

Senior Thesis in Mathematics

Implementing Network Flows into Reinforcement Learning

Author:
Francisco Xavier Morales Puente

Advisor:
Dr. Konrad Aguilar

Submitted to Pomona College in Partial Fulfillment
of the Degree of Bachelor of Arts

November 18, 2026

Abstract

In this paper, I explore the reinforcement learning pipeline and its mathematical foundations, with a focus on the reward function and how it shapes agent behavior. I then transition to operations research, focusing on network flows, bipartite matchings, and linear programming duality.

Using the connection between dual variables and resource scarcity, I develop a framework for dynamic reward design. In this approach, shadow prices from a linear program are used to penalize resource usage based on how constrained the agent is at each step.

This work highlights a connection between optimization and reinforcement learning, showing how resource constraints can be incorporated directly into the reward function to guide decision making.

Contents

1 Introduction **1**
- 1.1 What is Reinforcement Learning? . . . 1
- 1.2 How does robotics use reinforcement Learning? . . . 3
- 1.3 Limitations . . . 4
- 1.4 Review of the Literature and Relevant Past Results . . . 5

2 Reinforcement Learning Foundations **7**
- 2.1 Markov Decision Processes . . . 7
- 2.2 Value Functions and the Role of Reward . . . 9
- 2.3 Constrained MDPs and Lagrangian POV . . . 10

3 Network Flows into Optimization **12**
- 3.1 Bipartite Matching . . . 12
- 3.2 Network Flows . . . 17
- 3.3 Capacity Constraints and Optimal Allocation . . . 18
- 3.4 From Flow Structure to Linear Optimization . . . 19
- 3.5 Linear Programming and Duality . . . 20

4 Flow-Based Dynamic Reward Design for Reinforcement Learning **25**
- 4.1 Problem Setting and Design Objective . . . 25
- 4.2 Framework Overview and Environment Model . . . 26
  - 4.2.1 Environment . . . 26
- 4.3 Resource Allocation via LPs . . . 28
- 4.4 Dynamic Reward Construction . . . 29

5 Conclusion **31**
- 5.1 Is it worth it? . . . 31

5.2 Next steps 32

Acknowledgments **33**

Technology Statement **34**

Bibliography **35**

Chapter 1

Introduction

1.1 What is Reinforcement Learning?

Reinforcement learning (RL) is a type of machine learning technique in which an *agent* (the learner or decision maker that interacts with an environment) learns to make decisions through trial and error in order to maximize a cumulative reward. In other words, it is a computer learning by doing: improving its performance through repeated interaction with its surroundings.

Reinforcement learning is composed of four main components:

- **State:** The environment or situation an agent is in at any point in time. Agents can move between states by taking different actions.
- **Action:** The set of options available to an agent for interacting with the environment. Actions may correspond to movements, specific operations, or even doing nothing. The agent selects actions based on the current state and the expected reward associated with the next state.
- **Reward:** The feedback an agent receives after each action. Rewards allow the agent to self-discover the most effective paths to achieve its goals. They can be positive or negative, encouraging desirable behavior and discouraging undesirable outcomes.
- **Policy:** The policy refers to the mathematical decision-making process by which an agent uses information about its state, possible actions, and potential rewards to decide what to do next. This policy evolves iteratively as training continues.

There are, of course, many additional elements that contribute to reinforcement learning, but these four form the foundation. Later chapters will explore these components in greater mathematical detail. For now, it is important to gain an intuitive understanding of RL's broader significance and applications.

Reinforcement learning has several benefits:

Example 1.1. RL excels in complex environments. These algorithms can operate effectively in systems with numerous rules, dependencies, and uncertainties. Where a human might become overwhelmed by the amount of information to consider before making a decision, an RL model can quickly adapt to changing conditions, learning through interaction to maximize its performance.

Example 1.2. RL also differs from other learning paradigms. In *supervised learning*, models rely on labeled datasets with predefined inputs and outputs to learn a mapping for prediction. In contrast, *unsupervised learning* uses unlabeled data to uncover hidden patterns. RL occupies a space between the two where it requires less human supervision than supervised learning, since it learns autonomously from feedback rather than explicit labels. Over time, the agent adapts its behavior based on a few basic rules or heuristics.

Example 1.3. A distinctive strength of RL is its focus on long-term optimization. Instead of merely reacting to immediate feedback, RL agents learn from actions that have prolonged or delayed consequences. This forward-looking perspective enables RL to plan sequences of actions that optimize overall success rather than short-term gain.

Reinforcement learning is applied across many domains. In data and content systems, RL enhances personalized recommendations for users on platforms such as Netflix, Spotify, and Amazon. In finance and logistics, RL models develop automated strategies for stock trading or supply-chain management to maximize long-term returns. Most notably, RL plays a crucial role in robotics and autonomous systems, where agents must learn to interact intelligently with their environments under uncertainty.

1.2 How does robotics use reinforcement Learning?

Robotics is an interdisciplinary field of engineering and science that focuses on the design, construction, operation, and application of robots. It is crucial that these systems operate effectively to ensure the successful completion of critical tasks and for the safety of the individuals interacting with them daily.

Reinforcement learning is one of the paradigms used to enable robots to operate autonomously and achieve their intended objectives. Due to its ability to learn through interaction and adapt to complex environments, RL allows robots to refine their behavior over time without explicit human supervision.

Reinforcement learning is applied across many types of robots and a wide range of tasks. Consider the following past results of different frameworks utilizing reinforcement learning in robotics:

Example 1.4 (Autonomous Navigation and Pathfinding)**.** Classical pathfinding algorithms such as Dijkstra's or A* perform well in static and well-structured environments but struggle in dynamic or unpredictable settings. However, RL-based navigation, enables robots to learn adaptive strategies through interaction with their surroundings, without requiring full prior knowledge of the environment. This approach is seen most often in autonomous vehicles and mobile robots.

