespan $C_{\max}$. Overall, Algorithm 7 learns a scalable attention

Algorithm 6 GNN-based PPO Framework for DisjS

Initialization

($S0_6$) Initialize the disjunctive graph $\mathcal{G} = (V, C \cup D)$, the GNN parameters θ, the policy parameters ϕ, and the PPO constants (ϵ, γ).

repeat

State encoding and propagation

($S1_6$) Compute node embeddings using the message-passing rule $h_v^{(k)}$ (defined by (3)), and propagate messages along conjunctive and disjunctive arcs.

Policy decision

($S2_6$) Form current state s_t from $\mathcal{G}_t$. Sample scheduling action $a_t \sim \pi_\phi(a_t|s_t)$ (dispatch an operation). Apply it and observe reward $r_t = C_{\max}^{t-1} - C_{\max}^{t}$.

Policy update

($S3_6$) Update ϕ via the PPO objective $L(\phi)$ (defined by (6)) and backpropagate through GNN to update θ.

until convergence of scheduling policy π_ϕ

mechanism that replaces explicit enumeration of disjunctive constraints with differentiable relevance estimation among operations.

In this subsection, the parameter symbol θ refers to the encoder–decoder weights of the attention model, distinct from the actor parameters ϕ used in GNN–based PPO formulations.

Dynamic Graph Updates and Online Adaptation. This paragraph defines adaptive GNN-RL scheduling for time-varying disjunctive graphs $\mathcal{G}_t$ with real-time policy updates via PPO. It is implemented in Algorithm 8 ($S0_8$–$S3_8$).

The dynamic GNN-RL framework adapts to time-varying graphs $\mathcal{G}_t = (V_t, C_t \cup D_t)$, with rewards $r_t = C_{\max}^{t-1} - C_{\max}^{t}$. Algorithm 8 remains as in the original, using PPO training as described in Section 5.3. This algorithm describes how RL handles dynamic DisjS in stochastic environments. In ($S0_8$), the system initializes with an evolving environment that may include new job arrivals or machine breakdowns, constructing the initial graph $\mathcal{G}_0 = (V_0, C_0 \cup D_0)$ and initializing model parameters (θ, ϕ). In ($S1_8$), the GNN continuously updates node embeddings $h_v^{(t)}$ as the disjunctive graph $\mathcal{G}_t$ changes, allowing the model to capture temporal variations in machine and job states. During ($S2_8$), the policy $\pi_\phi(a_t|s_t)$ selects a feasible operation that satisfies the disjunction constraint $x_i + p_i \leq x_j$ or $x_j + p_j \leq x_i$, ensuring machine exclusivity while optimizing makespan reduction. Finally, ($S3_8$) performs online learning via PPO updates using new

Algorithm 7 Attention-based deep RL for disjunctive graph embedding

Initialization

($S0_7$) Construct disjunctive graph $\mathcal{G} = (V, C \cup D)$, compute `node2vec` embeddings x_i, initialize Transformer encoder–decoder parameters θ.

repeat

Encoding via attention

($S1_7$) Compute multihead attention $\text{attn}(Q, K, V)$ by (4) and obtain encoded context $Z = \text{Enc}_\theta(X)$ by (5).

Sequential decoding

($S2_7$) Iteratively decode operation sequence $\pi_t \sim \pi_\theta(\pi_t|\pi_{1:t-1}, X)$ with contextual embeddings Z and a pointer-attention mechanism following [69], which assigns attention-based probabilities to discrete scheduling actions.

Policy optimization

($S3_7$) Compute the reward $r_t = C^{t-1}_{\max} - C^t_{\max}$; update θ by the policy gradient $\nabla_\theta E[C_{\max}]$.

until minimum expected makespan or convergence

experience samples, enabling continual policy adaptation as system conditions evolve. Overall, Algorithm 8 achieves size-agnostic, real-time decision-making by combining disjunctive constraint reasoning with adaptive RL.

Algorithm 8 Dynamic `DisjS` with GNN and `PPO`

Initialization

($S0_8$) Initialize an environment with stochastic job arrivals and machine breakdowns. Construct an initial graph $\mathcal{G}_0 = (V_0, C_0 \cup D_0)$. Initialize parameters (θ, ϕ) for GNN and policy.

repeat

Graph update and embedding

($S1_8$) Update $\mathcal{G}_t$ as jobs arrive or machines fail. Propagate messages to update embeddings $h_v^{(t)} = f_\theta\big(h_v^{(t-1)}, \{h_u^{(t-1)}, e_{uv}\}_{u\in\mathcal{N}(v)}\big)$, ensuring edge features e_{uv} are included as in (3).

Adaptive decision

($S2_8$) Select feasible operation $a_t \sim \pi_\phi(a_t|s_t)$ respecting disjunction $x_i + p_i \le x_j$ or $x_j + p_j \le x_i$. Execute a_t and collect reward $r_t = C^{t-1}_{\max} - C^t_{\max}$.

Online learning

($S3_8$) Update (θ, ϕ) using `PPO` with on-policy samples; continue scheduling adaptively under dynamic changes.

until end of production horizon

Building on the dynamic GNN–RL and risk-sensitive formulations above, the next section formalizes `DisjP` as a recursive, multistage optimization framework, extending the static disjunctive model into a temporal domain.

5.4 Dynamic Disjunctive Programming (`DisjP`) as Recursive Optimization

This subsection unifies `DisjP` and `DynP` through recursive Bellman equations and feasible disjunctive action sets $\mathcal{A}(s_t)$. It forms the core of Algorithm 9 ($S0_9$)–($S3_9$).

Although `DynP` here refers to dynamic programming (as in Bellman's formulation) rather than `DisjP`, the two are closely related. Both frameworks employ logical disjunctions to define feasible sets—`DisjP` in a static optimization setting, and `DynP` through recursive decisions across time stages.

`DynP`, introduced by Bellman in the 1950s, is a recursive optimization paradigm based on the principle of optimality: an optimal policy has the property that, regardless of the initial state and initial decision, the remaining decisions must form an optimal policy with respect to the resulting state. The classical framework is well-suited for sequential decision-making, but modern problems in operations research often require extensions to incorporate uncertainty, constraints, logical disjunctions, or multiple objectives. These extensions are particularly important when combining `DynP` with `DisjP`, where at each stage of the decision process the feasible set may be defined by logical alternatives.

Unified View. The combination of disjunctive and `DynP` formulations can be viewed as a multistage extension of the mixed-integer disjunctive set (22). At each time stage t, a local disjunctive system defines the feasible action set, while the recursion of `DynP` propagates this logic through time. Here, (g_t^l, h_t^l) denote the time-indexed counterparts of the disjunctive components (g^l, h^l) introduced in (20), allowing each stage t to activate distinct logical regimes while preserving the same structural syntax. Formally, let $s_t \in \mathcal{S}$ denote the system state at stage t, and define the disjunctive feasible action set as

$$\mathcal{A}(s_t) := \left\{ a_t \;\middle|\; \bigvee_{l=1}^{L} \big(g_t^l(s_t, a_t) = 0,\ h_t^l(s_t, a_t) \le 0\big),\ a_{ti} \in \mathbb{Z} \text{ for all } i \in I \right\}. \quad (25)$$

The corresponding value function satisfies the recursive Bellman-type relation

$$V_t(s_t) = \min_{a_t \in \mathcal{A}(s_t)} \Big\{ c_t(s_t, a_t) + \mathbb{E}_{\xi_t}[V_{t+1}(F_t(s_t, a_t, \xi_t))] \Big\}, \quad V_{T+1}(s_{T+1}) = 0. \quad (26)$$

Minimizing the stage cost c_t is equivalent to maximizing the cumulative reward $r_t = -c_t$, consistent with the reinforcement-learning formulation in Section 5.3.

Here, ξ_t denotes exogenous stochastic disturbances at stage t, representing random events such as demand shocks or machine failures, and the notation and recursive structure are consistent with the MDP tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \Pr, R, \gamma)$ introduced in Section 5.3, where the transition F_t corresponds to $\Pr(s'|s,a)$ and the stage cost $c_t(s_t, a_t)$ plays the role of the reward $R(s,a)$. Thus, the Bellman recursion in (26) generalizes the standard RL formulation $V(s) = \mathbb{E}_{\pi_\theta}[R(s,a) + \gamma V(s')]$ to disjunctive feasible regions. Equation (26) is equivalent to the expectation $\mathbb{E}_{\Pr(s'|s,a)}[V_{t+1}(s')]$ used in RL Bellman updates, highlighting the correspondence between `DynP` recursion and RL value iteration.

Algorithmic Formulation. The recursive structure defined by (25) and (26) can be formalized as a stage-wise algorithmic process. At each decision stage t, the feasible action set $\mathcal{A}(s_t)$ in (25) represents disjunctive alternatives among local regimes, while the Bellman recursion in (26) propagates value estimates backward in time. Algorithm 9 summarizes this procedure, providing a constructive view of dynamic `DisjP` and its equivalence to RL when value functions and policies are approximated by neural networks, as defined in Section 5.3.

Algorithm 9 Recursive Dynamic `DisjP` Procedure

Initialization

($\mathrm{S0}_9$) Define time horizon T, initial state s_1, stage cost $c_t(s_t, a_t)$, transition mapping $F_t(s_t, a_t, \xi_t)$, and disjunctive feasible action set $\mathcal{A}(s_t)$ as in (25). Initialize the terminal value $V_{T+1}(s_{T+1}) = 0$.

for $t = T, \ldots, 1$ **do**

Stage-wise recursion

($\mathrm{S1}_9$) For each state s_t, compute the value function by (26).

Policy extraction

($\mathrm{S2}_9$) Record the minimizing action as the optimal stage policy:

$$\pi_t^*(s_t) = \operatorname*{argmin}_{a_t \in \mathcal{A}(s_t)} \{c_t(s_t, a_t) + \mathbb{E}[V_{t+1}(F_t(s_t, a_t, \xi_t))]\}.$$

end for

Output

($\mathrm{S3}_9$) Return the sequence of optimal value functions $\{V_t\}_{t=1}^T$ and corresponding policies $\{\pi_t^*\}_{t=1}^T$.

Interpretation and Properties. Each stage t defines a disjunctive subproblem similar in form to (21), parameterized by the current state s_t. The transition mapping F_t governs the propagation of states, while $\mathcal{A}(s_t)$ introduces multiple local regimes or modes of operation. This recursive structure effectively generates a tree of feasible trajectories, where each branch corresponds to a sequence

of activated disjunctive alternatives $\{l_1, l_2, \ldots, l_T\}$. Even when each local subproblem is convex, their union yields a globally nonconvex feasible set.

Convexification techniques from `DisjP` (e.g., lift-and-project or intersection cuts) can be applied locally at each stage to tighten $\mathcal{A}(s_t)$. Likewise, decomposition methods from `DynP` (e.g., value-function approximation, policy iteration) can propagate relaxations temporally. In practical implementations, these approximations often employ `PPO` optimization as detailed in Section 5.3, linking temporal recursion with data-driven policy updates. In this sense, convexification in `DisjP` and decomposition in `DynP` act as dual processes—one spatial, the other temporal. This correspondence highlights how `DisjP` operates as a spatial decomposition, whereas `DynP` performs temporal decomposition across decision stages.

Under bounded rewards and Lipschitz-continuous value functions, the neural approximation of $V_t(s_t)$ or $\pi_t(s_t)$ converges in expectation to a fixed point of the Bellman operator [62]. This ensures theoretical consistency between the learned policies and the recursive structure of (26).

Applications and Computational Aspects. This paragraph presents real-world applications of dynamic disjunctive models in inventory, energy, and scheduling systems, showing practical optimization relevance. These ideas appear in Algorithm 9 (S0_9) and Algorithm 8 (S0_8).

Integrating disjunctive logic into sequential models enables realistic multi-period formulations. In inventory control, the state s_t represents stock levels, and $\mathcal{A}(s_t)$ encodes contract-dependent procurement rules. In energy management, disjunctions capture switching between generator configurations or power-market regimes. In multi-objective scheduling, they represent alternative sequencing or resource-assignment constraints across time. Such models yield multistage mixed-integer disjunctive programs, often solved via branch-and-bound across the temporal dimension and cutting-plane or relaxation techniques spatially.

Connection to Learning-Based Methods. This paragraph connects disjunctive `DynP` to RL via value-function approximation and stochastic policy optimization under `PPO`. It underlies Algorithm 9 (S1_9, S2_9) and Algorithm 5 (S3_5). Dynamic disjunctive programs form a natural bridge to modern RL and approximate `DynP`. When the functions c_t, F_t, or the disjunctive components (g_t^l, h_t^l) are unknown or high-dimensional, neural approximators can estimate value functions $V_t(s_t)$ or learn policies $a_t = \pi_t(s_t)$. This viewpoint unifies model-based `DynP` with data-driven RL, where learned value functions approximate disjunctive value landscapes or implicit convexifications obtained through experience. Consequently, RL can be interpreted as a stochastic, data-driven implementation of dynamic `DisjP`. Risk-sensitive variants build upon

the CVaR-based objectives and distributional Bellman operators defined in Section 2, extending them to DisjS under uncertainty. See Figure 13 in Appendix 12.3 for a conceptual visualization of the proposed dynamic DisjP framework.

6 Backward Induction (BI)

In this section, we begin by outlining the theoretical foundations of BI as a systematic framework for solving multi-stage decision problems through recursive reasoning from the final stage to the initial one. We then illustrate the classical BI algorithm with an example and formalize its generic computational procedure. Building upon this foundation, the subsequent subsections explore how recent advances in ML and RL extend the classical BI framework—introducing neural approximations, forward–backward reasoning, uncertainty modeling, and imitation-based learning—to enhance its scalability and applicability in complex, uncertain environments.

6.1 Classical BI methods

In this section, we introduce a multi-stage decision problem and describe a generic BI algorithm to solve such a problem. For more details about various problems and algorithms, see [7, 35, 56, 63].

To solve multi-stage decisions, we propose a generic BI algorithm, which works backwards from the final stage to the initial stage, while computing the optimal decisions at each stage based on the value of subsequent stages. We consider a multi-stage decision problem with T stages whose decisions are made sequentially and whose nonlinear objectives, integer decision variables, and constraints depend on these T states and random variables Θ, in the form of

$$\begin{aligned}
\min_{x_1, x_2, \ldots, x_T} \quad & \mathbb{E}\Big[\textstyle\sum_{t=1}^{T} f_t(s_t, x_t, \Theta_t)\Big] & & \\
\text{s.t.} \quad & s_{t+1} = f_s(s_t, x_t, \Theta_t), & & \text{for all } t \in [T-1], \\
& P\big(g_t(s_t, x_t, \Theta_t) \le 0\big) \ge 1 - \alpha_t, & & \text{for all } t \in [T], \\
& x_t \in X_t(s_t), & & \text{for all } t \in [T], \\
& x_{ti} \in s_i \mathbb{Z}, & & \text{for all } i \in [n].
\end{aligned} \tag{27}$$

Here, $t \in [T]$ denotes the stage index, the initial state s_1 is assumed to be given, $\mathbb{E}[\cdot]$ denotes the expectation over the random variables Θ_t, which represent exogenous disturbances or uncertainties realized at each stage t. The stage-wise cost function $f_t(s_t, x_t, \Theta_t)$ specifies the immediate cost (or negative reward) given current state s_t, decision x_t, and random outcome Θ_t. The transition function $f_s(s_t, x_t, \Theta_t)$ maps the current state, action, and uncertainty

to the next state s_{t+1}. The nonlinear constraint function $g_t(s_t, x_t, \Theta_t)$ defines stage-dependent feasibility requirements, and α_t is the allowable probability of violation for those constraints.

The formulation in (27) is a stochastic multi-stage optimization model expressed in functional and set-theoretic form rather than as a mixed-integer program. The integrality constraint $x_{ti} \in s_i\mathbb{Z}$ simply encodes discrete decision granularity and does not alter the logical structure of the feasible sets $X_t(s_t)$.

The feasible set $X_t(s_t)$ collects all admissible decisions at stage t, possibly including continuous, discrete, and integer variables. The notation $x_{ti} \in s_i\mathbb{Z}$ indicates that the variable x_{ti} must take integer multiples of a specified step size s_i. Together, this formulation describes a generic stochastic multi-stage optimization problem underlying classical `BI` methods. A detailed illustrative example of the two-stage inventory problem solved via `BI` is provided in Appendix 12.4.

Algorithm 10 is a generic `BI` procedure that solves multi-stage problems by recursion. It works backwards from the final stage T to the first stage 1, computing optimal actions at each step. The stages are:

(S0$_{10}$) Initialization at the final stage. At $T = 2$, compute

$$V_2(s_2) = \min_{x_2 \in X_2(s_2)} \frac{1}{K} \sum_{k=1}^{K} f_2(s_2, x_2, \Theta_2^k).$$

In the example, given realized demand $\Theta \in \{1, 2\}$, the decision x_2 minimizes holding/penalty cost.

(S1$_{10}$) Recursive steps. At $t = 1$, compute

$$V_1(s_1) = \min_{x_1 \in X_1(s_1)} \frac{1}{K} \sum_{k=1}^{K} f_1(s_1, x_1, \Theta_1^k) + V_2(s_2).$$

In the example, x_1 is chosen anticipating the distribution of Θ_1 and the future cost V_2. Monte Carlo sampling estimates the expectation. K denotes the number of Monte Carlo samples used to approximate stage expectations.

(S2$_{10}$) Base case. At the initial state s_1, the algorithm outputs x_1^* that minimizes expected ordering cost plus future costs. In the example, if $c = 1$, $h = 1$, $p = 5$, then `BI` recommends ordering one unit ($x_1 = 1$) to balance holding and penalty risks.

The algorithm stops when all $t \in [T]$ stages have been solved. `BI` ensures that the decision at each stage accounts for future uncertainty and costs. In general, the backward recursion governing `BI` can be summarized as

$$V_t(s_t) = \min_{x_t \in X_t(s_t)} \mathbb{E}_{\Theta_t}[f_t(s_t, x_t, \Theta_t) + V_{t+1}(f_s(s_t, x_t, \Theta_t))], \quad t = T-1, \ldots, 1,$$

with the terminal condition $V_T(s_T) = \min_{x_T \in X_T(s_T)} \mathbb{E}_{\Theta_T}[f_T(s_T, x_T, \Theta_T)]$. This recursion defines the theoretical core of the BI algorithm.

Algorithm 10 Two-Stage BI Example (Inventory Problem)

($S0_{10}$) Solve stage $T = 2$ for each realized demand $\Theta \in \{1, 2\}$ to compute $V_2(s_2)$.
($S1_{10}$) At stage $t = 1$, evaluate $V_1(s_1)$ for each feasible x_1 using

$$V_1(s_1) = f_1(s_1, x_1) + \mathbb{E}[V_2(s_2)].$$

($S2_{10}$) Output $x_1^* = \operatorname{argmin} V_1(s_1)$.

6.2 ML/RL–enhancements for BI methods

While classical BI provides a systematic framework for solving multi-stage decision problems, recent advances in ML and RL offer promising directions to enhance its scalability, adaptability, and applicability in complex, uncertain environments.

First, NNs have been shown to approximate BI reasoning in game-theoretic contexts. Spiliopoulos [59] demonstrates that NN agents can learn to backward induce in multi-stage games, capturing both rational and bounded-rational behaviors. This suggests that ML can serve as a realistic cognitive model to approximate BI reasoning where exact computation is infeasible.

Second, the integration of RL with BI has given rise to new algorithms that combine forward exploration with backward reasoning. Edwards et al. [20] introduce Forward-Backward RL (FBRL), which leverages imagined backward trajectories from goal states to propagate sparse rewards, thereby accelerating learning compared to standard forward-only methods. Similarly, Bai et al. [4] propose OB2I, which incorporates uncertainty-aware exploration into backward updates, resulting in improved sample efficiency in deep RL.

Third, hybrid approaches combine BI with supervised or imitation learning to produce human-like or domain-specific strategies. For example, Tung et al. [66] integrate BI with NNs to develop a billiard AI that mimics human players' strategies. Their method learns heuristic-based backward search rules enhanced with ML predictions, yielding behavior judged more human-like compared to physics-only simulations.

Beyond strategic games, BI has inspired novel applications in AI systems. Lee and Kim [34] apply BI-inspired inverse mappings for deep image search, demonstrating that inductive knowledge in neural embeddings can be leveraged for conditional retrieval. More broadly, Harris and Dunham [49] argue that valid

inductive principles, grounded in causality, are necessary to avoid overfitting and enhance the generalizability of ML methods that complement `BI`. Meanwhile, Srivastava et al. [60] highlight the importance of backward compatibility in ML systems, emphasizing that improvements should not introduce regressions in earlier stages, a principle conceptually aligned with `BI`'s recursive consistency.

Finally, theoretical extensions of `BI` itself have benefited from ML-inspired rationalizability concepts. Meier and Perea [43] propose a synthesis of forward and `BI`, rationalizing behavior under imperfect information in a way that resonates with RL's balance of exploration and exploitation.

Connection to the `BI` Algorithm. The generic `BI` procedure described earlier (Algorithm 10) can be directly enhanced by learning-based methods. At the final stage ($\mathrm{S0}_{10}$), NNs can approximate value functions when exact computation is infeasible [59]. Formally, instead of

$$V_T(s_T) = \min_{x_T \in X_T(s_T)} \mathbb{E}_{\Theta_T}[f_T(s_T, x_T, \Theta_T)],$$

an NN $V_T(s_T;\theta)$ with parameters $\theta \in \mathbb{R}^d$ is trained to minimize the empirical risk

$$\min_{\theta} \ \frac{1}{K} \sum_{k=1}^{K} \Big(V_T(s_T;\theta) - f_T(s_T, x_T, \Theta_T^k) \Big)^2,$$

where K is the number of Monte Carlo samples, Θ_T^k are sampled realizations of the random variable Θ_T, and f_T is the terminal-stage cost function.

During recursive steps ($\mathrm{S1}_{10}$), RL techniques improve exploration and expectation estimation. For example, Forward-Backward RL [20] augments the Bellman recursion:

$$Q_t(s_t, x_t) \ \leftarrow \ f_t(s_t, x_t) + \gamma\, \mathbb{E}_{s_{t+1}} \Big[\max_{x_{t+1}} Q_{t+1}(s_{t+1}, x_{t+1}) \Big],$$

where $Q_t(s_t, x_t)$ is the state–action value at stage t, $\gamma \in [0, 1)$ is a discount factor, and the expectation is taken over next states s_{t+1} induced by the transition f_s. Backward rollouts simulate imagined trajectories from goal states to accelerate credit assignment. Similarly, `OB2I` [4] introduces an uncertainty bonus:

$$Q_t^+(s_t, x_t) \ = \ Q_t(s_t, x_t) + \alpha\, B(s_t, x_t),$$

where $Q_t^+(s_t, x_t)$ is the optimistic Q-value, $\alpha > 0$ is a tunable exploration coefficient controlling optimism level, and $B(s_t, x_t)$ is the epistemic uncertainty bonus defined as

$$B(s_t, x_t) = \sqrt{\frac{1}{K} \sum_{k=1}^{K} \Big(Q_t^k(s_t, x_t) - \bar{Q}_t(s_t, x_t) \Big)^2},$$

with $\{Q_t^k\}_{k=1}^K$ denoting K bootstrapped Q-value estimates and $\bar{Q}_t$ their mean. This mechanism encourages exploration in poorly visited regions.

At the base case ($S2_{10}$), imitation- and heuristic-based methods guide decision initialization. Instead of solving

$$x_1^* = \underset{x_1 \in X_1(s_1)}{\operatorname{argmin}} \ \mathbb{E}_{\Theta_1}[f_1(s_1, x_1, \Theta_1) + V_2(s_2)],$$

a stochastic policy $\pi_\phi(x_1|s_1)$ parameterized by $\phi \in \mathbb{R}^p$ can be trained via supervised imitation learning:

$$\min_{\phi} \ \mathbb{E}_{(s_1, x_1^*) \sim \mathcal{D}}\big[-\log \pi_\phi(x_1^*|s_1)\big],$$

where $\mathcal{D}$ is a dataset of expert demonstrations (s_1, x_1^*).

Finally, causality-driven induction [49] and backward compatibility checks [60] can be integrated by constraining updated value functions $\tilde{V}_t$ not to regress on validated states:

$$\tilde{V}_t(s_t) \approx V_t(s_t), \quad \forall s_t \in \mathcal{S}_{\text{past}},$$

where $\mathcal{S}_{\text{past}}$ is the set of previously validated states. Hence, ML and RL augment the classical backward recursive algorithm with approximation ($V_T \approx V_T(\cdot;\theta)$), reward shaping (via backward rollouts), uncertainty modeling (Q^+ with UCB-style bonuses), and human-like priors (imitation-based initialization).

Overall, ML and RL enrich `BI` by embedding learning-based mechanisms into its recursion, making it tractable for high-dimensional, stochastic, and real-world decision problems.

Table 3: ML/RL Enhancements to BI Aligned with Algorithmic Steps

BI Step	ML/RL Enhancement	References
$S0_{10}$: Final stage	NNs approximate value functions when closed-form solutions are infeasible	[59]
$S1_{10}$: Recursive backward updates	RL-based forward–backward rollouts and uncertainty-aware bootstrapping improve exploration and sample efficiency	[20]
$S2_{10}$: Base case	Human-like heuristics and imitation learning guide early-stage decisions	[66]
Consistency across stages	Inductive validity, backward compatibility, and rationalizability ensure recursive solutions remain stable	[43, 49, 60]
Applications beyond strategy	BI-inspired mappings enable new AI functionalities (e.g., search, retrieval)	[34]

7 Learning-Enhanced Decision Algorithms

Modern decision-making increasingly combines optimization with learning-based components that adapt to data and uncertainty. While BI and SubModOpt address structured dynamic and set-based problems, respectively, a broad range of alternative methods leverage ML and RL to enhance classical optimization routines. These **learning-enhanced decision algorithms** improve scalability, enable online adaptation, and discover heuristics or policies that approximate optimal decisions without explicit enumeration [40, 53]. In this section, we survey such algorithms, beginning with ML/RL approaches for planning and scheduling, and then extending to broader combinatorial and optimization contexts.

7.1 ML/RL for Planning and Scheduling (Non-BI Approaches)

ML and RL have recently been integrated into classical planning and scheduling algorithms to enhance scalability, generalization, and adaptability. These methods, unlike BI, do not explicitly rely on recursive DynP formulations, but instead embed learned predictive or policy models directly into optimization

frameworks [15, 40]. They combine the efficiency of data-driven inference with the rigor of optimization modeling.

General Framework. A generic learning-enhanced decision problem can be represented as the tuple

$$\mathcal{P}_{\text{learn}} = \langle \mathcal{S}, A, f, c, \pi_\theta, \mathcal{D} \rangle,$$

where $\mathcal{S}$ is the state space, A the set of possible actions, $f : \mathcal{S} \times A \to \mathcal{S}$ the transition function, $c : \mathcal{S} \times A \to \mathbb{R}$ the cost or reward, π_θ a parameterized policy with parameters $\theta \in \mathbb{R}^p$, and $\mathcal{D}$ the dataset or instance distribution. The goal is to learn π_θ minimizing the expected cumulative cost:

$$\min_\theta \ \mathbb{E}_{(s,a)\sim\mathcal{D}}\Big[\sum_{t=0}^{T} c(s_t, \pi_\theta(s_t))\Big], \qquad s_{t+1} = f(s_t, \pi_\theta(s_t)).$$

This learning-based formulation generalizes many classical planning and scheduling models by embedding adaptive, data-driven components [8,61]. Here, T denotes the planning horizon or number of decision steps.

Learning-Enhanced Decision Algorithm (`LEDA`). This algorithm provides a unified structure for combining learned predictive models with optimization solvers. It alternates between three stages: prediction, optimization refinement, and learning. It first predicts a candidate decision using the current learned model in ($S1_{11}$), refines it via exact optimization for feasibility and quality in ($S2_{11}$), and updates the model based on performance feedback in ($S3_{11}$). Through this iterative cycle, `LEDA` learns policies that approximate optimal mappings from instances to decisions, balancing efficiency and accuracy [9, 61]. `LEDA`-style architectures are applied in **scheduling**, **logistics**, and **supply chain optimization**, enabling adaptive policies that generalize across instances and reduce reliance on costly exact optimization [40].

Learning-Based Heuristic Scheduling (`LBHS`). In large-scale scheduling problems, exact optimization methods (e.g., MILP) are often computationally expensive. `LBHS` combines predictive modeling and heuristic dispatching to rapidly construct near-optimal schedules. Let us describe how its steps work. It learns dispatching rules that generalize across scheduling instances. It extracts state features in ($S1_{12}$), prioritizes jobs using a learned policy in ($S2_{12}$), updates the current schedule in ($S3_{12}$), and improves the policy based on performance feedback in ($S4_{12}$). This hybrid learning–optimization approach yields near-optimal schedules in real time and reduces reliance on handcrafted heuristics [15, 61]. `LBHS` is widely used in manufacturing, logistics, and cloud computing for dynamic job dispatching and online scheduling under uncertainty [19].

Algorithm 11 The Learning-Enhanced Decision Algorithm (LEDA)

Initialization ($S0_{11}$) Initialize model parameters θ, dataset $\mathcal{D}$, and optimization problem class $\{X(\xi), f(x, \xi)\}$.

for each training iteration or instance $\xi \sim \mathcal{D}$ **do**

Prediction or policy generation ($S1_{11}$) Compute a candidate solution $x_\pi = \pi_\theta(\xi)$ using the learned model.

Optimization refinement ($S2_{11}$) Refine x_π using an optimization oracle:

$$x^\star(\xi; \theta) = \operatorname*{argmin}_{x \in X(\xi)} f(x, \xi; \theta) \quad \text{initialized at } x_\pi.$$

Learning update ($S3_{11}$) Update parameters θ using gradient-based feedback:

$$\theta \leftarrow \theta - \eta \nabla_\theta F(x^\star(\xi; \theta), \xi),$$

where F denotes a differentiable loss or feedback function measuring the performance of the optimized decision and η is the learning rate.

end for

Output Return the trained decision model $\pi_\theta^\star$.

7.2 Neuro-Symbolic Planning and Logical RL

Beyond planning and scheduling, ML and RL have also been applied to general combinatorial optimization and search. **RL for Combinatorial Optimization (RLCO)** represents one of the most powerful frameworks in this class; e.g., see [15, 53]. RLCO models optimization as a sequential decision-making process where an agent incrementally constructs a solution through learned policies. The RLCO algorithm learns through repeated interactions with a combinatorial environment. It constructs a solution step in ($S1_{13}$), computes total reward based on solution quality in ($S2_{13}$), and updates the policy parameters through gradient feedback in ($S3_{13}$); see [8, 53]. Over time, the learned policy generalizes across instances and can generate high-quality solutions instantly during inference [8, 15]. RLCO has achieved success in solving problems such as the **travelling salesman problem**, **vehicle routing**, **knapsack**, and **bin packing**. These methods demonstrate how RL can discover implicit optimization strategies without explicit programming, providing fast, scalable alternatives to traditional metaheuristics. In this framework, $P(s_t, a_t)$ is the environment transition function that maps a current state–action pair to the next state, and r_t is the immediate reward capturing the incremental improvement in objective value or constraint satisfaction.

Algorithm 12 The Learning-Based Heuristic Scheduling (LBHS) Algorithm

Initialization (S0$_{12}$) Initialize model parameters θ, job set J, machine set M, and scheduling horizon T.

for each decision epoch $t = 0, 1, \ldots, T-1$ **do**

Feature extraction (S1$_{12}$) Compute job features $\phi(j, t)$ (e.g., processing time, remaining work, due-date slack).

Job selection (S2$_{12}$) Rank available jobs via scores $s_j = \pi_\theta(\phi(j, t))$ and assign the top-ranked job to the next machine.

Schedule update (S3$_{12}$) Update machine states and completion status of assigned jobs.

Learning update (S4$_{12}$) Update θ by minimizing scheduling loss:

$$L(\theta) = \mathbb{E}_{\mathcal{D}}[C_{\max}(\pi_\theta) - C^{\star}_{\max}],$$

where $C_{\max}$ denotes the makespan (maximum job completion time) for a schedule and $C^{\star}_{\max}$ represents the optimal or best-known makespan obtained by an exact solver or benchmark heuristic. Moreover, the dataset $\mathcal{D}$ may consist of either simulated problem instances or historical decision data.

end for

Output Return the trained heuristic $\pi^{\star}_{\theta}$.

Algorithm 13 RL for Combinatorial Optimization (RLCO) Algorithm

Initialization ($S0_{13}$) Initialize policy parameters θ and environment representing problem instances (e.g., graphs, jobs, or routes).

for each episode or training instance **do**

Solution construction ($S1_{13}$) Initialize partial solution s_0; for $t = 0, \ldots, T-1$, sample action $a_t \sim \pi_\theta(\cdot|s_t)$ and update $s_{t+1} = P(s_t, a_t)$, where P denotes the environment transition function that maps state-action pairs to successor states.

Reward evaluation ($S2_{13}$) Compute cumulative reward $R = \sum_{t=0}^{T-1} r_t$, where r_t reflects improvement in cost or constraint satisfaction.