Example 1.5 (Adaptive Control)**.** Traditional control systems often rely on a fixed model of the robot and the environment. If conditions unexpectedly change, it can lead to failure or poor performance. RL allows for a robot to learn and refine its control policy through continuous interaction and a feedback loop, making it that much more adaptive. This is most prevalent in surgical robots, which strive for precision and control to ensure optimal safety and success for patients.

Example 1.6 (Human-Robot Interactions)**.** Previous robots that interacted with humans involved rigid, manually coded rules that produced inflexible and unnatural robotic behavior. The RL approach allows robots to learn optimal social behaviors through continuous trial-and-error and observation of human feedback.

Example 1.7 (Complex Decision Making and Optimization)**.** Lastly, and most importantly to my thesis. RL excels at complex decision making and

optimization by allowing agents to discover high-level strategies and sequential actions in environments with numerous variables, constraints, and long-term goals. For example, in test where a robot has to accomplish many different tasks and optimize its value to some environment, RL would allow for a robot to attempt different actions approaches to them, and combination to improve after each and every iteration.

However, RL is not the perfect system. There are inefficiencies and challenges surrounding it which I'd like to talk about.

1.3 Limitations

Despite its versatility, reinforcement learning (RL) faces several practical and theoretical limitations that motivate the exploration of more structured frameworks. Aside from the most prominent challenges, such as the extensive time and data required for training since RL agents often need millions of interactions with their environment, and the high computational cost of these processes, several additional limitations should be considered:

Reward design challenges. Crafting a single scalar reward function that appropriately balances multiple objectives and encodes complex constraints is notoriously difficult. Poorly engineered rewards often lead to *reward hacking*, where an agent exploits shortcuts that maximize immediate reward but fails to achieve the true goal.

Sparse and delayed rewards. In many real world tasks, meaningful rewards occur only after long sequences of actions, making it difficult for the agent to determine which actions contributed to success, which ends up slowing down learning.

Training instability. Learning can be highly unstable and sensitive to hyperparameter tuning, often leading to inconsistent or irreproducible results.

Interpretability. Even after training, the decision process of an RL agent is often compared to a "black-box", making it difficult for humans to understand why particular actions are chosen.

These challenges motivate exploring more structured mathematical approaches for altering the RL paradigm. Integrating network flow formulations like *multi-commodity flows* into the RL framework may address several of these limitations. A flow-based structure could represent actions and constraints as graph relationships, assigning different weights to objectives such

as cost, time, or safety. By following a consistent flow network, agents could make decisions that are both more interpretable and more stable, while naturally incorporating constraint satisfaction into the learning process.

1.4 Review of the Literature and Relevant Past Results

The ideas explored in this thesis build upon several key areas of past research: the math behind reinforcement learning, structured and sparse-reward learning, combinatorial optimization through network flows, and results from graph theory that motivate studying structure in large systems.

- **Mathematical foundations of reinforcement learning.** Zhao's *Mathematical Foundations of Reinforcement Learning* [7] formalizes the framework of Markov Decision Processes (MDPs), defining value and policy functions and proving convergence of algorithms such as value and policy iteration through contraction mappings. These results guarantee the existence of optimal stationary policies and establish RL as a well-defined optimization problem over sequential decisions. This provides the basis for the work proposed in this thesis to be built on.

- **Structured and sparse-reward learning.** Reddy, Dragan, and Levine's SQIL framework [3] shows that reinforcement learning can function effectively with minimal feedback by embedding structure directly into the reward function. Instead of imitating expert actions, SQIL aligns an agent's behavior with expert state–action distributions. This demonstrates how adding structure to rewards can shape learning outcomes—an idea conceptually similar to introducing network flow constraints within the RL process.

- **Combinatorics and network flows.** Shahriari's text [4] and lecture notes on *Linear Programming, Duality, and Network Flows* alongside Taha's *Operations Research: An Introduction* [6] provides the Operations Research context for formulating and solving network flow problems. These works outline how bipartite matchings can be expressed as flow models and solved efficiently, offering the background needed to apply such formulations within reinforcement learning for constraint-aware decision making.

- **Structure in large systems.** The canonical Ramsey theorem with list constraints, proven by Alvarado et al. [1], shows that structured patterns inevitably arise in large combinatorial systems. Though not directly related to RL, this result motivates viewing structure as an inherent property of complex and constrained environments, supporting the exploration of matching and flow-based representations.

Structured rewards can meaningfully shape agent behavior, while network flows and matchings provide a compact mathematical language for modeling constrained allocation. Building on these ideas, my thesis asks how the structure and guarantees of network flows can be embedded within reinforcement learning to achieve constraint-aware, efficient multi-task behavior.

Chapter 2

Reinforcement Learning Foundations

We now formalize the reinforcement learning framework and the aspects of the reward function that motivate this work. The definitions and ideas in this chapter are based on standard references in reinforcement learning [7, 2].

2.1 Markov Decision Processes

Reinforcement learning models how an agent interacts with a stochastic environment by taking actions, transitioning between states, and receiving feedback in the form of rewards.

Definition 2.1 (Markov Decision Process (MDP))**.** The standard mathematical framework for describing this interaction is called *Markov Decision Process* (MDP), and it consists of the following main components:

- **State Space,** denoted $\mathcal{S}$: the set of all states.
- **Action Space,** denoted $\mathcal{A}(s)$: a set of actions that are associated with each state $s \in \mathcal{S}$.
- **Reward Set,** denoted $\mathcal{R}(s, a)$: a set of rewards associated with each state-action pair (s, a).
- **Transiton Model,** denoted $p(s' \mid s, a)$: is the state transition probability, defined by

$$p(s' \mid s, a) = \Pr(S_{t+1} = s' \mid S_t = s, A_t = a),$$

describing the probability of transitioning to state s' after taking action a in state s.