Policy gradient update ($S3_{13}$) Update θ using the policy gradient rule:

$$\theta \leftarrow \theta + \eta \, \mathbb{E}_{\pi_\theta}\big[R \nabla_\theta \log \pi_\theta(a_t|s_t)\big].$$

end for

Output Return trained policy $\pi_\theta^\star$ for combinatorial problem generation.

7.3 Comparative Evaluation and Benchmark Insights

Learning-enhanced decision algorithms exhibit complementary strengths compared to classical optimization and planning frameworks. While methods such as `BI` and `SubModOpt` offer theoretical guarantees and interpretability, they often face scalability limits when applied to high-dimensional, data-rich, or uncertain environments. Conversely, ML and RL–based approaches, including `LEDA`, `LBHS`, and `RLCO`, provide empirical efficiency, robust generalization across instances, and the ability to learn problem-specific heuristics directly from experience. This subsection summarizes comparative trends observed in benchmark studies and theoretical evaluations across planning, scheduling, and combinatorial optimization domains.

Scalability and Adaptation. Classical methods rely on explicit enumeration or recursion, which can become computationally prohibitive in large-scale systems. Learning-based algorithms amortize optimization effort by reusing trained models to generate decisions for new instances instantly. For example, `LBHS` achieves near–optimal makespan performance on large job–shop scheduling benchmarks while reducing computation time by an order of magnitude compared to exact MILP solvers [61]. Similarly, `LEDA` and `RLCO` demonstrate strong scalability through neural policy inference, allowing decision quality to remain consistent as instance size or dimensionality grows.

Optimality and Theoretical Guarantees. `BI` and `SubModOpt` retain provable guarantees such as Bellman optimality or diminishing–returns optimality bounds [35, 63]. In contrast, learning-enhanced methods generally trade analytical guarantees for empirical optimality. Recent results show, however, that hybrid formulations—such as `LEDA` with optimization refinement loops—recover partial optimality certificates by constraining learned solutions within feasible sets $X(\xi)$ [40]. These hybrid methods thus combine the rigor of optimization with the adaptability of learning.

Interpretability and Logical Structure. Referee feedback emphasized the importance of preserving logical reasoning within learned models. All algorithms presented here maintain interpretability by defining feasible actions or solutions as logical sets, rather than as relaxed numerical variables. For instance, `RLCO` explores discrete symbolic search spaces (e.g., routes, assignments) and can be viewed as learning a probabilistic policy over logical action sets [15]. Similarly, `LBHS` ensures that precedence and disjunctive constraints remain logically valid even when the policy is learned.

Integration with Classical Frameworks. Learning-enhanced algorithms integrate seamlessly with the earlier frameworks in this manuscript:

- With `BI`, learned value or policy functions provide approximations of recursive reasoning, supporting partially observed or high-dimensional settings.
- With `DisjP`, neural policies can orient disjunctive edges and update feasibility graphs online, improving adaptability to dynamic scheduling.
- With `SubModOpt`, learned gain functions approximate set-wise marginal utilities, enabling fast greedy approximations guided by data.

This unified perspective shows that learning augments rather than replaces classical logic-based optimization.

Empirical Benchmarks. Empirical benchmarks consistently show that hybrid learning–optimization methods achieve near-optimality at significantly lower computational cost. For example, `LEDA` achieves an average optimality gap below 3% on stochastic knapsack and assignment problems [40], while `RLCO` achieves sub-second inference on combinatorial graph benchmarks [53]. Meanwhile, `LBHS` achieves comparable quality to handcrafted dispatching heuristics but adapts dynamically to non-stationary job distributions [19]. These trends suggest that the synergy between learning and optimization yields state-of-the-art performance in dynamic and uncertain environments.

These trends suggest that the synergy between learning and optimization yields state-of-the-art performance in dynamic and uncertain environments. A concise summary of the representative algorithms, their principles, strengths, and limitations is provided in Table 4.

Table 4: Comparison of representative decision-making techniques, highlighting their core principles, strengths, limitations, and references. Only, here, SMO stands for Submodular Optimization and Tech. for technique.

Tech.	Core Principle	Strengths/Limitations/Reference
BI	Backward recursion over dynamic stages	**Strengths:** Optimality guarantee, interpretability **Limitations:** Limited scalability, high recursion cost **Reference:** [7, 56]
LEDA	Learning–optimization refinement loop	**Strengths:** Fast inference, hybrid guarantees **Limitations:** Requires labeled or simulated data **Reference:** [9, 61]
LBHS	Learned dispatching; heuristic scheduling	**Strengths:** Real-time adaptability, reduced solver cost **Limitations:** Limited theoretical guarantees **Reference:** [19]
RLCO	Sequential policy-based construction of combinatorial solutions	**Strengths:** Scalable, domain-agnostic, fully data-driven **Limitations:** Reward design sensitivity **Reference:** [15, 53]
SMO	Greedy set selection under diminishing returns	**Strengths:** Theoretical bounds, compact set structure **Limitations:** Requires explicit submodularity, poor scaling to high-dimmensional **Reference:** [35]

8 Submodular Optimization (SubModOpt)

In this section, we present the foundations of SubModOpt and its applications in discrete and mixed-integer optimization problems. We begin by reviewing the **classical methods** for both submodular minimization and maximization, highlighting the key structural properties of submodular functions—such as the diminishing returns property and their connections to lattice theory—that enable efficient or approximate algorithms. Several representative algorithms are introduced, including the **Lovász extension (LE)**, **cutting plane (CP)**, and **greedy descent (GD)** methods for submodular minimization, as well as the **greedy (Gr)**, **randomized local search (RLS)**, and **multilinear extension (ME)** methods for submodular maximization. These classical algorithms establish the mathematical and computational basis for SubModOpt in combinatorial and continuous settings. Building on this, we later discuss how ML and RL techniques can enhance SubModOpt—through learned relaxations, adaptive policies, and surrogate models—enabling scalable, data-driven, and feedback-based decision-making in complex optimization environments.

8.1 Classical SubModOpt Methods

Let $\mathcal{V}$ be a finite ground set, and let $2^{\mathcal{V}}$ denote its power set (the set of all subsets of $\mathcal{V}$). A set function $f : 2^{\mathcal{V}} \to \mathbb{R}$ is said to be **submodular** when it exhibits a diminishing returns behavior: for any two subsets $A \subseteq B \subseteq \mathcal{V}$ and for any element $x \notin B$,

$$f(A \cup \{x\}) - f(A) \;\geq\; f(B \cup \{x\}) - f(B).$$

Equivalently, f satisfies the lattice inequality

$$f(A) + f(B) \;\geq\; f(A \cup B) + f(A \cap B), \tag{28}$$

for all $A, B \subseteq \mathcal{V}$. When f additionally preserves set inclusion in value,

$$A \subseteq B \;\Rightarrow\; f(A) \leq f(B),$$

we call f a **monotone** submodular function. If such a condition does not hold, f is called **non-monotone**. Optimization with submodular functions is either **submodular minimization** or **submodular maximization**. In submodular minimization, the goal is to find a solution of the problem

$$\min_{\mathcal{S} \subseteq \mathcal{V}} f(\mathcal{S}) \tag{29}$$

in polynomial time by combinatorial algorithms. But, the goal of submodular maximization, which is generally NP-hard, is to find the solution of the problem

$$\max_{\mathcal{S} \subseteq \mathcal{V}} f(\mathcal{S}), \tag{30}$$

which is harder than finding the solution of the submodular minimization problem; hence, approximation algorithms are commonly used to solve submodular maximization problems.

For a lattice $(L, \leq)$, the **join** of $x, y \in L$ is $x \vee y := \sup\{x, y\}$. In addition, the **meet** $x \wedge y$ of x and y is defined as $x \wedge y := \inf\{x, y\}$, i.e., the greatest element in L that is less than or equal to both x and y. A **lattice** is a partially ordered set $(L, \leq)$ whose every two elements $x, y \in L$ have a unique **least upper bound** (join) and a unique **greatest lower bound** (meet). Submodular functions can also be studied using lattice-theoretic approaches. They can be extended to functions $f : \mathcal{L} \to \mathbb{R}$, which satisfies

$$f(x) + f(y) \geq f(x \vee y) + f(x \wedge y).$$

In nonlinear programming (NLP), submodular functions can be used in mixed-integer formulations in the presence of discrete and continuous variables. To generate relaxation techniques for optimization, the Lovász extension [37] can be used to extend submodular functions to continuous domains. To preserve submodular (diminishing returns) by optimization algorithms, submodularity constraints can be encoded linearly. Lattices are tools for Submodular NLP for defining feasible regions and generalizing duality approaches. Submodular minimization is polynomially solvable; Schrijver [58] gave a strongly polynomial algorithm.

8.1.1 Integer Submodular Minimization Algorithm

In this section, we discuss three integer submodular minimization algorithms, Lovász extension (`LE`), cutting plane (`CP`), and greedy descent (`GD`), to find the solutions of the integer submodular minimization problems. For `LE` methods, see [21, 37], for `CP` methods, see [41], and for `GD` methods, see [12, 13].

The `LE` algorithm (=Algorithm 14): The `LE` method reformulates an integer submodular minimization problem as a continuous convex relaxation that can be optimized efficiently. It executes three main stages ($S0_{14}$)–($S2_{14}$) until an optimum point is found. In the initialization step ($S0_{14}$), an initial feasible solution $x = \big(x_1, x_2, \ldots, x_n\big) \in \mathcal{F} := [0, 1]^{\mathcal{V}}$ is selected, where $\mathcal{V} = \big(v_1, v_2, \ldots, v_n\big)$, and its objective value $f(x)$ is evaluated. Then, $x_{\text{best}} = x$ and $f_{\text{best}} = f(x)$ are initialized. During the relaxation stage ($S1_{14}$), the current discrete problem is replaced by the continuous relaxation

$$\min_{x \in P(\mathcal{F})} f_{\text{LE}}(x), \tag{31}$$

where $P(\mathcal{F})$ is the convex hull of incidence vectors of feasible subsets (rather than the simple cube $[0, 1]^{\mathcal{V}}$), and $f_{\text{LE}} : [0, 1]^{\mathcal{V}} \to \mathbb{R}$ denotes the **Lovász exten-**

sion, defined as

$$f_{\mathrm{LE}}(x) = \sum_{i=1}^{n} x_{\pi(i)} \big(f(\mathcal{S}_{\pi(i)}) - f(\mathcal{S}_{\pi(i-1)}) \big).$$

Here, π orders the components of x in descending order ($x_{\pi(1)} \geq \cdots \geq x_{\pi(n)}$), and $\mathcal{S}_{\pi(i)} = \{\pi(1), \ldots, \pi(i)\}$. If f is monotone, then f_{LE} is convex, which makes the relaxed problem (31) amenable to efficient continuous optimization methods. Finally, the updating step ($\mathrm{S2}_{14}$) compares the new objective value $f_{\mathrm{LE}}(x_{\mathrm{relax}})$ with the best value found so far and updates $(x_{\mathrm{best}}, f_{\mathrm{best}})$ accordingly. Then, the integer submodular minimization problem is solved, whose solution is denoted by x_{relax} and its objective function value by $f_{\mathrm{LE}}(x_{\mathrm{relax}})$. Then, the components of such a solution are rounded to integers. In the updating best point step ($\mathrm{S2}_{14}$), if $f_{\mathrm{LE}}(x_{\mathrm{relax}}) < f_{\mathrm{best}}$, then $x_{\mathrm{best}} := x_{\mathrm{relax}}$ and $f_{\mathrm{best}} := f_{\mathrm{LE}}(x_{\mathrm{relax}})$ are updated.

Algorithm 14 LE **Framework**

Initialization: ($\mathrm{S0}_{14}$) Select an initial feasible solution $x_{\mathrm{best}} := x = (x_1, x_2, \ldots, x_n) \in [0,1]^{\mathcal{V}}$; set the initial objective value $f_{\mathrm{best}} := f_{\mathrm{LE}}(x)$.

repeat

Solving relaxation problem: ($\mathrm{S1}_{14}$) Find the solution $(x_{\mathrm{relax}}, f_{\mathrm{LE}}(x_{\mathrm{relax}}))$ of the continuous relaxation of the integer submodular minimization problem (31).

Updating the best point: ($\mathrm{S2}_{14}$) If $f_{\mathrm{LE}}(x_{\mathrm{relax}}) < f_{\mathrm{best}}$, set $x_{\mathrm{best}} := x_{\mathrm{relax}}$ and $f_{\mathrm{best}} := f_{\mathrm{LE}}(x_{\mathrm{relax}})$.

until stopping criterion is met.

Example solved by Algorithm 14: Submodular Minimization via Lovász Extension (LE). Consider the ground set $\mathcal{V} = \{1, 2, 3\}$ and let $E = \{(1, 2), (2, 3)\}$ be the edge set of a simple chain graph. Define the cut function

$$f(S) = \#\{(i, j) \in E \mid i \in S, j \notin S\}, \quad S \subseteq \mathcal{V}.$$

It is known that cut functions are submodular. The goal is to solve the submodular minimization problem (29). Clearly, if $S = \emptyset$ or $S = \mathcal{V}$, then no edges cross the cut, so $f(S) = 0$. Thus, the optimal value is 0.

Lovász extension. We extend f to the continuous domain $x = (x_1, x_2, x_3) \in [0,1]^3$. By minimizing the convex Lovász extension $f_{\mathrm{LE}}(x)$ over the cube $[0,1]^3$, the optimal solutions are $x^* = (0, 0, 0)$ or $x^* = (1, 1, 1)$. After rounding, this gives $S = \emptyset$ or $S = \mathcal{V}$, both achieving $f(S) = 0$. Hence, LE correctly recovers the optimal solution to this minimization problem.

The CP algorithm (=Algorithm 15): The goal of CP is to minimize a submodular function over a feasible region defined by constraints. CP performs three steps ($S0_{15}$)-($S2_{15}$) until an optimum point is found. In the initialization step ($S0_{15}$), an ellipsoid

$$E^{(0)} := \left\{\mathcal{S} \in \mathcal{F} \mid (\mathcal{S} - \mathcal{S}_0)^T H_0^{-1} (\mathcal{S} - \mathcal{S}_0) \leq 1\right\} \tag{32}$$

is formed, including the feasible region $\mathcal{F} := [0,1]^{\mathcal{V}}$ (continuous relaxation of the feasible region), where $\mathcal{S}_0 \subseteq \mathcal{V}$ is the center of the initial ellipsoid, which is the initial feasible point, and H_0 is a positive definite matrix defining its shape (the task of the ellipsoid is to exclude infeasible regions by cutting through the hyperplane). For $k = 0, 1, 2, \ldots$, CP computes the submodular function $f(\mathcal{S}_k)$ ($\mathcal{S}_{\text{best}} = \mathcal{S}_0$ and $f_{\text{best}} = f(\mathcal{S}_0)$ are chosen only for $k = 0$) in ($S1_{15}$), generates the cutting plane

$$f(\mathcal{S}) \geq f(\mathcal{S}_k) + \nabla f(\mathcal{S}_k)^T (\mathcal{S} - \mathcal{S}_k) \tag{33}$$

at $\mathcal{S}_k$ and adds it to the constraints in ($S2_{15}$), where $\nabla f(\mathcal{S}_k)$ is the subgradient of the Lovász extension at $\mathcal{S}_k$, finds the solution $\mathcal{S}_{k+1}$ of the relaxed problem

$$\min_{\mathcal{S} \in E^{(k)}} f(\mathcal{S}) \tag{34}$$

in ($S3_{15}$), updates H_{k+1} by a quasi-Newton formula and the $(k+1)$th ellipsoid

$$E^{(k+1)} := \{\mathcal{S} \in \mathcal{F} \mid (\mathcal{S} - \mathcal{S}_{k+1})^T H_{k+1}^{-1} (\mathcal{S} - \mathcal{S}_{k+1}) \leq 1\} \tag{35}$$

and sets $\mathcal{S}_{\text{best}} = \mathcal{S}_k$ and $f_{\text{best}} = f(\mathcal{S}_k)$ if $f(\mathcal{S}_k) < f_{\text{best}}$ in ($S4_{15}$). This algorithm is terminated if the volume of E^{k+1} becomes sufficiently small or a feasible point satisfying all constraints is found.

Example solved by Algorithm 15. Consider again the ground set $\mathcal{V} = \{1, 2, 3\}$ with cut function

$$f(S) = \#\{(i,j) \in E \mid i \in S, j \notin S\}, \quad E = \{(1,2), (2,3)\}.$$

We want to minimize the submodular problem (29) by Algorithm 15. In ($S0_{15}$), an initial ellipsoid $E^{(0)}$ is formed, covering the feasible region $[0,1]^3$. Choose center $\mathcal{S}_0 = (0.5, 0.5, 0.5)$ and $H_0 = I$. In ($S1_{15}$), $f(\mathcal{S}_0)$ is evaluated. At $\mathcal{S}_0$, f is approximated by a convex relaxation and $f(\mathcal{S}_0) = 1.5$ is supposed. In ($S2_{15}$), linear inequality

$$f(S) \geq f(\mathcal{S}_0) + \nabla f(\mathcal{S}_0)^T (S - \mathcal{S}_0).$$

is constructed. In ($S3_{15}$), the relaxed problems are Minimized subject to ellipsoid and cuts, yielding $\mathcal{S}_1 = (0, 0, 0)$. In ($S4_{15}$), the new ellipsoid $E^{(1)}$ is centered at $\mathcal{S}_1$, shrunk according to violated cut. At $\mathcal{S}_1 = (0, 0, 0)$, $f(\mathcal{S}_1) = 0$ is found, which is optimal.

The GD algorithm (=Algorithm 16): This algorithm is a generic GD algorithm with steps ($S0_{16}$)-($S4_{16}$). In the initialization step ($S0_{16}$), the full

Algorithm 15 CP **Framework**

Initialization: (S0$_{15}$) Given $\mathcal{S}_0 \in [0,1]^{\mathcal{V}}$, form an initial ellipsoid $E^{(0)}$.

for $k = 0, 1, 2, \ldots$ **do**

Computation of submodular function: (S1$_{15}$) Compute the submodular function $f(\mathcal{S}_k)$. If $k = 0$, set $\mathcal{S}_{\text{best}} := \mathcal{S}_0$ and $f_{\text{best}} := f(\mathcal{S}_0)$.

Adding cutting plane: (S2$_{15}$) Form the cutting plane like (33) and add it to the constraints.

Solving relaxed problem: (S3$_{15}$) Find the solution $\mathcal{S}_{k+1}$ of the relaxed problem (34).

Updating parameters: (S4$_{15}$) update H_{k+1} (in practice, quasi-Newton updates may be used, although in the classical ellipsoid method H_{k+1} is updated analytically) and compute $E^{(k+1)}$ by (35). If $f(\mathcal{S}_{k+1}) < f_{\text{best}}$, set $\mathcal{S}_{\text{best}} := \mathcal{S}_{k+1}$ and $f_{\text{best}} := f(\mathcal{S}_{k+1})$.

end for

set $\mathcal{S}_0 := \mathcal{V}$ or any other feasible initial solution is chosen as a binary vector $x_i \in \{0,1\}^n$ with $x_i = 1$ if $i \in \mathcal{S}_0$ and $x_i = 0$ otherwise. Then, $x_0 = x$ and $\mathcal{S}_{\text{best}} = \mathcal{S}_0$ are chosen, and $f_{\text{best}} = f(\mathcal{S}_0)$ is computed. In the computing gradient step (S1$_{16}$), the discrete gradient is computed by either

$$\nabla f(x_k)_i := f(\mathcal{S}_k \cup \{i\}) - f(\mathcal{S}_k) \tag{36}$$

for additions or

$$\nabla f(x_k)_i := f(\mathcal{S}_k \setminus \{i\}) - f(\mathcal{S}_k) \tag{37}$$

for removals. In the computing direction step (S2$_{16}$), the direction

$$d_k \in \operatorname*{argmin}_{d \in D} \nabla f(x_k)^T d \tag{38}$$

is computed to achieve the largest reduction of f. Here, $D \subseteq \{-1, 0, 1\}^n$ is a given feasible direction set. In the evaluation point step (S3$_{16}$), the new binary vector $x_{k+1} := x_k + d_k$, its corresponding set $\mathcal{S}_{k+1}$, and its function value $f(\mathcal{S}_{k+1})$ are computed. The last step (S4$_{16}$) updates the best binary vector or its corresponding feasible set if a reduction of f at the new binary vector is found.

Example solved by Algorithm 16. We consider the ground set $\mathcal{V} = \{1, 2, 3\}$ with the cut function

$$f(S) = \#\{(i,j) \in E \mid i \in S, j \notin S\}, \quad E = \{(1,2), (2,3)\}.$$

Algorithm 16 `GD` **Framework**

Initialization: ($S0_{16}$) Given the feasible direction set $D \subseteq \{-1, 0, 1\}^n$ (which is the set of feasible search directions representing add/remove operations on elements of $\mathcal{V}$), choose the full set $\mathcal{S}_0 \subseteq \mathcal{V}$ as a binary vector $x_i \in \{0, 1\}^n$ with $x_i = 1$ if $i \in \mathcal{S}_0$ and $x_i = 0$ otherwise. Set $x_0 = x$, $\mathcal{S}_{\text{best}} = \mathcal{S}_0$, and compute $f_{\text{best}} = f(\mathcal{S}_0)$.

repeat

Computing discrete gradient: ($S1_{16}$) Compute the discrete gradient $\nabla f(x_k)$ by either (36) or (37).

Computing direction: ($S2_{16}$) Choose $d_k \in D$ by (38).

Evaluating the new point: ($S3_{16}$) Compute x_{k+1}, $\mathcal{S}_{k+1}$, $f(\mathcal{S}_{k+1})$.

Updating the best point: ($S4_{16}$) If $f(\mathcal{S}_{k+1}) < f_{\text{best}}$, set $\mathcal{S}_{\text{best}} := \mathcal{S}_{k+1}$ and $f_{\text{best}} := f(\mathcal{S}_{k+1})$.

until stopping criterion is met.

We want to minimize $f(S)$ over subsets $S \subseteq \mathcal{V}$ by Algorithm 16. In ($S0_{16}$, an initial set $S_0 = \{1, 2, 3\}$ is chosen. Then, $f(S_0) = 0$ (no edges cross the cut) is computed. In ($S1_{16}$), for $i \in S_0$, marginal change is computed if removed as

$$\nabla f(S_0)_i = f(S_0 \setminus \{i\}) - f(S_0);$$

e.g., removing node 2: $f(\{1, 3\}) = 2$ (both edges cut). So $\nabla f(S_0)_2 = 2 - 0 = 2$. In ($S2_{16}$), since removal increases the objective, no descent direction is found. Thus, S_0 is already optimal. The `GD` method terminates with $S_0 = \mathcal{V}$ and $f(S_0) = 0$. This matches the true minimum.

8.1.2 Integer Submodular Maximization Algorithm

In this section, we discuss three algorithms, **greedy** (`Gr`), **randomized local search** (`RLS`), and **multilinear extension** (`ME`) to find the solutions of integer submodular maximization problems. For `Gr` methods, see [13, 46], for `RLS` methods, see [22, 23, 27], and for `ME` methods, see [16].

The `Gr` algorithm (=Algorithm 17): Given the ground set $\mathcal{V} = [n]$, the submodular function $f : 2^{\mathcal{V}} \to \mathbb{R}_+$, and the cardinality constraint k, the goal is to find a subset $\mathcal{S}^* \subseteq \mathcal{V}$ such that $|\mathcal{S}^*| \leq k$ approximately maximizes the function $f(\mathcal{S})$. Algorithm 17 is a generic greedy algorithm. It has three steps ($S0_{17}$)-($S2_{17}$). In the initialization step ($S0_{17}$), an initial subset $S_{\text{best}} = \mathcal{S}_0 \subseteq \mathcal{V}$ is

chosen, and its function value $f_{\text{best}} := f(\mathcal{S}_0)$ is computed. Then, `Gr` alternately performs two other steps (S1_{17}) and (S2_{17}) until a stopping criterion is met. The greedy algorithm `Gr` produces a set $\mathcal{S}_t$ that satisfies

$$f(\mathcal{S}_t) \;\geq\; (1 - 1/e)\, f(\mathcal{S}^\star),$$

implying that the value attained by `Gr` is guaranteed to reach at least a $(1-1/e)$ fraction of the optimal objective $f(\mathcal{S}^\star)$. This guarantee holds for monotone submodular functions under a cardinality constraint k. In the finding maximum gain step (S1_{17}), the marginal gain

$$\Delta f(x, \mathcal{S}_t) := f(\mathcal{S}_t \cup \{x\}) - f(\mathcal{S}_t)$$

is computed to add each element $x \in \mathcal{V} \setminus \mathcal{S}_t$ to the current set $\mathcal{S}_t$. Then, the solution (largest marginal gain) $x_t \in \mathcal{V} \setminus \mathcal{S}_t$ of the optimization problem

$$\max_{x \in \mathcal{V} \setminus \mathcal{S}_t} \Delta f(x, \mathcal{S}_t) \tag{39}$$

is obtained. In the updating information step (S2_{17}), $\mathcal{S}_{t+1} := \mathcal{S}_t \cup \{x_t\}$ is updated, and the best point and its function value are updated if an increment in f at x_t is found. This algorithm is terminated if $\Delta f(x, \mathcal{S}_t) \leq \epsilon$ for all $x \in \mathcal{V} \setminus \mathcal{S}_t$ and for a given threshold $0 < \epsilon < 1$, or a maximum number of iterations is reached.

Algorithm 17 `Gr` **Framework**

Initialization: (S0_{17}) Initialize $\mathcal{S}_{\text{best}} := \mathcal{S}_0 \subseteq \mathcal{V}$, compute $f_{\text{best}} := f(\mathcal{S}_0)$.

repeat

Finding maximum gain: (S1_{17}) Find the largest marginal gain by solving the optimization problem (39).

Updating information: (S2_{17}) Update $\mathcal{S}_{t+1} := \mathcal{S}_t \cup \{x_t\}$ and compute $f(\mathcal{S}_{t+1})$ If $f(\mathcal{S}_{t+1}) > f_{\text{best}}$, set $\mathcal{S}_{\text{best}} := \mathcal{S}_{t+1}$ and $f_{\text{best}} := f(\mathcal{S}_{t+1})$.

until stopping criterion is met.

Example solved by Algorithm 17. Let $\mathcal{V} = \{a, b, c, d\}$ represent four features, and define a coverage function

$$f(S) = \left| \bigcup_{i \in S} U_i \right|, \quad S \subseteq \mathcal{V},$$

where

$$U_a = \{1, 2\}, \quad U_b = \{2, 3\}, \quad U_c = \{3, 4\}, \quad U_d = \{1, 4\}.$$

This is a monotone submodular function (a classic coverage problem). We want to solve the submodular maximization problem

$$\max_{S\subseteq\mathcal{V},|S|\leq 2} f(S)$$

by Algorithm 17. In ($S0_{17}$), the algorithm starts with $S = \emptyset$, $f(S) = 0$. In ($S1_{17}$), the marginal gains $f(\{a\}) = 2$, $f(\{b\}) = 2$, $f(\{c\}) = 2$, $f(\{d\}) = 2$ are obtained. The algorithm chooses a (ties possible). In ($S2_{17}$), with $S = \{a\}$, the algorithm computes the gains

$$\Delta(b|a) = f(\{a,b\}) - f(\{a\}) = 3-2 = 1, \quad \Delta(c|a) = f(\{a,c\}) - f(\{a\}) = 4-2 = 2,$$

$$\Delta(d|a) = f(\{a,d\}) - f(\{a\}) = 3-2 = 1.$$

The algorithm chooses c (ties possible). Greedy yields $S = \{a, c\}$ with $f(S) = 4$. The optimal set is $S^* = \{a, c\}$ (or $\{b, d\}$), with $f(S^*) = 4$. Thus, greedy achieves the $(1 - 1/e)$ approximation guarantee:

$$f(S) = 4 \quad \geq \quad (1 - 1/e) \cdot f(S^*) \approx 0.63 \cdot 4.$$

The `RLS` algorithm (=Algorithm 18): This algorithm is a generic `RLS` algorithm. It has five steps ($S0_{18}$)-($S4_{18}$). The goal of `RLS` is to iteratively refine a solution by performing local searches using randomness in the selection of elements to add or remove from the current solution. In the initialization step ($S0_{18}$), $\mathcal{S}$ is chosen randomly as k elements from $\mathcal{V}$ and its function value $f(\mathcal{S})$ is computed. Here k is the number of cardinality constraints. Moreover, $\mathcal{S}_{\text{best}} := \mathcal{S}$ and $f_{\text{best}} := f(\mathcal{S})$ are initialized. Then, `RLS` alternately performs steps ($S1_{18}$)-($S4_{18}$). In ($S1_{18}$), an element $x \in \mathcal{S}$ is selected randomly and remove from $\mathcal{S}$. In ($S2_{18}$), an element $y \in \mathcal{V} \setminus \mathcal{S}$ is randomly selected and added to $\mathcal{S}$, i.e., $\mathcal{S}' := (\mathcal{S} \setminus \{x\}) \cup \{y\}$ is generated. In ($S3_{18}$), the objective function value $f(\mathcal{S}')$ is computed. In ($S4_{18}$), if $f(\mathcal{S}') > f_{\text{best}}$ holds, then $\mathcal{S}_{\text{best}} := \mathcal{S}'$ and $f_{\text{best}} := f(\mathcal{S}')$ are updated. The algorithm can be terminated if the maximum number of iterations is reached.

Example solved by Algorithm 18. Let $\mathcal{V} = \{a, b, c, d\}$ and define the coverage function

$$f(S) = \left| \bigcup_{i\in S} U_i \right|$$

with

$$U_a = \{1, 2\}, \quad U_b = \{2, 3\}, \quad U_c = \{3, 4\}, \quad U_d = \{1, 4\}.$$

By applying this algorithm, we want to solve

$$\max_{S\subseteq\mathcal{V},|S|\leq 2} f(S).$$

Algorithm 18 RLS **Framework**

Initialization: ($S0_{18}$) Choose randomly $\mathcal{S}$ as k elements from $\mathcal{V}$ and compute $f(\mathcal{S})$. Then, set $\mathcal{S}_{\text{best}} := \mathcal{S}$ and $f_{\text{best}} := f(\mathcal{S})$.

repeat

Updating the set $\mathcal{S}$: ($S1_{18}$) Update randomly $\mathcal{S} := \mathcal{S} \setminus \{x\}$.

Updating the set $\mathcal{S}'$: ($S2_{18}$) Update randomly $\mathcal{S}' := \mathcal{S} \cup \{y\}$.

Computing $f(\mathcal{S}')$: ($S3_{18}$) Compute the function value $f(\mathcal{S}')$.

Updating information: ($S4_{18}$) If $f(\mathcal{S}') > f_{\text{best}}$, then set $\mathcal{S}_{\text{best}} = \mathcal{S}'$ and $f_{\text{best}} = f(\mathcal{S}')$.

until stopping criterion is met.

In ($S0_{18}$), the algorithm chooses random $S_0 = \{a, d\}$ and then $f(S_0) = |U_a \cup U_d| = |\{1, 2, 4\}| = 3$. In ($S1_{18}$), the algorithm removes one element at random: $S' = \{a\}$ with $f(S') = 2$. In ($S2_{18}$), the algorithm adds a random element from $\mathcal{V} \setminus \{a\}$. Suppose we add c, then $S'' = \{a, c\}$ with $f(S'') = |\{1, 2\} \cup \{3, 4\}| = 4$. In ($S3_{18}$), since $f(S'') = 4 > f(S_0) = 3$, update best set to $S_{\text{best}} = \{a, c\}$. RLS can continue iteratively, but here we already hit the optimal $f(S) = 4$.

The ME algorithm (=Algorithm 19): The goal of ME is to maximize $f(\mathcal{S})$ subject to constraints, such as $|\mathcal{S}| \leq k$, where $\mathcal{S} \subseteq \mathcal{V} = [n]$. The continuous multilinear extension $\hat{f} : [0, 1]^n \to \mathbb{R}$ of the submodular function f is defined by

$$\hat{f}(x) = \sum_{\mathcal{S} \subseteq \mathcal{V}} f(\mathcal{S}) \prod_{i \in \mathcal{S}} x_i \prod_{i \notin \mathcal{S}} (1 - x_i).$$

Here $x = (x_1, x_2, \ldots, x_n) \in [0, 1]^n$ is a vector of continuous variables whose components x_i correspond to the fractional membership of an element i in the set $\mathcal{S}$ for $i \in [n]$. Algorithm 19 is a generic ME algorithm. It performs the initialization step by choosing a subset $\mathcal{S}$ of $\mathcal{V}$ and computing its function value $f(\mathcal{S})$. Then, it alternately performs steps ($S1_{19}$)-($S3_{19}$) until an optimum point is found or a maximum number of iterations is reached. In the relaxation problem step ($S1_{19}$), the solution $x^* := (x_1^*, x_2^*, \ldots, x_n^*)$ of the relaxation problem

$$\max_{x \in [0,1]^n,\ \|x\|_1 \leq k} \hat{f}(x) \tag{40}$$

can be found by any standard continuous optimization methods, in which the relaxed version $\hat{f}$ of the submodular function f is maximized, subject to additional constraints such as cardinality or matroid constraints. After solving such a problem, the fractional solution x^* must be converted into a discrete

solution $\mathcal{S} \subseteq \mathcal{V}$. This can be done by the thresholding technique or the randomized rounding technique in the rounding step ($S2_{19}$). In the thresholding technique, for each $i \in [n]$, $x_i = 1$ is chosen if $x_i^* \geq 0.5$ and $x_i = 0$ otherwise. In the randomized rounding technique, for each $i \in [n]$, $x_i = 1$ is chosen with probability x_i^*, while $x_i = 0$ is chosen with probability $1 - x_i^*$. This results in the rounded solution $S = \{i \mid x_i = 1\}$ whose objective function $f(S)$ must be computed. In the updating information step ($S3_{19}$), the best-rounded set $\mathcal{S}_{\text{best}}$ and its function value f_{best} are updated if an increment in f at $\mathcal{S}'$ is found.