- **Reward Model,** denoted as $p(r \mid s, a)$: Which describes the probability of obtaining reward, r, when action a is taken in state s ($r(s, a)$).
- **Policy,** denoted as $\pi(a \mid s)$: The probability of choosing action a in state s. Note that the sum of all probabilities from all possible actions for a given state will always equal 1.

Remark 2.2 (Markov Property). The defining assumption of an MDP is the *Markov property*, which states that the next state depends only on the current state and action, not the full history:

$$\Pr(S_{t+1} \mid S_1, A_1, \ldots, S_t, A_t) = \Pr(S_{t+1} \mid S_t, A_t).$$

It is imperative to understand that although the Markov property makes it such that the next state depends only on the current state, we still might want a way to represent a history of states and actions that were taken over the course of some time by a given policy π.

Definition 2.3 (Trajectory)**.** A **trajectory** is a sequence generated by following a policy π in the MDP:

$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots)$$

Moreover, the concept of return summarizes the cumulative reward collected along a trajectory.

Definition 2.4 (Return)**.** The **return** G_t is the cumulative discounted reward from time step t onward:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1},$$

where $0 \leq \gamma \leq 1$ is the **discount factor** that balances immediate vs. future rewards.

Given an MDP, the overall goal of reinforcement learning is to find a policy that maximizes the expected discounted return.

Definition 2.5 (Optimization Objective)**.** The reinforcement learning problem can be written as

$$\max_{\pi} \ \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right].$$

Remark 2.6. Here, the summation

$$\sum_{t=0}^{\infty} \gamma^t r_t$$

represents the total discounted reward along a trajectory, where γ^t reduces the influence of rewards received further in the future.

Next, the expectation $\mathbb{E}_{\pi}[x]$ appears because the total reward obtained under a policy may vary from one execution to another. We therefore optimize the average long-term reward achieved by the policy.

In other words, reinforcement learning seeks a rule for choosing actions so that, on average, the agent accumulates as much long-term reward as possible.

Most importantly, the reward function r determines what "as much as possible" actually means. Thus, changing the reward function changes the objective being optimized and can alter the resulting optimal policy something that we will explore in Chapter 4.

2.2 Value Functions and the Role of Reward

Definition 2.7 (Value Function)**.** At each time step t, the agent is in a state S_t, takes an action, and receives a reward r_t. Let G_t denote the total future reward starting from time t. The **value function** under a policy π is the expected return starting from state s.

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[G_t \mid S_t = s\right],$$

Here, $S_t = s$ means the agent starts in state s at time t, and the expectation averages over all possible future trajectories under policy π.

Definition 2.8 (Optimal Policy)**.** Let G_0 denote the total future reward starting from time 0. An **optimal policy** π^* maximizes the expected return:

$$\pi^* = \arg\max_{\pi} \ \mathbb{E}_{\pi}\left[G_0\right].$$

Remark 2.9. The value of a policy depends directly on the reward function. Changing the reward changes the return, which changes the value function and therefore the optimal policy.

This means that modifying the reward function directly changes the behavior of the agent.

At each time step, the agent receives a scalar reward $r(s, a)$. The value function aggregates these rewards over time, and the agent selects actions to maximize:

$$\max_{\pi} \mathbb{E}_{\pi} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right].$$

It's important to note that the agent does not observe the true goal directly, but only the reward signal, which defines its behavior. As a result, poorly designed rewards can lead to unintended behavior, like *reward hacking*, where the agent exploits the reward structure instead of solving the intended task.

Thus, modifying the reward function changes the return, which alters the value function and ultimately the optimal policy.

2.3 Constrained MDPs and Lagrangian POV

In many problems, the agent must satisfy constraints in addition to maximizing its reward. For example, an agent could be limited by time, energy, or safety requirements.

We can write a constrained reinforcement learning problem can be written as

$$\max_{\pi} \mathbb{E}_{\pi} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right] \quad \text{subject to resource constraints.}$$

One way to handle constraints is through a Lagrangian formulation, where constraints are incorporated into the objective using additional variables.

Concretely, a constrained problem of the form

$$\max_{\pi} \mathbb{E}_{\pi} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right] \quad \text{subject to } c_i(\pi) \leq d_i$$

can be rewritten as the unconstrained objective

$$\max_{\pi} \; \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right] \; - \; \sum_i \lambda_i \big(c_i(\pi) - d_i\big),$$

where $c_i(\pi)$ is the expected usage of resource i under π, d_i is its limit, and $\lambda_i \geq 0$ is a weight called a *Lagrange multiplier*.

The term $c_i(\pi) - d_i$ measures how much constraint i is violated. When the agent stays within its limit this term is non-positive and contributes nothing; when the limit is exceeded it subtracts from the objective, penalizing the policy. The multiplier λ_i controls how severely: a larger value pushes the policy harder away from the constraint boundary.

Constraints are therefore no longer enforced as hard barriers. Instead they are absorbed into the reward, and the agent trades off performance against resource usage through the multipliers.

Remark 2.10. The multipliers λ_i assign a value to each constraint. A large λ_i means the resource is tight, while a value near zero means it is not limiting.

In the next chapter, we will further formalize this idea with linear programming and duality.

Chapter 3

Network Flows into Optimization

We now turn to the mathematical tools that support this work, including network flows, linear programming, and the necessary graph theory.

The definitions and remarks in this chapter are primarily based on Shahriar Shahriari's text on linear programming [5].