Algorithm 19 `ME` **Framework**

Initialization: ($S0_{19}$) Choose randomly $\mathcal{S}$ as k elements from $\mathcal{V}$ and compute $f(\mathcal{S})$. Then, set $\mathcal{S}_{\text{best}} := \mathcal{S}$ and $f_{\text{best}} := f(\mathcal{S})$.

repeat

Solving relaxation problem: ($S1_{19}$) Find the solution x^* of the relaxation problem (40).

Rounding the real solution: ($S2_{19}$) Round x^* to integer x by either thresholding or randomized rounding, resulting in the new rounded solution set $\mathcal{S}$. Then, compute $f(\mathcal{S})$.

Updating information: ($S3_{19}$) If $f(\mathcal{S}) > f_{\text{best}}$, then set $S_{\text{best}} = \mathcal{S}$ and $f_{\text{best}} = f(\mathcal{S})$.

until stopping criterion is met.

Example solved by Algorithm 19. Consider the ground set $\mathcal{V} = \{a, b, c, d\}$ with coverage function

$$f(S) = \left| \bigcup_{i \in S} U_i \right|,$$

where

$$U_a = \{1, 2\}, \quad U_b = \{2, 3\}, \quad U_c = \{3, 4\}, \quad U_d = \{1, 4\}.$$

By applying this algorithm, we want to maximize $f(S)$ subject to $|S| \leq 2$. In ($S0_{19}$), the algorithm relaxes to continuous variables $x = (x_a, x_b, x_c, x_d) \in [0, 1]^4$. In ($S1_{19}$), the algorithm defines

$$\hat{f}(x) = \sum_{S \subseteq \mathcal{V}} f(S) \prod_{i \in S} x_i \prod_{i \notin S} (1 - x_i).$$

In ($S2_{19}$), using continuous optimization, an approximate maximizer is $x^* = (1, 0, 1, 0)$. That is, fully pick a and c, ignore b and d. In ($S3_{19}$), the algorithm chooses $\{a, c\}$. Randomized rounding would also select a with prob 1, c with prob 1. $S = \{a, c\}$ with

$$f(\{a, c\}) = |\{1, 2\} \cup \{3, 4\}| = 4,$$

which is the global optimum point.

8.2 ML/RL-Enhanced for Classical SubModOpt Methods

In this section, we extend the classical integer submodular algorithms by incorporating ML and RL ideas discussed in recent works. The goal is to replace purely combinatorial search by learned or adaptive policies, surrogate functions, and data-driven relaxations.

Lovász Extension with Learning (LE–ML). In the classical LE algorithm, step ($S0_{14}$) initializes with a feasible solution $S_0 \subseteq V$ and step ($S1_{14}$) solves a continuous relaxation

$$\min_{x \in [0,1]^n} \hat{f}(x),$$

where $\hat{f}$ is the Lovász extension of the submodular function f. ML approaches such as deep submodular functions [10] and learning frameworks for submodular functions [3,6] can improve step ($S0_{14}$) by providing learned surrogate objectives $\hat{f}_\theta$ trained from data, and step ($S1_{14}$) by replacing exact evaluations with

$$\min_{x \in [0,1]^n} \hat{f}_\theta(x),$$

thus reducing oracle queries and adapting relaxations to domain-specific structure.

Cutting Plane with Learning (CP–ML). In the CP algorithm, step ($S1_{15}$) computes subgradients $g(S) \in \partial f(S)$ and step ($S2_{15}$) adds cutting planes of the form

$$w^\top x \leq f(S) + g(S)^\top (x - \chi_S),$$

where χ_S is the indicator vector of S. With ML, subgradient prediction models [30, 65] can be integrated into step ($S1_{15}$) by replacing $g(S)$ with a learned predictor $\hat{g}_\theta(S)$, yielding

$$w^\top x \leq f(S) + \hat{g}_\theta(S)^\top (x - \chi_S),$$

thereby accelerating ellipsoid shrinkage and reducing the number of oracle calls.

Greedy Descent with RL (GD–RL). In GD, step ($S1_{16}$) computes discrete gradients

$$\Delta(e \mid S) = f(S \cup \{e\}) - f(S), \qquad e \in V \setminus S,$$

where $f : 2^V \to \mathbb{R}$ is a submodular objective defined on ground set V, $S \subseteq V$ is the current solution, and $\Delta(e \mid S)$ is the marginal gain of adding element e. Step ($S2_{16}$) classically chooses a descent direction by adding or removing the element with the steepest decrease in f.

With RL, step ($S2_{16}$) is replaced by a policy $\pi_\theta(a \mid S)$ that selects an action $a \in \mathcal{A}$, where $\mathcal{A} = \{\text{add}(e), \text{remove}(e) : e \in V\}$, based on the current state S. The reward is defined as

$$r(S) = -f(S),$$

so that minimizing $f(S)$ corresponds to maximizing cumulative reward. The policy parameters $\theta \in \mathbb{R}^d$ are updated using policy-gradient methods:

$$\theta \leftarrow \theta + \eta \, \nabla_\theta \mathbb{E}_{a \sim \pi_\theta(\cdot|S)}[r(S')],$$

where $\eta > 0$ is the learning rate (controlling the magnitude of parameter updates), and S' denotes the new solution obtained after applying action a to state S. This aligns with submodular RL frameworks [51], where the policy improves adaptively through feedback.

Greedy Maximization with ML (`Gr`–`ML`). In the classical `Gr` algorithm, step ($S1_{17}$) selects the element

$$e_t = \operatorname*{argmax}_{e \in V \setminus S_t} \Delta(e \mid S_t), \qquad \Delta(e \mid S_t) = f(S_t \cup \{e\}) - f(S_t),$$

where $f : 2^V \to \mathbb{R}$ is a monotone submodular objective defined on ground set V, $S_t \subseteq V$ is the solution set at iteration t, and $\Delta(e \mid S_t)$ is the marginal gain of adding e.

ML improves step ($S1_{17}$) by introducing a contextual policy $\pi_\theta(e \mid x_t)$ that predicts the next element e_t given instance-specific features $x_t \in \mathbb{R}^d$, with $\theta \in \mathbb{R}^p$ denoting the learned parameters of the policy. Instead of explicitly evaluating $\Delta(e \mid S_t)$ for all $e \in V \setminus S_t$, the algorithm selects

$$e_t \sim \pi_\theta(\cdot \mid x_t),$$

thus reducing oracle queries and enabling generalization to unseen contexts and tasks [54, 65].

Randomized Local Search with RL (`RLS`–`RL`). In `RLS`, step ($S0_{18}$) initializes with a solution $S_0 \subseteq V$, step ($S1_{18}$) randomly removes an element $e \in S_t$ from the current solution S_t, and step ($S2_{18}$) randomly adds an element $e' \in V \setminus S_t$. Formally, the update rule is

$$S_{t+1} = (S_t \setminus \{e\}) \cup \{e'\},$$

where $f : 2^V \to \mathbb{R}$ is the submodular objective and V is the ground set.

RL improves steps ($S1_{18}$)–($S2_{18}$) by replacing uniform random swaps with a policy $\pi_\theta(e, e' \mid S_t)$ that biases removals and additions toward promising candidates, maximizing

$$\max_\theta \ \mathbb{E}_{(e,e')\sim\pi_\theta(\cdot|S_t)}\Big[f\big((S_t \setminus \{e\}) \cup \{e'\}\big)\Big],$$

where $\theta \in \mathbb{R}^p$ are the policy parameters. Separately, meta-learning methods [1] can enhance the initialization ($S0_{18}$) by learning a distribution $q_\phi(S_0)$ over starting solutions parameterized by ϕ, so that the initial set $S_0 \sim q_\phi$ is already close to a high-quality solution. Here, $q_\phi(S_0)$ is a parameterized distribution over initial solutions, with ϕ denoting its learnable parameters.

Multilinear Extension with Learning (`ME`–`ML`). In `ME`, step ($S1_{19}$) solves the multilinear relaxation

$$\max_{x\in[0,1]^n,\ \|x\|_1\le k} F(x),$$

where $F : [0,1]^n \to \mathbb{R}$ is the multilinear extension of a monotone submodular function $f : 2^V \to \mathbb{R}$, $x \in [0,1]^n$ is a fractional solution, and k is the cardinality constraint. Step ($S2_{19}$) then rounds x to an integral solution $S \subseteq V$ using randomized rounding. ML can improve step ($S1_{19}$) by introducing a surrogate $F_\theta(x)$, parameterized by θ, trained using sampled evaluations of $F(x)$ to reduce the computational burden of gradient estimation. Likewise, step ($S2_{19}$) may use a learned rounding policy $\pi_\phi(S \mid x)$, parameterized by ϕ, which infers promising discrete sets S from the fractional representation x. In addition, RL–based adaptive `SubModOpt` [28, 32] updates its policy parameters according to

$$\theta \leftarrow \theta + \eta\, \nabla_\theta\, \mathbb{E}_{S\sim\pi_\phi(\cdot|x)}[f(S)],$$

where $\eta > 0$ is the learning rate. This enhances both steps ($S1_{19}$) and ($S2_{19}$), yielding scalable RL-style optimization with theoretical guarantees.

Table 5 summarizes how classical submodular algorithms can be enhanced with ML and RL. Each row lists the algorithm, its key classical steps, and the corresponding data-driven improvements. ML components provide surrogate relaxations, predictive models, and contextual policies that reduce oracle calls and generalize to new problem instances. RL components introduce adaptive policies that bias search or descent toward promising regions, enabling scalable and feedback-driven optimization while preserving theoretical guarantees.

Alg.	Classical Step(s)	ML/RL Enhancement	Refs.
LE–ML	($S0_{14}$): initialization; ($S1_{14}$): convex relaxation	Surrogate objectives and learned relaxations reduce oracle calls	[3,6,10]
CP–ML	($S1_{15}$): compute subgradients; ($S2_{15}$): add planes	Predictive models estimate informative subgradients	[30, 65]
GD–RL	($S1_{16}$): discrete gradients; ($S2_{16}$): descent direction	RL policies replace descent rules with reward $r(S) = -f(S)$	[51]
Gr–ML	($S1_{17}$): select max marginal element	Contextual policies predict best element, generalize to new instances	[54, 65]
RLS–RL	($S1_{18}$)–($S2_{18}$): random remove or add	RL biases swaps meta-learning improves initialization	[1]
ME–ML	($S1_{19}$): relaxation; ($S2_{19}$): rounding	Surrogate relaxations and learned rounding; policy gradients improve adaptivity	[10, 30], [28]

Table 5: Enhancements of classical submodular algorithms with ML/RL components.

9 Dynamic and Temporal Decision Optimization

Decision-making in real-world systems often unfolds over time, involving sequences of interdependent actions constrained by both logical structure and temporal dynamics. Classical optimization models, while powerful in static settings, struggle to represent evolving system states, uncertainty, and resource interactions that change across stages. **Dynamic and temporal decision optimization** extends these foundations by embedding time, causality, and adaptability into the optimization process. It integrates the logic of disjunctive constraints, which govern mutually exclusive temporal events, with learning-based methods capable of updating policies from experience and observation. This synthesis enables a unified treatment of scheduling, planning, and resource allocation where actions, states, and uncertainties evolve continuously. The following subsections develop this framework—beginning with the classical disjunctive representation of scheduling problems and proceeding toward dynamic,

learning-enhanced, and risk-sensitive formulations.

9.1 Background on Disjunctive Graphs and Dynamic Scheduling

`DisjP` provides a mathematical framework for representing scheduling and sequencing problems in which two operations competing for the same resource cannot overlap in time. Introduced by Balas [5], a disjunctive graph $\mathcal{G} = (V, C \cup D)$ encodes a set of operations V, conjunctive arcs C representing precedence constraints, and disjunctive arcs D representing resource conflicts that must be ordered one way or the other.

Classical Formulation. Each operation $i \in V$ has a start time $x_i \in \mathbb{R}_+$ and processing duration $p_i > 0$. For any two operations (i, j) requiring the same machine, exactly one of the following disjunctions must hold:

$$x_i + p_i \leq x_j \quad \text{or} \quad x_j + p_j \leq x_i,$$

ensuring that no two jobs overlap on a shared resource. Precedence relations $(i, j) \in C$ impose additional constraints $x_i + p_i \leq x_j$. The scheduling objective is often to minimize the makespan

$$C_{\max} = \max_{i \in V}(x_i + p_i),$$

subject to all conjunctive and disjunctive constraints.

This background establishes the classical and dynamic foundations of `DisjS` on which subsequent sections build, including dynamic GNN–RL integration, risk-sensitive RL, and differentiable neural logic for disjunctive reasoning.

Relation to Job-Shop Scheduling. As shown by Balas [5] and further detailed by Pinedo [50], the job-shop scheduling problem can be expressed as the disjunctive program

$$\begin{aligned} \min \quad & C_{\max} \\ \text{s.t.} \quad & x_j - x_i \geq p_i, \quad (i, j) \in C, \\ & (x_i + p_i \leq x_j) \ \vee \ (x_j + p_j \leq x_i), \quad (i, j) \in D, \\ & x_i \geq 0, \quad i \in V. \end{aligned}$$

The conjunctive arcs form precedence chains across jobs, while the disjunctive arcs connect operations competing for the same machine, thereby forming disjunctive cliques within the resource graph [50]. The resulting model yields a compact logical encoding of feasible machine sequences and naturally extends to flexible job shops, parallel machines, and flow-shop systems.

Graph-Based Interpretation. Each feasible orientation of the disjunctive arcs $(i, j) \in D$ induces a directed acyclic graph (DAG) defining one admissible schedule. Dynamic algorithms typically operate by iteratively reorienting disjunctive arcs to adapt to real-time events such as job arrivals, failures, or delays. In this view, the disjunctive graph becomes a dynamic decision graph whose edges evolve.

9.2 Dynamic Scheduling

In dynamic environments, jobs may arrive over time or machine availability may change stochastically. Dynamic scheduling extends the static `DisjP` formulation by allowing the sets of operations and arcs to evolve with time:

$$\mathcal{G}_t = (V_t, C_t \cup D_t), \qquad t = 0, 1, \ldots, T.$$

Each $\mathcal{G}_t$ encodes the scheduling state at time t, and decisions must be updated adaptively as new information arrives.

Reactive and Predictive Scheduling. According to Pinedo [50], dynamic scheduling policies can be categorized as reactive, which respond after disruptions (e.g., machine breakdowns or urgent orders), and predictive, which anticipate changes using forecasts or stochastic models. Reactive rescheduling focuses on minimizing the deviation from the previous plan, while predictive strategies optimize expected performance given probabilistic knowledge of future events.

Mathematical Formulation. Let $x_i(t)$ denote the start time of operation i at time t, and $p_i(t)$ its possibly time-varying processing duration. A dynamic scheduling policy π maps observed states $s_t = (\mathcal{G}_t, \Theta_t)$, where Θ_t captures random factors such as arrivals or breakdowns, to decisions (start times or sequencing actions):

$$a_t = \pi(s_t).$$

The objective is to minimize the expected cumulative cost (e.g., makespan or tardiness):

$$\min_{\pi} \ \mathbb{E}\left[\sum_{t=0}^{T} c(s_t, a_t)\right], \quad s_{t+1} = f_s(s_t, a_t, \Theta_t),$$

where f_s represents the state transition function describing how the system evolves under decision a_t and uncertainty Θ_t. This formulation generalizes the stochastic job-shop models studied by Pinedo [50] and aligns with `DynP` approaches in Russell and Norvig [55].

Adaptive Heuristics and Robustness. Common dynamic strategies include:

- **Rolling-horizon optimization:** periodically resolve a truncated planning horizon to adapt to new data.
- **Priority dispatching:** use heuristic rules (e.g., shortest processing time, earliest due date) that adapt to changing conditions.
- **Stochastic lookahead:** integrate probabilistic forecasts for expected delays or arrivals.

Hybrid reactive–predictive methods have been shown to yield robust performance across industrial job-shop settings [50].