3.1 Bipartite Matching

Example 3.1. Your friend Zayra is planning on opening a restaurant, and she has already done all of her research for the logistics behind it; however, she is currently struggling in hiring skilled people for "Zayra's Tasty 'Za". She currently has 3 different openings and 4 applicants with varying skills. She wants to maximize the number of qualified people she hires.

To analyze problems like Zayra's restaurant hiring scenario, we need a mathematical structure that captures relationships between two distinct sets. In this case, applicants and job openings. Graph theory provides such a framework. By representing individuals and positions as vertices, and connecting them with edges whenever a pairing is possible, we can formalize allocation and pairing problems. This abstraction allows us to reason about the existence of feasible pairings, optimal assignments, and the properties of these relationships using tools from operations research and combinatorics.

To formalize this kind of representation, we begin by recalling some fundamental graph concepts.

Definition 3.2 (Graph and Subgraph)**.** A graph G is a pair of sets (V, E) where V is a non-empty set of vertices of G, and E is a (possibly empty) set of unordered pairs of distinct elements of V called the set of edges.

If $G = (V, E)$ and $G' = (V', E')$ are both graphs, then G' is a subgraph of G if $V' \subseteq V$ and $E' \subseteq E$.

Example 3.3. Consider the graph $G = (V, E)$ where $V = \{1, 2, 3, 4\}$ and $E = \{(1, 2), (2, 3), (3, 4), (3, 1), (4, 1)\}$.

Also, consider the subgraph $G' = (V', E')$ where $V = \{1, 2, 3\}$ and $E = \{(1, 2), (2, 3), (3, 1)\}$.

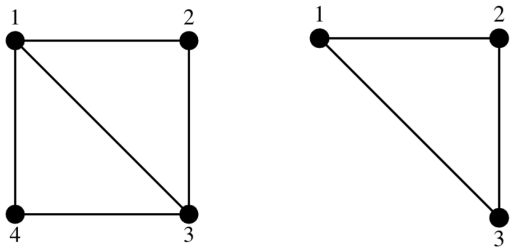


Figure 3.1: Graph G shown on left and Subgraph G' shown on right

So far, we have assumed that each pair of vertices can be connected by at most one edge. However, in many real-world settings, it is natural to allow multiple connections between the same two entities.

Definition 3.4 (Multigraph)**.** In terms of a graph, if we allow E to be a multiset (with repeated edges) then E is called a multigraph or a graph with repeated edges. If we also allow pairs of non distinct elements in E (allow loops) then G is called a general graph.

Consider the multigraph below:

Once we have vertices and edges, we can describe how they relate to one another through basic notions such as adjacency, incidence, and order.

Definition 3.5 (Order, Adjacencency, Incidence)**.** Let G = (V,E) be a general graph. Then $|V|$, the number of vertices is called the order of G

If $\alpha = \{x, y\} \in E$ (there is an edge connecting the vertices x and y) then we say that x and y are vertices or ends of α, α joins x and y, that x and y are adjacent, and that x and α are incident.

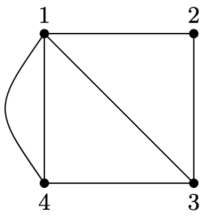


Figure 3.2: Multigraph

Understanding adjacency leads to the idea of counting how many edges meet at a given vertex.

Definition 3.6 (Degree)**.** If x is a vertex of a graph G, then the number of edges incident with x is called the degree of x. In the case of general graphs, a loop at vertex x contributes 2 to the edges of x.

We can actually connect these vertices together with a certain order with the concept of paths.

Definition 3.7 (Paths)**.** Let n be a positive integer and let $V = [n] = \{1, 2, ..., n\}$ and let $E = \{\{1, 2\}, \{2, 3\}, ..., \{n-1, n\}\}$ be the set of consecutive pairs of elements of $[n]$. A path of length $n - 1$, denoted often by P_{n-1} has order n and $n - 1$ edges. Note that the subscript in P_k refers to the length of the path.

Example 3.8. Consider the graph $G = (V, E)$ where $V = \{1, 2, 3, 4\}$ and $E = \{(1, 2), (2, 3), (3, 4), (3, 1), (4, 1)\}$. A path in this graph is shown highlighted in red below: $P_4 = \{(1, 2), (2, 3), (3, 4), (4, 1)\}$

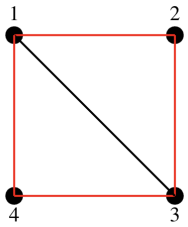


Figure 3.3: Path

Having established these foundational ideas, we can now introduce a special type of graph that models pairings between two distinct sets.

Definition 3.9 (Bipartite Graph)**.** Let $G = (V, E)$ be a simple graph (or a multigraph). Then G is a bipartite graph (or multigraph) if we can partition the set of vertices into two sets X and Y (i.e., $V = X \cup Y$ and $X \cap Y = \emptyset$) such that all the edges of G have one end in X and one end in Y.

If G is a bipartite graph, then often we write $G = (X, E, Y)$, where $X \cup Y$ is the set of vertices, E is the set of edges, and all the edges have one end in X and one end in Y. The sets X and Y are called the parts of G.

We can now return to our motivating example.

Let X be the set of applicants applying for Zayra' restaurant. Next, let Y be the set of positions that are open. Now let each edge from the set X going into a vertex from the set Y represent whether a certain applicant is skilled to do that job. The result is a bipartite graph representation of Zayra's hiring problem, shown in Figure 3.4.

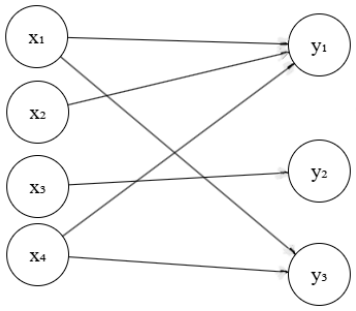


Figure 3.4: Bipartite

Okay so we have a graph representation to model our problem; however, what exactly is a matching problem.

Definition 3.10 (Matching)**.** Let $G = (V, E)$ be an undirected graph. Let $M \subseteq E$. Then M is a matching, if for every $v \in V$, $\exists$ at most one edge in M incident with v. A maximum matching is a matching in G of maximum possible size.

Finding a matching, or more specifically, a maximum matching, provides a direct solution to Zayra's problem: it identifies the largest possible set of qualified applicant–position pairs with no conflicts.

In later sections, we will see how this process can be reformulated and efficiently solved using techniques from *network flow theory*.

3.2 Network Flows

Definition 3.11 (Digraph)**.** A digraph G is a pair (V, E) where V is a nonempty set of vertices and E is a (possibly empty) set of arcs denoted as ordered pairs of distinct vertices. We can designate a vertex s (source) and a vertex t (sink). A digraph with both of these is sometimes called a flow network.

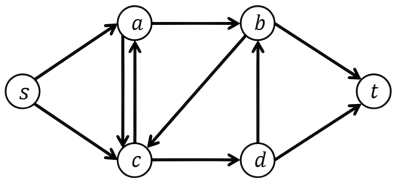


Figure 3.5: Digraph

Definition 3.12 (Flow Network with Arc Capacities)**.** A *flow network* is a directed graph $G = (V, E)$ with two distinguished vertices: a *source* $s \in V$ and a *sink* $t \in V$. Each arc $(x, y) \in E$ has an associated *capacity* given by a function

$$c : E \to \mathbb{R}_{\geq 0} \cup \{\infty\},$$

where $c(x, y)$ represents the maximum amount of flow that can pass from x to y.

Consider the network flow below:

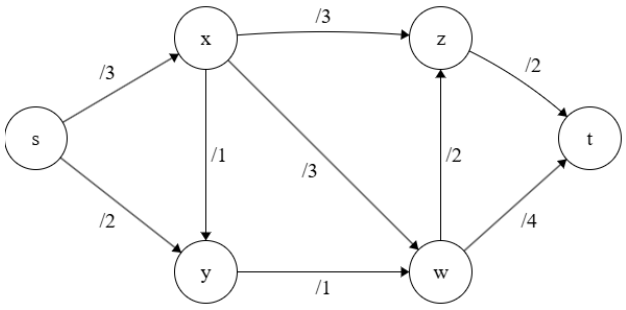


Figure 3.6: Network Flow

Definition 3.13 (Flow and Feasibility)**.** A *flow* in a flow network $G = (V, E)$ with source s and sink t is a function $f : E \to \mathbb{R}_{\geq 0}$ satisfying the following conditions:

1. **Flow conservation:** For every vertex $x \in V - \{s, t\}$,

$$f(x, V) = f(V, x),$$

   meaning that the total flow into x equals the total flow out of x.

2. **Capacity constraint:** For every arc $(x, y) \in E$,

$$0 \leq f(x, y) \leq c(x, y).$$

   meaning the flow through an arc is non-negative and at most the capacity.

A flow that satisfies both conditions is said to be *feasible*.

Remark 3.14 (Value of a Flow). The *value* of a flow f, denoted $|f|$, is defined as

$$|f| = f(s, V) - f(V, s),$$

which represents the total amount of flow leaving the source (equivalently, entering the sink). A flow is called a *maximum flow* if there is no feasible flow with a larger value.

3.3 Capacity Constraints and Optimal Allocation

Having defined flow networks and feasibility, we now consider what it means to *solve* a network flow problem.

Given a flow network $G = (V, E)$ with source s and sink t, the maximum flow problem asks: among all feasible flows, which one sends the greatest total amount of flow from s to t?

Formally, we seek a feasible flow f maximizing its value

$$|f| = f(s, V) - f(V, s).$$

Algorithms such as Ford–Fulkerson compute a maximum flow by repeatedly identifying augmenting paths along which additional flow can be pushed.

The procedure terminates when no such path exists, at which point the flow is optimal.

Additionally, we could also utilize the max-flow min-cut theorem. However, for our purposes, the computational details are secondary and not the focus of this section. Instead, what is essential is the structural interpretation:

- A feasible flow respects all capacity constraints and conservation conditions.
- A maximum flow represents the largest allocation of a resource that simultaneously satisfies those constraints.

Thus, solving a network flow problem amounts to determining the optimal allocation of limited capacity through a constrained system.

Importantly, the constraints defining feasibility are linear. Flow conservation imposes linear balance equations at each intermediate vertex, and capacity constraints impose linear upper and lower bounds on each edge.

Since both the objective $|f|$ and the constraints are linear in the flow variables, the maximum flow problem can be expressed as a linear optimization problem.

This observation allows us to move to a continuous optimization perspective when describing flows.

3.4 From Flow Structure to Linear Optimization

Network flow provides structural intuition for allocation under constraints. However, the mathematical problem being solved is not inherently graphical. It is a linear optimization problem.

While flow offers modeling clarity and interpretability, the task allocation problem itself is solved as a linear program where:

- objectives and constraints are explicit,
- feasibility is encoded through linear relations,
- global tradeoffs are quantified through optimization.

Thus, flow serves as a structured representation, but linear programming provides the computational and theoretical foundation.

We can now shift from network flow's graphical representation to the general framework of linear optimization. In the next section we will also show an example of this transformation from a network flow into a linear program.