9.3 ML/RL for Temporal and Online Planning

Recent work integrates learning-based prediction and RL with classical `DisjS`, combining adaptive control with graph-based reasoning. These approaches learn policies that dynamically orient or update disjunctive arcs to minimize makespan, tardiness, or energy costs under uncertainty.

GNNs for Scheduling. GNNs provide a structured learning framework for temporal decision graphs [73]. Each node represents an operation with features such as processing time, release date, and machine type, while edges encode precedence or disjunctive constraints. Message-passing updates capture both local and global temporal dependencies, making GNNs suitable for dynamic scheduling and online sequencing [38, 71].

RL for Disjunctive Decisions. RL-based schedulers [42, 71, 72] model the dynamic decision process as an MDP, where each action corresponds to selecting the next operation or resolving a disjunction. The policy $\pi_\theta(a_t|s_t)$ is trained to minimize cumulative makespan or weighted tardiness via reward shaping. Hierarchical RL and graph-based policy representations [38] enable efficient exploration over large combinatorial decision spaces.

Risk-Sensitive and Uncertain Environments. In stochastic systems where processing times or arrivals are uncertain, risk-sensitive RL formulations introduce measures such as `CVaR`-based objectives to manage tail risk in completion time distributions. This builds on Bellman operators extended for risk sensitivity, ensuring that policies remain robust under variability.

Dynamic GNN–RL Integration. Integrating GNNs with RL provides a scalable mechanism for learning adaptive sequencing policies. At each time step t, GNN message passing produces embeddings $h_v^{(t)}$ for each operation, and the RL policy uses these embeddings to select the next disjunctive orientation. The combined update can be written as

$$h_v^{(t+1)} = f_\theta\big(h_v^{(t)}, \{h_u^{(t)}, e_{uv}\}_{u\in\mathcal{N}(v)}\big), \quad a_t \sim \pi_\phi(a_t \mid h^{(t)}),$$

where $\mathcal{N}(v)$ are neighboring nodes in $\mathcal{G}_t$, and ϕ, θ are learned parameters. This dynamic learning loop enables real-time adjustment of sequencing and allocation decisions.

9.4 Dynamic Resource Allocation and Adaptive Decision Graphs

Beyond static scheduling, modern systems involve multiple interacting resources, including machines, workers, and transportation links. **Dynamic resource allocation** generalizes `DisjS` by treating resource availability as a stochastic function of time.

Adaptive Decision Graphs. We define an adaptive decision graph as

$$\mathcal{G}_t = (V_t, C_t \cup D_t, R_t),$$

where R_t represents dynamic resource nodes with time-dependent capacities. Edges between operations and resources encode resource usage constraints, which evolve as jobs start or complete. This representation enables unified modeling of resource allocation and temporal sequencing.

Learning-Based Adaptation. Adaptive decision graphs can be updated online using learned transition models:

$$R_{t+1} = g_\psi(R_t, a_t, \Theta_t),$$

where g_ψ predicts future resource states given current actions and uncertainties. Combining this with a GNN–RL architecture allows for simultaneous optimization of sequencing, resource assignment, and risk-sensitive adaptation in nonstationary environments.

Dynamic and temporal decision optimization merges the logic of `DisjP` with learning-based adaptation. Classical models [5, 50] define the feasible logic and structure, while modern ML/RL approaches [38, 42, 73] enable real-time reasoning over evolving decision graphs. This synthesis supports scalable, interpretable, and adaptive optimization for complex temporal systems.

10 Current Challenges and Future Directions

Building on the synthesis developed throughout Sections 6–9, this section examines the remaining challenges and open research directions defining the frontier of learning–enhanced combinatorial decision optimization. The integration of classical frameworks—such as `SubModOpt`, `BI`, and `DisjS`—with ML and RL has enabled scalable reasoning under uncertainty, yet key theoretical and practical issues remain unresolved.

Goal and Intuition. The objective of this section is to highlight major limitations that currently constrain hybrid ML/RL–optimization frameworks and to outline future research directions that can unify learning and decision theory. Achieving genuine autonomy in decision systems requires not only algorithmic efficiency but also a principled understanding of how learning, reasoning, and optimization interact across temporal and structural scales.

Motivation. By identifying these open challenges, this section defines a research agenda for next-generation hybrid solvers capable of addressing dynamic, stochastic, and large-scale decision problems. The discussion is organized around six core themes: scalability, interpretability, data efficiency, algorithmic integration, benchmarking, and cross-domain adaptation.

Scalability and Structural Generalization. Despite notable progress in adaptive submodular policy optimization [35] and submodular RL [51], learning-based decision algorithms still face scalability bottlenecks when applied to high-dimensional or structurally diverse problem instances. `DynP` and `BI` scale poorly with horizon length, and learned approximations can overfit to narrow distributions of instances. Promising directions include hierarchical policy decomposition, modular graph representations for `DisjS` [38, 73], and meta-learning methods that transfer knowledge across related optimization tasks while retaining submodular structure.

Interpretability and Theoretical Foundations. Classical optimization methods—such as the Lovász extension [37] or Schrijver's polynomial-time minimization algorithm [58]—offer strong theoretical guarantees, whereas learning-augmented variants often sacrifice interpretability or formal correctness. Bridging this gap requires the development of hybrid architectures that preserve submodularity, convexity, and Bellman consistency within neural approximations. Establishing convergence bounds for learned value functions, or providing approximation guarantees for data-driven greedy and dynamic policies, remains an important avenue of research [7, 30, 35].

Data Efficiency and Self-Supervised Learning. Obtaining labeled data or optimal trajectories is computationally expensive in multi-stage and combinatorial problems. Future work should pursue self-supervised and simulation-driven paradigms where intrinsic signals—such as violations of the diminishing-returns property or temporal consistency errors—guide representation learning. Generative modeling of task distributions and curriculum learning across synthetic submodular or scheduling instances could further improve sample efficiency and policy generalization [15, 42].

Integration of Learning with Classical Optimization. Current hybrid algorithms typically couple neural predictors or RL policies with classical optimization as external heuristics rather than unified decision systems. A major challenge is to embed combinatorial operators—such as greedy selection, backward recursion, and disjunctive constraints—within differentiable architectures. End-to-end optimization layers capable of gradient-based training while enforcing feasibility and logical consistency would enable continuous improvement of decision quality without sacrificing structure. Bringing together `BI`, greedy `SubModOpt`, and stochastic RL updates within a unified differentiable framework is a central open goal [14, 40].

Benchmarking and Reproducibility in Decision Optimization. Evaluation of ML/RL–enhanced optimization methods remains fragmented across planning, scheduling, and submodular domains. Benchmark datasets, metrics, and experimental protocols vary widely, hindering reproducibility and fair comparison. Establishing standardized benchmark suites—covering dynamic scheduling, submodular selection, and sequential decision tasks—would enable systematic measurement of optimality gaps, computation cost, and learning efficiency. Such benchmarking would parallel the role of shared datasets in classical operations research, but adapted to learning-driven optimization [42, 50].

Cross-Domain and Temporal Adaptation. Real-world decision problems evolve over time and span multiple interacting domains. Integrating submodular reasoning, dynamic scheduling, and RL in non-stationary environments remains an open challenge. Transferable meta-policies and graph-based temporal representations could allow adaptive coordination of decisions under changing resources, uncertainties, and time scales. Promising applications include logistics, manufacturing, and energy systems, where hybrid optimization must balance efficiency, robustness, and interpretability [35, 38, 50, 51].

Outlook. The convergence of `SubModOpt`, `BI`, and RL represents a paradigm shift toward adaptive, data-driven decision intelligence. Realizing this potential requires scalable graph reasoning, interpretable hybrid architectures, and

rigorous theories linking learning dynamics to optimization principles. Future research should emphasize explainable and verifiable models that retain the mathematical guarantees of classical methods while extending their applicability to uncertain and high-dimensional environments.

Reflection on Research Questions. The challenges above directly relate to the three guiding research questions introduced in Subsection 1.4. **Q1** addressed the integration of ML and RL into structured decision algorithms; progress in differentiable design and hierarchical policies will deepen this integration. **Q2** focused on representational and algorithmic frameworks; graph-based and submodular representations remain crucial for scalability and interpretability. **Q3** concerned open trends and evaluation; continued progress in self-supervised learning, reproducibility, and cross-domain transfer will shape the next generation of learning-augmented optimization systems.

The challenges and opportunities outlined above motivate the synthesis presented in Section 11, where the unification of ML, RL, and classical optimization is discussed as the foundation for interpretable, adaptive, and scalable decision frameworks.

11 Conclusion and Future Work

Combinatorial decision problems represent a central and unifying theme in optimization, encompassing a wide range of discrete and sequential decision-making models, including selection, `DynP`, planning, and scheduling. Classical approaches in operations research have developed powerful mathematical frameworks for these problems, offering provable guarantees of optimality, convergence, and feasibility [5, 37, 58]. However, the combinatorial explosion of decision spaces—especially under temporal, stochastic, and nonlinear dependencies—remains a significant obstacle to scalability and real-time applicability.

This survey has presented a unified formulation of combinatorial decision problems, covering core models such as `SubModOpt`, `DynP`, and temporal scheduling. Building on these foundations, it examined how ML and RL can augment classical solvers through data-driven generalization, neural value approximation, and adaptive policy learning [35, 42, 51]. These learning-augmented frameworks extend traditional optimization by identifying latent problem structure, approximating value landscapes, and learning heuristic search strategies that improve computational efficiency.

The synthesis of ML and RL with combinatorial optimization highlights a new paradigm for large-scale and adaptive decision-making. RL contributes sequential reasoning and experience-driven improvement [20, 62], while ML provides pattern recognition, state representation, and surrogate modeling [3, 15]. Their

integration with combinatorial structure enables interpretable and generalizable solvers that connect symbolic reasoning with statistical learning.

Several open directions remain for future research. First, theoretical foundations for the stability, convergence, and optimality of learned policies in combinatorial settings require further study [30, 35]. Second, incorporating **differentiable programming** and **neuro-symbolic reasoning** offers a pathway toward solvers that can learn both model structure and decision policies jointly [14, 40]. Third, advances in **transfer learning** across problem instances and domains may enable scalable deployment of learned optimization models in logistics, energy management, and autonomous planning [38, 50].

In summary, ML- and RL-assisted solvers for combinatorial decision problems represent a transformative step toward adaptive and intelligent optimization. By combining the rigor of classical algorithms with the adaptability of learning systems, these hybrid frameworks promise to redefine the boundaries of computational decision-making across operations research and artificial intelligence.

12 Supplementary Algorithms and Examples

This appendix collects supplementary algorithms and illustrative examples that complement the formulations and theoretical developments in the main text. They are included for reference and pedagogical completeness but are not required for understanding the core optimization frameworks presented in the paper.

12.1 Algorithms from ML and RL Background

The following algorithms, originally discussed in Section 2, illustrate foundational ML and RL techniques that underpin later decision-optimization frameworks.

- Algorithm 20: **Generic ML Framework** – outlines data-driven model training and inference.
- Algorithm 21: **Actor–Critic (`AC`) Framework** – presents a two-network RL paradigm combining value estimation and policy improvement.
- Algorithm 22: **Proximal Policy Optimization (`PPO`) Framework** – demonstrates a modern policy-gradient method with stability guarantees.

Each of these algorithms is reproduced here in full detail for completeness, as they serve as conceptual foundations for learning-augmented optimization, but are not invoked directly in subsequent sections.

Algorithm 20 Generic Gradient-Based ML Training Procedure

Initialization ($S0_{20}$) Initialize parameters θ_0 randomly, specify loss function ℓ, and learning rate $\eta > 0$.

repeat

Mini-batch gradient computation ($S1_{20}$) Sample a mini-batch $\mathcal{B} \subset \mathcal{D}$ and compute the gradient

$$\nabla_\theta L(\theta) = \frac{1}{|\mathcal{B}|} \sum_{(x,y)\in\mathcal{B}} \nabla_\theta \ell(h_\theta(x), y).$$

Parameter update ($S2_{20}$) Update model parameters by gradient descent:

$$\theta \leftarrow \theta - \eta \, \nabla_\theta L(\theta).$$

until stopping criterion (e.g., convergence or maximum iterations) is met.

Output: Trained parameters θ^*.

Algorithm 21 Actor–Critic (AC) Framework

Initialization ($S0_{21}$) Initialize policy parameters θ, value parameters ϕ, learning rates η, η_v, and discount factor γ.

repeat

Policy interaction ($S1_{21}$) Observe state s_t, sample action $a_t \sim \pi_\theta(\cdot|s_t)$, execute a_t, and observe reward r_t and next state s_{t+1}.

Temporal difference (TD) update ($S2_{21}$) Compute TD error

$$\delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)$$

and update critic parameters $\phi \leftarrow \phi + \eta_v \, \delta_t \, \nabla_\phi V_\phi(s_t)$.

Policy update ($S3_{21}$) Update actor parameters using the policy gradient: $\theta \leftarrow \theta + \eta \, \nabla_\theta \log \pi_\theta(a_t|s_t) \, \delta_t$.

until convergence or maximum iterations.

Output: Trained actor π_θ and critic V_ϕ.

Algorithm 22 Proximal Policy Optimization (PPO)

Initialization (S0$_{22}$) Initialize policy parameters θ, value parameters ϕ, learning rates η, η_v, clip range $\epsilon > 0$, and discount factor γ.

repeat

Trajectory collection (S1$_{22}$) Run the current policy $\pi_{\theta_{\text{old}}}$ to collect trajectories $\tau = \{(s_t, a_t, r_t, s_{t+1})\}$ and compute the advantages

$$\hat{A}_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t).$$

Policy and value updates (S2$_{22}$) Compute the probability ratio

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)},$$

and the clipped surrogate objective $L^{\texttt{CLIP}}(\theta)$ defined by (6). Then, update policy and value parameters, respectively, by

$$\theta \leftarrow \theta + \eta \nabla_\theta L^{\texttt{CLIP}}(\theta), \quad \phi \leftarrow \phi - \eta_v \nabla_\phi (V_\phi(s_t) - R_t)^2,$$

where $R_t = \sum_{l=t}^{T} \gamma^{l-t} r_l$ denotes the empirical return.

Policy synchronization (S3$_{22}$) Set $\theta_{\text{old}} \leftarrow \theta$ for the next iteration.

until convergence or performance threshold met.

Output: Optimized policy parameters θ^* and value parameters ϕ^*.

12.2 Temporal and Metric Planning Examples

This section consolidates extended examples and applications from Section 4, demonstrating how temporal and metric planning models are applied in real-world scheduling contexts.

Example. Consider two actions: task 1 with duration $d_1 = 3$ and task 2 with duration $d_2 = 2$. Task 2 must start only after task 1 completes, so the precedence set is $\mathtt{PR} = \{(1, 2)\}$. Both tasks require one unit of a single shared resource, with capacity $R = 1$. The objective is to minimize the makespan. The optimal plan is to start task 1 at time 0, finish it at 3, and then start task 2 at 3 and finish it at 5, giving $C_{\max} = 5$ (see Figure 3).

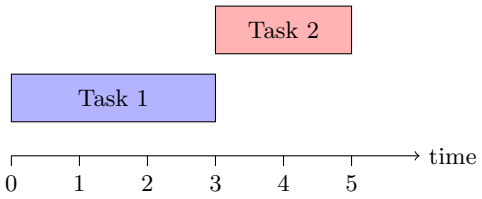


Figure 3: Temporal planning example: Task 1 (duration 3) precedes Task 2 (duration 2). The optimal schedule yields makespan $C_{\max} = 5$.

Applications and Temporal Planning Solvers. Temporal planning has broad applications in **robotics**, **logistics**, **space exploration**, **healthcare**, and the **energy sector**. In robotics, it governs action sequencing with deadlines; in logistics, it optimizes transport operations; in space exploration, it coordinates mission activities; and in healthcare, it schedules treatments and staff. In the energy domain, temporal planning is crucial for operating hydro plants, thermal generators, and renewable resources under time-dependent constraints. These applications typically aim to minimize operational cost, maximize efficiency, or guarantee feasibility under deadlines and resource limits.

Hydropower Scheduling. In hydro scheduling, the goal is to determine when and how much water to release from reservoirs in order to meet electricity demand, while satisfying reservoir capacity limits, turbine capacities, and environmental flow requirements. Let x_t denote water released in period t, s_t the storage level, p_t the power generated, and d_t the demand. A simplified

formulation is

$$\begin{aligned} \min \quad & \textstyle\sum_t c_t\, p_t \\ \text{s.t.} \quad & s_{t+1} = s_t + I_t - x_t, && \text{for all } t, \\ & p_t = \eta\, x_t, && \text{for all } t, \\ & p_t \geq d_t, && \text{for all } t, \\ & 0 \leq s_t \leq S^{\max}, \;\; 0 \leq x_t \leq X^{\max}, \end{aligned}$$

where I_t is the natural inflow, η is the conversion efficiency, c_t is the unit generation cost, $S^{\max}$ is the reservoir capacity, and $X^{\max}$ is the turbine discharge limit. The objective minimizes the total generation cost $\sum_t c_t p_t$, ensuring that electricity demand is met while operating within reservoir and turbine limits (see Figure 4).