3.5 Linear Programming and Duality

Definition 3.15 (Components of a Linear Program)**.** A linear program consists of three parts:

- **Decision variables:** quantities we choose, typically written as an $n \times 1$ vector, x.
- **Objective function:** a linear expression in the decision variables that we aim to minimize or maximize, such as $c^T x$, where c is a cost vector that assigns a cost (or value) to each variable.
- **Constraints:** linear conditions on the decision variables that determine which solutions are allowed. These encode feasibility or resource limits and are written in forms such as

  $$Ax \geq b, \quad Ax \leq b, \quad \text{or} \quad Ax = b,$$

  where A is a $m \times n$ matrix that describes how variables interact within each constraint, and b is a $m \times 1$ vector that represents the required value.

Definition 3.16 (Primal and Dual Linear Programs)**.** Let A be an $m \times n$ matrix, x an $n \times 1$ vector of decision variables, c an $n \times 1$ cost vector, and b an $m \times 1$ vector representing resource requirements.

The linear program

$$\begin{aligned} \text{Minimize} \quad & c^T x \\ \text{subject to} \quad & Ax \geq b, \\ & x \geq 0 \end{aligned}$$

is called the *primal LP*.

Associated with it is the *dual LP*

$$\begin{aligned} \text{Maximize} \quad & b^T y \\ \text{subject to} \quad & A^T y \leq c, \\ & y \geq 0 \end{aligned}$$

where y is an $m \times 1$ vector corresponding to the constraints of the primal LP.

The dual is obtained by transforming constraints into variables and variables into constraints. This transformation follows the pattern below:

Primal Element	Dual Element
Minimize $z = c^\mathsf{T} x$	Maximize $w = b^\mathsf{T} y$
$\sum_j a_{ij} x_j \geq b_i$	$y_i \geq 0$
$\sum_j a_{ij} x_j = b_i$	y_i unrestricted
$\sum_j a_{ij} x_j \leq b_i$	$y_i \leq 0$
$x_j \geq 0$	$\sum_i a_{ij} y_i \leq c_j$
x_j unrestricted	$\sum_i a_{ij} y_i = c_j$
$x_j \leq 0$	$\sum_i a_{ij} y_i \geq c_j$

Figure 3.7: Correspondence between primal and dual elements. From Shahriar Shahriari, *Linear Programming, Duality, and Games*

Remark 3.17. The primal focuses on allocating resources subject to constraints, while the dual assigns values to those constraints. Together, they provide two equivalent views of the same optimization problem.

Lemma 3.18 (Weak Duality)**.** *Let x be any feasible solution to the primal and y any feasible solution to the dual. Then*

$$c^T x \geq b^T y.$$

Remark 3.19. Every feasible dual solution provides a lower bound on the primal objective. Equivalently, every feasible primal solution provides an upper bound on the dual objective.

Lemma 3.20 (Weak Duality Optimality Criterion)**.** *If x and y are feasible and*

$$c^T x = b^T y,$$

then both x and y are optimal.

Remark 3.21 (Scarcity and Value Interpretation). Each dual variable corresponds to a constraint in the primal and measures how much the objective would improve if that constraint were relaxed (that is, if its bound were increased).

Once we solve the dual, these values are determined explicitly. Constraints with larger dual values are more limiting, while those with value 0 do not affect the solution.

Thus, the primal determines how to allocate resources, while the dual reveals how scarce those resources are.

Thus, solving the dual does more than guarantee optimality. It decides which constraints are limiting and how strongly they affect the objective. These values are often called ***shadow prices.***

Remark 3.22. These values can be computed using standard methods such as the simplex or dual simplex algorithm. However in this work, we rely on software such as `R`.

We can show this connection by expressing a network flow problem as a linear program and examining its dual.

Example 3.23 (Network flow as a linear program and its dual)**.** Consider the network in Figure 3.8. Let each variable represent the flow along its corresponding edge.

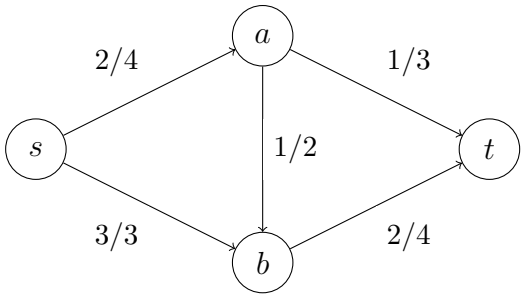


Figure 3.8: A directed network with edge labels cost/capacity.

We can express this network flow problem as a linear program by assigning a variable to each edge and enforcing capacity and flow conservation constraints.

We get the primal LP below:

$$\begin{aligned}
\text{Maximize} \quad & x_{sa} + x_{sb} \\
\text{subject to} \quad & x_{sa} = x_{at} + x_{ab} \\
& x_{sb} + x_{ab} = x_{bt} \\
\\
& x_{sa} \leq 4, \quad x_{sb} \leq 3 \\
& x_{at} \leq 3, \quad x_{bt} \leq 4 \\
& x_{ab} \leq 2 \\
\\
& x_{sa}, x_{sb}, x_{at}, x_{bt}, x_{ab} \geq 0
\end{aligned}$$

Solution. An optimal solution is

$$x_{sa} = 4, \quad x_{sb} = 3, \quad x_{at} = 3, \quad x_{bt} = 4, \quad x_{ab} = 1,$$

which gives a maximum flow of 7.

Dual. Introducing a dual variable for each constraint, the corresponding dual LP is

$$\begin{aligned}
\text{Minimize} \quad & 4y_{sa} + 3y_{sb} + 3y_{at} + 4y_{bt} + 2y_{ab} \\
\text{subject to} \quad & y_{sa} \geq 1 \\
& y_{sb} \geq 1 \\
& -y_{sa} + y_{at} \geq 0 \\
& -y_{sb} + y_{bt} \geq 0 \\
& -y_{sa} + y_{sb} + y_{ab} \geq 0 \\
& y_{sa}, y_{sb}, y_{at}, y_{bt}, y_{ab} \geq 0
\end{aligned}$$

An optimal dual solution is

$$y_{at} = 1, \quad y_{bt} = 1,$$

with all other variables equal to 0, giving value 7.