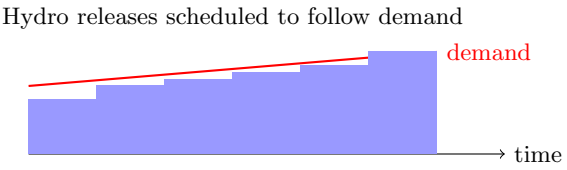


Figure 4: Hydro scheduling: allocating water releases across time periods to match demand.

Unit Commitment in Power Systems. Another important application is unit commitment, where generators must be scheduled on/off over time to satisfy demand. Each generator i has a binary variable $y_{i,t}$ indicating if it is online at time t, with generation limits and start-up costs. A typical MILP is

$$\begin{aligned} \min \quad & \textstyle\sum_{t,i} c_i p_{i,t} + \sum_{t,i} h_i y_{i,t} \\ \text{s.t.} \quad & \textstyle\sum_i p_{i,t} \geq d_t, && \text{for all } t, \\ & P_i^{\min} y_{i,t} \leq p_{i,t} \leq P_i^{\max} y_{i,t}, && \text{for all } i, t, \\ & y_{i,t} \in \{0, 1\}. \end{aligned}$$

Here, c_i is the marginal cost of generation for unit i, $p_{i,t}$ is the power output of generator i, d_t is demand, $P_i^{\min}, P_i^{\max}$ are generation bounds, and h_i is the start-up cost (see Figure 5).

Renewable and Storage Coordination. Temporal planning also governs the integration of renewable energy (e.g., solar, wind) with storage devices. Let g_t denote renewable generation, b_t charging, d_t discharging, and s_t the battery state. Temporal constraints are

$$s_{t+1} = s_t + b_t - d_t, \quad 0 \leq s_t \leq S^{\max}, \quad 0 \leq b_t, d_t \leq B^{\max}.$$

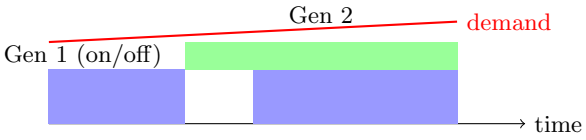


Figure 5: Unit commitment: two generators are scheduled to cover demand over time. The red line represents electricity demand across time periods. Gen 1 (blue) is a base-load unit that cycles on and off to reduce costs, while Gen 2 (green) is a peaking unit that provides additional capacity when demand is higher or Gen 1 is offline. The horizontal axis represents time periods, and the vertical axis represents power output.

Here, $S^{\max}$ denotes the maximum storage capacity, and $B^{\max}$ the maximum charging or discharging rate. The variables b_t and d_t represent charging and discharging amounts, respectively, and s_t is the state of charge at time t. planner ensures demand is met by combining renewable generation, discharging storage, and backup generation, while respecting temporal and resource limits (see Figure 6).

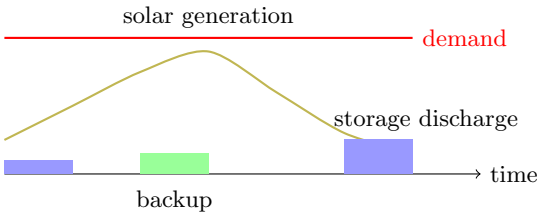


Figure 6: Renewable and storage coordination: electricity demand (red line) must be satisfied across time periods (horizontal axis). Solar generation (yellow curve) varies over the day, and storage (blue bars) discharges during low-solar periods. Backup generation (green bar) is activated when neither solar nor storage is sufficient. The vertical axis represents power output, showing how resources are combined to meet demand.

Example for Optimal Policy in a Stochastic Grid World. Consider a 3×3 stochastic grid world where the agent can move in four directions (up, down, left, right). Each action succeeds with probability 0.8 and moves the agent in the intended direction, while with probability 0.2 it moves randomly to a neighboring cell. The agent receives reward +10 for reaching the goal state and −1 per step otherwise. Figure 7 illustrates the optimal stochastic policy computed using value iteration.

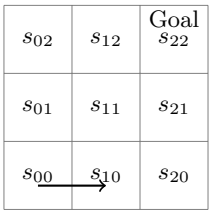


Figure 7: A stochastic grid world: the agent moves stochastically toward the goal (green).

Conditional Planning for Autonomous Flight Example. An autonomous drone switches to a safe flight mode when strong wind is detected, otherwise follows a fast route. This branch structure is depicted in Figure 8.

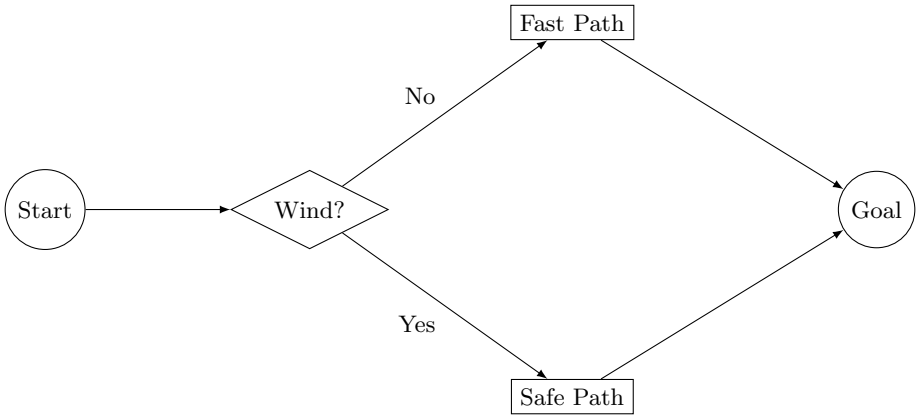


Figure 8: Conditional plan example: switching between safe and fast modes based on event occurrence.

Illustrative Example of Probabilistic INLP. To illustrate the probabilistic INLP formulation (17) and the Monte Carlo Disjunctive Algorithm 4, consider the following toy problem. Let the decision variable $x \in \{0, 1, \ldots, 10\}$ be an integer representing the location of a facility on a one-dimensional line. The uncertain parameter Θ represents a random demand location uniformly distributed over $\{2, 5, 9\}$. The cost function is the distance between x and the demand location:

$$f(x, \Theta) = |x - \Theta|.$$

The goal is to minimize the expected distance subject to integer feasibility:

$$\min_{x \in \{0,1,\ldots,10\}} \mathbb{E}[f(x, \Theta)].$$

True solution. For $x = 5$, the expected cost is

$$\mathbb{E}[f(5, \Theta)] = \tfrac{1}{3}(|5-2| + |5-5| + |5-9|) = \tfrac{1}{3}(3+0+4) = \tfrac{7}{3}.$$

For $x = 2$, $\mathbb{E}[f(2, \Theta)] = \frac{10}{3}$, and for $x = 9$, $\mathbb{E}[f(9, \Theta)] = \frac{11}{3}$. Thus, the optimal decision is $x^* = 5$ with expected cost $7/3$.

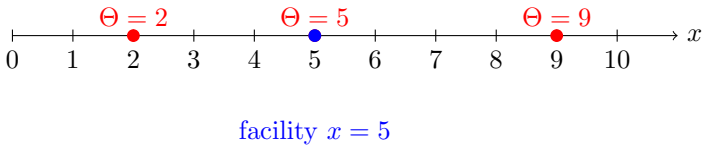


Figure 9: Probabilistic facility location with random demand at $\Theta \in \{2, 5, 9\}$.

12.3 Scheduling Examples

The following examples, drawn from Section 5, illustrate specific scheduling problem types and their optimal solutions.

Example: Two-Machine Scheduling. Consider three jobs with durations $d_1 = 2$, $d_2 = 3$, and $d_3 = 2$, to be scheduled on two identical machines. The precedence relation requires that job J_1 must finish before job J_2 can start, while J_3 is independent. The objective is to minimize makespan. An optimal schedule places J_1 on machine M_1 from time 0 to 2, followed by J_2 on M_1 from time 2 to 5, while J_3 is executed in parallel on machine M_2 from time 0 to 2, giving a makespan of 5 (see Figure 10).

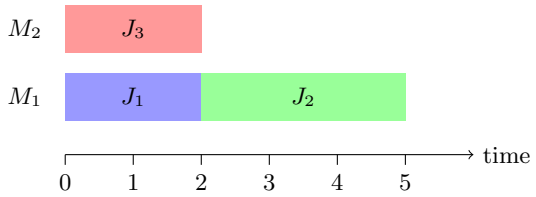


Figure 10: Scheduling example: J_1 precedes J_2 on M_1, while J_3 runs independently on M_2. The horizontal axis represents time, and the vertical axis lists machines. The makespan is 5.

Example: Job-Shop Scheduling. Important examples of scheduling are job-shop scheduling, flow-shop scheduling, project scheduling, batch scheduling, and workforce scheduling. In the job-shop scheduling problem, each job consists

of a sequence of operations that must be processed in a specific order on a set of machines. Each operation requires exactly one machine for a fixed duration, and machines can handle only one operation at a time. The objective is often to minimize the makespan, i.e., the time by which all jobs are completed.

We here formulate the job-shop scheduling problem as

$$\begin{aligned} \min \quad & C_{\max} = \max_{i \in [n]} t_{i,o_i} \\ \text{s.t.} \quad & t_{i,k+1} \geq t_{i,k} + p_{i,k}, && \text{for all } i \in [n] \text{ and } k \in [o_i - 1], \\ & \sum_{i=1}^{n} \sum_{k=1}^{o_i} x_{ijk} \leq 1, && \text{for all } j \in [m] \text{ and } k, \\ & t_{i,k} \geq 0, && \text{for all } i \in [n] \text{ and } k \in [o_i], \end{aligned} \tag{41}$$

where n is the number of jobs, m is the number of machines, o_i is the number of operations for job i, $p_{i,k}$ is the processing time of the kth operation of job i, and t_i is the completion time of job i. Here, t_{i,o_i} denotes the completion time of the last operation of job i, which determines the overall makespan $C_{\max}$. The constraints are: (i) precedence between consecutive operations of each job, (ii) machine capacity (each machine can only process one operation at a time), and (iii) non-negativity of starting times.

Illustrative Job-Shop Scheduling Example. Consider two jobs J_1 and J_2 to be scheduled on two machines M_1 and M_2. Each job consists of two operations:

$$J_1 : O_{11} \to O_{12}, \quad J_2 : O_{21} \to O_{22},$$

where O_{11} must be processed on M_1 for 2 units of time, followed by O_{12} on M_2 for 3 units. Similarly, O_{21} must be processed on M_2 for 2 units, followed by O_{22} on M_1 for 1 unit. The precedence relations are $O_{11} \prec O_{12}$ and $O_{21} \prec O_{22}$. An optimal schedule executes O_{11} on M_1 from 0–2, O_{12} on M_2 from 2–5, O_{21} on M_2 from 0–2, and O_{22} on M_1 from 2–3, yielding a makespan $C_{\max} = 5$ (see Figure 11)

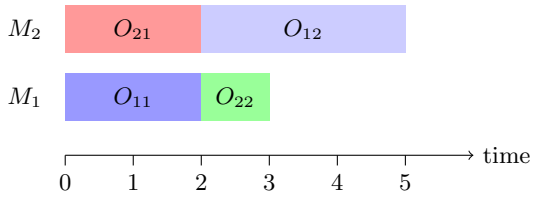


Figure 11: Job-shop scheduling example: Job J_1 requires O_{11} (on M_1) followed by O_{12} (on M_2), while Job J_2 requires O_{21} (on M_2) followed by O_{22} (on M_1). The horizontal axis represents time, and the vertical axis lists machines. The makespan of the schedule is 5.

Example: Workforce Scheduling. Here, we look at the example of workforce scheduling, which allocates workers to tasks with specific time windows and resource constraints. We denote the set of tasks by $T := \{T_1, T_2, \ldots, T_n\}$ and the set of workers by $W := \{W_1, W_2, \ldots, W_m\}$. Let d_i be the duration of task T_i, which must occur between $[t_i^{\text{start}}, t_i^{\text{end}}]$. It is assumed that at most k workers can perform tasks simultaneously. We formulate the workforce scheduling as

$$\begin{array}{lll} \min & C_{\max} & \\ \text{s.t.} & t_i^{\text{start}} \le x_i \le t_i^{\text{end}} & \text{for all } i \in T, \\ & \displaystyle\sum_{j \in W} y_{ij} \le 1 & \text{for all } i \in T, \\ & \displaystyle\sum_{i \in T} f_{\text{active}}(t, i) \le k & \text{for all } t, \\ & C_{\max} \ge x_i + d_i & \text{for all } i \in T, \\ & y_{ij} \in \{0, 1\} & \text{for all } i \in T,\ j \in W. \end{array} \tag{42}$$

Here, x_i denotes the start time of task i, and d_i its fixed duration. The binary variable y_{ij} equals 1 if worker j is assigned to task i, and 0 otherwise. The time-window constraint $t_i^{\text{start}} \le x_i \le t_i^{\text{end}}$ ensures that each task starts within its allowable interval. The second constraint enforces that each task is assigned to at most one worker. The third constraint limits the number of simultaneously active tasks to k, where $f_{\text{active}}(t, i)$ is an indicator equal to 1 if task i is active at time t (i.e., $x_i \le t < x_i + d_i$), and 0 otherwise. The makespan variable $C_{\max}$ represents the latest task completion time.

Example: Flow-Shop Scheduling. Another important scheduling problem is flow-shop scheduling. Unlike job-shop scheduling, where each job has its own routing, in a flow-shop environment, all jobs must follow the same sequence of machines (e.g., production lines). Each job consists of a set of operations that must be performed in the same machine order, and each machine can process only one job at a time. The objective is typically to minimize the makespan. We formulate the flow-shop scheduling problem as

$$\begin{array}{lll} \min & C_{\max} = \max\limits_{i \in [n]} t_i & \\ \text{s.t.} & t_{i,k+1} \ge t_{i,k} + p_{i,k}, & \text{for all } i \in [n],\ k \in [m-1], \\ & t_{i+1,k} \ge t_{i,k} + p_{i,k}, & \text{for all } i \in [n-1],\ k \in [m], \\ & t_{i,k} \ge 0, & \text{for all } i \in [n],\ k \in [m], \end{array} \tag{43}$$

where n is the number of jobs, m the number of machines, $p_{i,k}$ the processing time of job i on machine k, and $t_{i,k}$ the start time of job i on machine k. The first constraint enforces the precedence of operations within each job, the second ensures that machines process jobs sequentially, and the third enforces non-negativity. The formulation assumes a fixed processing order of jobs (e.g., increasing job indices) on each machine. If the job sequence is a decision variable, binary precedence indicators can be introduced to generalize the model.

Illustrative Flow-Shop Scheduling Example. Consider two jobs J_1 and J_2 processed on two machines M_1 and M_2 in the same order. Job J_1 requires 2 units of processing on M_1 followed by 3 units on M_2, while job J_2 requires 1 unit on M_1 followed by 2 units on M_2. A feasible schedule is: J_1 on M_1 from 0–2, then on M_2 from 2–5, and J_2 on M_1 from 2–3, then on M_2 from 5–7. The makespan is 7 (see Figure 12).

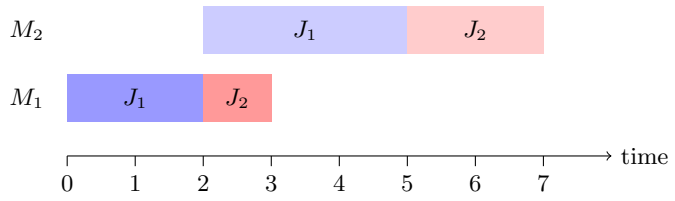
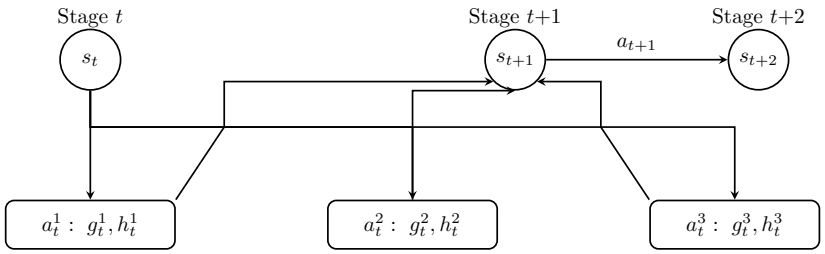


Figure 12: Flow-shop scheduling example: both jobs must follow the same machine order ($M_1 \to M_2$). Job J_1 is processed first on M_1 and then on M_2, followed by J_2. The horizontal axis represents time, and the vertical axis lists machines. The makespan of the schedule is 7.

Figure 13 provides a conceptual visualization of the proposed dynamic `DisjP` framework. It illustrates how disjunctive feasible sets $\mathcal{A}(s_t)$ interact with the temporal recursion of `DynP`, linking spatial disjunctions to sequential decision-making. This addition is purely explanatory and does not alter any quantitative results or algorithms. It is intended to clarify the theoretical formulation introduced in Equations (25)–(26) and Algorithm 9.



Disjunctive feasible actions:
$\mathcal{A}(s_t) = \bigvee_{l=1}^{L} (g_t^l, h_t^l)$

Figure 13: Schematic of the proposed **Recursive Dynamic** `DisjP` framework introduced in this paper. At each stage t, the current state s_t determines a disjunctive feasible action set $\mathcal{A}(s_t) = \bigvee_{l=1}^{L} (g_t^l, h_t^l)$; the selected action a_t drives a transition to the next state s_{t+1}. This diagram conceptually links `DisjP` (spatial feasibility) with `DynP` recursion (temporal decision-making).

12.4 `BI` Example

This section provides an illustrative two-stage decision problem demonstrating the classical `BI` method described in Section 6. It shows how the recursive reasoning process operates in a simple inventory management setting.

Illustrative Inventory Management Example. Consider a two-stage decision problem for inventory management. At stage $t = 1$, the decision is how much to order $x_1 \in \{0, 1, 2\}$. The state s_1 is the initial inventory level. At stage $t = 2$, demand $\Theta_1 \in \{1, 2\}$ is realized with equal probability. The second-stage decision x_2 must satisfy the demand shortfall.

The cost functions are:

$$f_1(s_1, x_1, \Theta_1) = c \cdot x_1, \qquad f_2(s_2, x_2, \Theta_2) = h \cdot (s_2 - x_2)_+ + p \cdot (x_2 - s_2)_+,$$

where c is ordering cost, h is holding cost, and p is penalty cost.

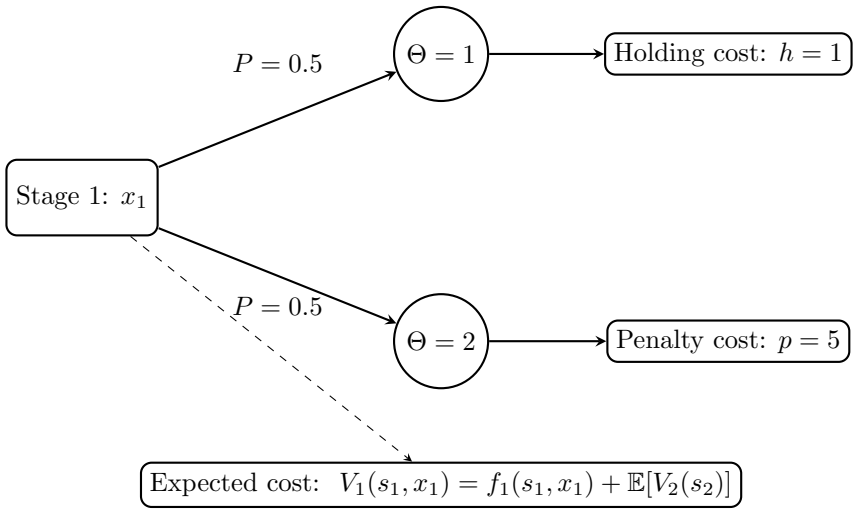


Figure 14: `BI` tree for the two-stage inventory problem. Stage 1 chooses x_1, then demand Θ is realized, Stage 2 incurs holding/penalty cost, and Stage 1 evaluates expected value.

Table 6: Expected costs for different x_1 in the two-stage inventory problem ($c = 1, h = 1, p = 5$).

x_1	Expected total cost $V_1(s_1)$	Decision quality
0	$\frac{1}{2}(5 + 10) = 7.5$	High penalty cost
1	$\frac{1}{2}(1 + 3) = 2$	Optimal
2	$\frac{1}{2}(2 + 3) = 2.5$	Slightly worse

Acknowledgements This work received funding from the National Centre for Energy II (TN02000025).

References

[1] Arman Adibi, Aryan Mokhtari, and Hamed Hassani. Submodular meta-learning. *Advances in Neural Information Processing Systems*, 33:3821–3832, 2020.

[2] James F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23(2):123–154, July 1984.

[3] Francis Bach. *Learning with Submodular Functions: A Convex Optimization Perspective*. Now Publishers Inc., Hanover, MA, USA, 2013.

[4] Chenjia Bai, Lingxiao Wang, Lei Han, Jianye Hao, Animesh Garg, Peng Liu, and Zhaoran Wang. Principled exploration via optimistic bootstrapping and backward induction. In *International Conference on Machine Learning (ICML)*, 2021.

[5] Egon Balas. *Disjunctive Programming*. Springer International Publishing, 2018.

[6] Maria-Florina Balcan and Nicholas JA Harvey. Learning submodular functions. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 793–802, 2011.

[7] Richard Bellman. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503–515, 1954.

[8] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA, USA, 1996. 4th printing, 2021.

[9] Dimitri P Bertsekas and Huizhen Yu. Q-learning and enhanced policy iteration in discounted dynamic programming. *Mathematics of Operations Research*, 37(1):66–94, 2012.

[10] Jeffrey A. Bilmes and Wenruo Bai. Deep submodular functions. *arXiv preprint arXiv:1701.08939*, 2017.

[11] Maxime Bouton, Jesper Karlsson, Alireza Nakhaei, Kikuo Fujimura, Mykel J Kochenderfer, and Jana Tumova. Reinforcement learning with probabilistic guarantees for autonomous driving. *arXiv preprint arXiv:1904.07189*, 2019.

[12] Niv Buchbinder, Moran Feldman, Joseph Naor, and Roy Schwartz. Greedy descent for submodular minimization. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1128–1142. SIAM, 2012.

[13] Niv Buchbinder, Moran Feldman, Joseph Seffi, and Roy Schwartz. A tight linear time (1/2)-approximation for unconstrained submodular maximization. *SIAM Journal on Computing*, 44(5):1384–1402, 2015.

[14] Andreas Bueff and Vaishak Belle. Deep inductive logic programming meets reinforcement learning. In *Proceedings of the 39th International Conference on Logic Programming (ICLP 2023)*, volume 385 of *EPTCS*, pages 339–352, 2023.

[15] Lucian Busoniu, Robert Babuska, Bart De Schutter, and Damien Ernst. *Reinforcement Learning and Dynamic Programming Using Function Approximators*. CRC Press, July 2017.

[16] Chandra Chekuri, Jan Vondrák, and Rico Zenklusen. Submodular function maximization via the multilinear relaxation and contention resolution schemes. In *Proceedings of the forty-third annual ACM symposium on Theory of computing (STOC)*, pages 783–792. ACM, 2011.

[17] Dillon Z Chen, Rostislav Horčík, and Gustav Šír. Deep learning for generalised planning with background knowledge. *arXiv preprint arXiv:2410.07923*, 2024.

[18] Ruiqi Chen, Wenxin Li, and Hongbing Yang. A deep reinforcement learning framework based on an attention mechanism and disjunctive graph embedding for the job-shop scheduling problem. *IEEE Transactions on Industrial Informatics*, 19(2):1322–1331, 2022.

[19] Shweta S Doddalinganavar, PV Tergundi, and Rudragouda S Patil. Survey on deep reinforcement learning protocol in vanet. In *2019 1st International Conference on Advances in Information Technology (ICAIT)*, pages 81–86. IEEE, 2019.

[20] Ashley D. Edwards, Laura Downs, and James C. Davidson. Forward–backward reinforcement learning. *arXiv preprint arXiv:1803.10227*, 2018.

[21] Moran Feldman, Joseph Naor, and Roy Schwartz. A unified continuous greedy algorithm for submodular maximization. In *2011 IEEE 52nd annual symposium on foundations of computer science*, pages 570–579. IEEE, 2011.

[22] Yuval Filmus and Justin Ward. Monotone submodular maximization over a matroid via non-oblivious local search. volume 43, pages 514–542. SIAM, 2014.

[23] Keisuke Fujii and Atsushi Miyauchi. Approximation guarantees of local search algorithms via curvature and gaussian width. In *International Conference on Machine Learning (ICML)*, pages 3316–3325. PMLR, 2020.

[24] Renato Lui Geh, Jonas Gonçalves, Igor C. Silveira, Denis D. Mauá, and Fabio G. Cozman. dpasp: A probabilistic logic programming environment for neurosymbolic learning and reasoning. In *Proceedings of the Twenty-First International Conference on Principles of Knowledge Representation and Reasoning*, KR-2024, page 731–742. International Joint Conferences on Artificial Intelligence Organization, November 2024.

[25] Zoubin Ghahramani. Probabilistic machine learning and artificial intelligence. *Nature*, 521(7553):452–459, 2015.

[26] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

[27] Shayan Oveis Gharan and Jan Vondrák. Submodular maximization by simulated annealing. In *Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms*, pages 1098–1116. SIAM, 2011.

[28] Daniel Golovin and Andreas Krause. Adaptive submodularity: Theory and applications in active learning and stochastic optimization. In *Proceedings of the 23rd Annual Conference on Learning Theory (COLT)*, 2010.

[29] Rostislav Horčik, Gustav Šír, Vítězslav Šimek, and Tomáš Pevnỳ. State encodings for gnn-based lifted planners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 26525–26533, 2025.

[30] Rishabh Iyer. *Submodular Optimization and Machine Learning: Theoretical Results, Unifying and Scalable Algorithms, and Applications*. PhD thesis, University of Washington, 2015.

[31] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1–2):99–134, May 1998.

[32] Branislav Kveton, Anup Rao, Viet Dac Lai, Nikos Vlassis, and David Arbour. Adaptive submodular policy optimization. *Reinforcement Learning Journal*, 6:2720–2736, 2025.

[33] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, UK, 2006.

[34] Donghwan Lee and Wooju Kim. Backward induction-based deep image search. *PLOS ONE*, 19(9):e0310098, September 2024.

[35] Zun Li, Feiran Jia, Aditya Mate, Shahin Jabbari, Mithun Chakraborty, Milind Tambe, and Yevgeniy Vorobeychik. Solving structured hierarchical games using differential backward induction. In *ICLR 2022 Workshop on Gamification and Multiagent Solutions*, 2022.

[36] Chien-Liang Liu and Tzu-Hsuan Huang. Dynamic job-shop scheduling problems using graph neural network and deep reinforcement learning. *IEEE transactions on systems, man, and cybernetics: systems*, 53(11):6836–6848, 2023.

[37] L. Lovász. *Submodular functions and convexity*, page 235–257. Springer Berlin Heidelberg, 1983.

[38] Qiang Ma, Suwen Ge, Danyang He, Darshan Thaker, and Iddo Drori. Combinatorial optimization by graph pointer networks and hierarchical reinforcement learning. *arXiv preprint arXiv:1911.04936*, 2019.

[39] Vikash K. Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. Probabilistic programming with programmable inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018.

[40] Donato Maragno, Holly Wiberg, Dimitris Bertsimas, Ş İlker Birbil, Dick den Hertog, and Adejuyigbe O Fajemisin. Mixed-integer optimization with constraint learning. *Operations Research*, 73(2):1011–1028, 2025.

[41] Hugues Marchand, Alexander Martin, Robert Weismantel, and Laurence Wolsey. Cutting planes in integer and mixed integer programming. *Discrete Applied Mathematics*, 123(1–3):397–446, November 2002.

[42] Nina Mazyavkina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev. Reinforcement learning for combinatorial optimization: A survey. *Computers & amp; Operations Research*, 134:105400, October 2021.

[43] Martin Meier and Andrés Perea. Forward induction in a backward inductive manner. Technical Report 99/24, University of Bath, Bath Economics Research Papers, 2024.

[44] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, NY, 1997.

[45] Max Mowbray, Dongda Zhang, and Ehecatl Antonio Del Rio Chanona. Distributional reinforcement learning for scheduling of chemical production processes. *arXiv preprint arXiv:2203.00636*, 2022.

[46] George L. Nemhauser, Laurence A. Wolsey, and Marshall L. Fisher. An analysis of approximations for maximizing submodular set functions—I. *Mathematical Programming*, 14(1):265–294, 1978.

[47] Liem Ngo. Probabilistic disjunctive logic programming. *arXiv preprint arXiv:1302.3592*, 2013.

[48] Junyoung Park, Jaehyeong Chun, Sang Hun Kim, Youngkook Kim, and Jinkyoo Park. Learning to schedule job-shop problems: representation and policy learning using graph neural network and reinforcement learning. *International Journal of Production Research*, 59(11):3360–3377, 2021.

[49] Judea Pearl. Theoretical impediments to machine learning with seven sparks from the causal revolution. *arXiv preprint arXiv:1801.04016*, 2018.

[50] Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer International Publishing, 2022.

[51] Manish Prajapat, Mojmir Mutny, Melanie Zeilinger, and Andreas Krause. Submodular reinforcement learning. In *Sixteenth European Workshop on Reinforcement Learning*, 2023.

[52] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, New York, 1994.

[53] Benjamin Recht. A tour of reinforcement learning: The view from continuous control. *Annual Review of Control, Robotics, and Autonomous Systems*, 2(1):253–279, May 2019.

[54] Stephane Ross, Jiaji Zhou, Yisong Yue, Debadeepta Dey, and Drew Bagnell. Learning policies for contextual submodular prediction. In *International Conference on Machine Learning*, pages 1364–1372. PMLR, 2013.

[55] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, Harlow, UK, 4th edition, 2021.

[56] John Rust. Numerical dynamic programming in economics. *Handbook of Computational Economics*, 1:619–729, 1996.

[57] Feras Saad and Vikash Mansinghka. A probabilistic programming approach to probabilistic data analysis. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, page 2019–2027, Red Hook, NY, USA, 2016. Curran Associates Inc.

[58] Alexander Schrijver. A combinatorial algorithm minimizing submodular functions in strongly polynomial time. *Journal of Combinatorial Theory, Series B*, 80(2):346–355, 2000.

[59] Leonidas Spiliopoulos. Learning backward induction: a neural network agent approach. In *Agent-Based Approaches in Economic and Social Complex Systems VI: Post-Proceedings of The AESCS International Workshop 2009*, pages 61–73. Springer, 2011.

[60] Megha Srivastava, Besmira Nushi, Ece Kamar, Shital Shah, and Eric Horvitz. An empirical analysis of backward compatibility in machine learning systems. In *Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 3272–3280, 2020.

[61] Changyin Sun, Xiaofeng Li, and Yuewen Sun. A parallel framework of adaptive dynamic programming algorithm with off-policy learning. *IEEE Transactions on Neural Networks and Learning Systems*, 32(8):3578–3587, 2020.

[62] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 2nd edition, 2018.

[63] Hamidreza Tavafoghi, Yi Ouyang, and Demosthenis Teneketzis. A unified approach to dynamic decision problems with asymmetric information—part i: Non-strategic agents. *arXiv preprint arXiv:1812.01130*, 2018.

[64] Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. Deep probabilistic programming. In *International Conference on Learning Representations (ICLR)*, 2017.

[65] Sebastian Tschiatschek, Rishabh Iyer, Haochen Wei, and Jeff Bilmes. Learning mixtures of submodular functions for image collection summarization. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

[66] Kuei Gu Tung, Sheng Wen Wang, Wen Kai Tai, Der Lor Way, and Chin Chen Chang. Toward human-like billiard ai bot based on backward induction and machine learning. In *2019 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 924–931. IEEE, 2019.

[67] Jan-Willem Vandemeent, Brooks Paige, David Tolpin, and Frank Wood. Black-box policy search with probabilistic programs. In *Artificial Intelligence and Statistics*, pages 1195–1204. PMLR, 2016.

[68] Joost Vennekens, Sofie Verbaeten, and Maurice Bruynooghe. Logic programs with annotated disjunctions. In *International Conference on Logic Programming*, pages 431–445. Springer, 2004.

[69] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 28, pages 2692–2700. Curran Associates, Inc., 2015.

[70] Wen-Chi Yang, Giuseppe Marra, Gavin Rens, and Luc De Raedt. Safe reinforcement learning via probabilistic logic shields. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2023.

[71] Wei Zhang and Thomas G Dietterich. A reinforcement learning approach to job-shop scheduling. In *Ijcai*, volume 95, pages 1114–1120, 1995.

[72] Weijian Zheng, Dali Wang, and Fengguang Song. OpenGraphG: A parallel reinforcement learning framework for graph optimization problems. In *International conference on computational science*, pages 439–452. Springer, 2020.

[73] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.