Dual and shadow prices. We determine the dual variables by measuring how the objective changes when constraints are relaxed.

For the constraint

$$x_{at} \leq 3,$$

increasing the bound to 4 increases the objective from 7 to 8, so its shadow price is 1.

Similarly, increasing

$$x_{bt} \leq 4$$

to 5 increases the objective by 1, so its shadow price is also 1.

All other constraints can be relaxed without increasing the objective, so their shadow prices are 0.

Interpretation. Only the constraints

$$x_{at} \leq 3 \quad \text{and} \quad x_{bt} \leq 4$$

limit the solution. Their shadow prices show how much the objective would increase per unit increase in their bounds.

Thus, solving the dual identifies which constraints matter and how scarce they are.

Chapter 4

Flow-Based Dynamic Reward Design for Reinforcement Learning

So far, we have seen that constraints can be folded into a reward function through Lagrange multipliers, and that solving the dual of a linear program tells us how scarce each resource actually is.

We now use that idea directly to build a reward function that updates as resources are consumed, so the agent naturally adjusts its behavior as the budget changes.

4.1 Problem Setting and Design Objective

We now consider a setting where a reinforcement learning agent must act under limited shared resources such as time and energy, where every action both contributes toward completing tasks and consumes part of the available budget. As a result, the agent must decide which tasks are worth completing, since not all of them can be done within the given constraints.

The main challenge is that these resource constraints build up over time, while reinforcement learning makes decisions one step at a time based on immediate rewards. Hence, the problem is not just learning a policy, but designing a reward function that reflects these global tradeoffs.

Our approach is to use linear optimization to reason about which tasks are feasible under resource limits, and then use this structure to shape the

reward function with the goal of guiding the agent toward decisions that balance task completion with resource usage.

4.2 Framework Overview and Environment Model

To incorporate resource constraints, we augment the state to include the agent's remaining resources:

$$S_t = (s_t, R_t),$$

where s_t is the original environment state and R_t represents our resources such as remaining time and energy. Resources will change based on the action taken. For example, if an action a consumes some amount of resources, we can write this as

$$R_{t+1} = R_t - \Delta R(a),$$

where $\Delta R(a)$ represents the cost of that action.

These resource costs are determined by the agent's environment.

4.2.1 Environment

We model this setting using a grid-based environment in which the agent navigates and completes tasks under limited resources, defined by the following components:

- **Grid world:** The agent moves between neighboring cells on a known map.
- **Baseline movement cost:** Each action consumes a small amount of time and energy, reflecting the cost of taking a step.
- **Delta squares:** Certain squares modify resource usage in a specific way. For example, moving through one square may consume more time but less energy, while another may do the opposite.
- **Unknown dynamics:** While the map is known, the agent does not initially know how different regions affect resource consumption. This is left for reinforcement learning itself to explore.

- **Goals:** The environment contains multiple goal locations where each goal provides a reward and requires varying resources to complete.
- **Non-repeating tasks:** Once a goal is completed, it cannot be repeated, and the agent may continue to other goals if resources allow.

This setup creates a setting where the agent must decide not only how to move, but which goals are worth attempting given limited resources. Different paths and goals induce different tradeoffs, and not all combinations are feasible within the available budget.

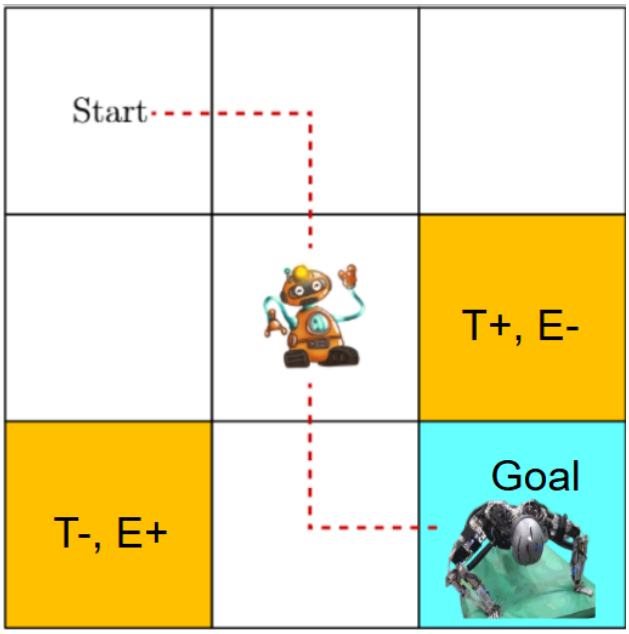


Figure 4.1: An example environment

Figure 4.1 highlights these costs. Different regions mean different time and energy tradeoffs.

These local costs accumulate over time, and the best sequence of actions depends on how much time and energy the agent has left at that given point in time. This is the type of setting where local reward signals alone are not enough to determine what the agent should do next.

At each state, we consider the set of available tasks along with their rewards and associated resource costs. Using the current resource levels

R_t, we solve an optimization problem to determine which tasks are worth prioritizing.

As the agent moves and resources are consumed, this optimization is recomputed, updating how important each resource is at the current state.

Putting this together, the system works as a loop:

- The environment provides the current state and remaining resources.
- An optimization problem evaluates which tasks are feasible.
- The result is used to update the reward function.
- The reinforcement learning agent selects actions based on this modified reward.

This raises a natural question: given the current resources, which goals are actually worth attempting? We answer this by formulating an optimization problem that evaluates task feasibility at each state.

4.3 Resource Allocation via LPs

Under this abstraction, we can model the environment as a collection of tasks $i = 1, \ldots, N$. Where each task has:

- reward r_i
- time cost t_i
- energy cost e_i

Let f_i indicate whether task i is selected. Ideally, $f_i \in \{0, 1\}$, but we instead allow $f_i \in [0, 1]$ to obtain a relaxed linear program that is easier to solve.

We can then write the resource allocation problem as:

$$\begin{aligned} \max \quad & \sum_{i=1}^{N} r_i f_i \\ \text{subject to} \quad & \sum_{i=1}^{N} t_i f_i \leq T \\ & \sum_{i=1}^{N} e_i f_i \leq E \\ & f_i \geq 0 \quad \forall i \end{aligned}$$

Solving this LP gives us our maximum reward under the current constraints and produces dual variables associated with the resource limits. The dual variables denoted by λ_T and λ_E correspond to time and energy, and measure how much the objective would improve if more of each resource were available. Larger values indicate that a resource is limiting, while values closer to zero mean that it isn't.

We use these values to guide the reward function, allowing the agent to account for resource scarcity during decision making.

4.4 Dynamic Reward Construction

We now use the dual variables to construct a reward function that accounts for resource constraints.

We begin with the standard reinforcement learning reward

$$r_{\text{env}}(s, a, s'),$$

which captures progress toward completing tasks but does not account for resource usage.

To incorporate resource constraints, we modify this reward using the shadow prices obtained from the optimization step:

$$r'(s, a, s') = r_{\text{env}}(s, a, s') - \big(\lambda_T \, \Delta T(a) + \lambda_E \, \Delta E(a)\big),$$

where $\Delta T(a)$ and $\Delta E(a)$ are the time and energy costs of action a.

Each part of this reward has a clear role. The term r_{env} encourages progress toward completing tasks. The Δ terms measure how much resource an action consumes locally. The λ values weight those costs based on how scarce each resource is.

When a resource is scarce, its corresponding λ is large, increasing the penalty for using it. When it is not limiting, the penalty is small. This allows the reward to adapt as the agent moves and resources change.

This construction changes how the agent evaluates actions. Instead of focusing only on immediate reward, decisions are shaped by how they use limited resources, and constraints become part of the objective itself rather than being enforced externally.

Chapter 5

Conclusion

5.1 Is it worth it?

This work looked at whether ideas from optimization can make reinforcement learning more structured and aware of resource limits. In the right setting, it can.

The main idea is to use dual variables as shadow prices to update the reward as resources change. When something becomes scarce, the agent reflects that through the reward and adjusts how it acts. This provides a natural way to incorporate constraints into the learning process. This directly addresses one of the core challenges in reinforcement learning: reward design. Instead of manually encoding complex constraints into a fixed reward, the structure of the problem is used to shape the reward dynamically.

At the same time, this remains more of a framework than a finished system. The experimental side is limited, and there are open questions about how well this scales and whether the approximations hold up in practice.

One limitation is state explosion. As the environment becomes more complex, the state grows with it, and solving an optimization problem at each step becomes more and more expensive. This slows the system down and makes it harder to use in larger or more realistic settings.

In addition, the method relies on a relaxed formulation, which may not fully capture discrete decision making.

Overall, the approach works best in structured settings where these costs remain manageable, and should be viewed as a way to add structure and interpretability to the reward design rather than a perfect solution.

5.2 Next steps

The next step is to complete the implementation and evaluate the approach more carefully. This includes finishing the code for training the reinforcement learning agent and comparing it against a baseline model.

A natural baseline is one where the optimization is solved once at the beginning, rather than being updated throughout the episode. We can then compare how quickly each approach learns, along with their performance in terms of total reward and resource usage such as time and energy.

It would also be useful to test how this method behaves in larger environments, where solving an optimization problem at each step may become more costly. This would give a clearer sense of when this approach is practical and where it falls apart.

Acknowledgments

I would like to thank my thesis advisor, Professor Konrad Aguilar, for his guidance, encouragement, and thoughtful check-ins throughout the process. His support was instrumental in shaping this work.

I am also grateful to Professor Shahriar Shahriari for allowing me to reference his textbook and for the inspiration his teaching has provided.

Finally, thank you to my friends and professors who listened patiently, offered feedback, and helped me think more clearly about my ideas. Your conversations and support meant a great deal.

Technology Statement

I would also like to acknowledge the tools utilized in writing this thesis. Grammarly and ChatGPT were used to assist with grammar, punctuation, and LaTeX formatting; however, the ideas presented remain my own or are properly attributed to their respective authors as cited in the bibliography.

Bibliography

[1] J. D. Alvarado, Y. Kohayakawa, P. Morris, and G. O. Mota, *A canonical Ramsey theorem with list constraints in random (hyper-)graphs*. arXiv:2304.01846 [math.CO] (2025), 16 pp.

[2] NVIDIA, *What Is Reinforcement Learning?* NVIDIA Glossary. `https://www.nvidia.com/en-us/glossary/reinforcement-learning/` (accessed April 12, 2026).

[3] S. Reddy, A. D. Dragan, and S. Levine, *SQIL: Imitation learning via reinforcement learning with sparse rewards*. arXiv:1905.11108 [cs.LG] (2019).

[4] S. Shahriari, *An Invitation to Combinatorics*. Cambridge Mathematical Textbooks, Cambridge Univ. Press, Cambridge, (2022).

[5] S. Shahriari, *Linear Programming, Duality, and Games*. Pomona College, Draft version 3.7, (2018).

[6] H. A. Taha, *Operations Research: An Introduction*. 10th ed. Pearson, London, (2017).

[7] S. Zhao, *Mathematical Foundations of Reinforcement Learning*. Springer Nature Press, (2025).