POLITECNICO
MILANO 1863

SCHOOL OF CIVIL, ENVIRONMENTAL AND LAND MANAGEMENT ENGINEERING

Optimal Truss Design using Reinforcement Learning

Tesi di Laurea Magistrale in
Civil Engineering - Ingegneria Civile

Author: **Syed Yusuf**

Student ID: 970804
Advisors: Prof. Alberto Corigliano and Dott. Ing. Luca Rosafalco
Academic Year: 2022-23

Abstract

In civil engineering, the process of designing, evaluating, and selecting the optimal design of a structure can be time-consuming and resource-intensive. Computational Design Synthesis (CDS) helps in automating this process, but selecting the optimal design from a large set of design options remains a challenge. Artificial intelligence, particularly reinforcement learning (RL), has shown potential to be used as an optimization algorithm. In this work, we explore the use of RL in optimal design synthesis by modeling the design synthesis process as a finite Markov Decision Process (MDP). In this framework, design configurations are represented as states, and available modifications are considered as actions. Immediate rewards are designed to reflect the improvement in the performance of the altered configuration in relation to the design objective consisting, for example, of compliance minimization. As immediate rewards in optimal design synthesis are typically unknown at the start of the process, RL is used to effectively solve the MDP. The objective of the RL agent is to maximize cumulative rewards, thereby generating the optimal design with the best performance. The suggested framework has been applied to obtain optimal designs for different planar trusses defined over different domains, loading conditions, and unique constraints. The results obtained show that this approach has the potential to be extended to complex design criteria. Overall, this work demonstrates the potential of RL as an optimization algorithm for CDS and provides insights for future research in this field.

Keywords: computational synthesis, truss optimization, design synthesis, machine learning for engineering applications, reinforcement learning

Abstract in lingua italiana

In ingegneria civile, il processo di progettazione, valutazione e selezione della soluzione di progettazione ottimale può richiedere molto tempo e risorse. I principi alla base della Computational Design Synthesis possono aiutare ad automatizzare questo processo, ma selezionare la soluzione di progettazione ottimale da un ampio set di opzioni di progettazione rimane una sfida. L'intelligenza artificiale, in particolare l'apprendimento per rinforzo (Reinforcement learning-RL), può essere utilizzata come algoritmo di ottimizzazione. In questo lavoro, esploriamo l'uso di RL all'interno della CDS modellando il processo di sintesi della progettazione come un Processo Decisionale di Markov (Markov Decision Process-MDP). In questa cornice procedurale, le configurazioni di progettazione sono schematizzate come stati, e le modifiche disponibili sono considerate come azioni. Le ricompense immediate sono progettate per riflettere il miglioramento delle prestazioni della configurazione modificata in relazione all'obiettivo di progettazione consistente, per esempio, nella minimizzazione della cedevolezza della struttura. Poiché le ricompense immediate nella sintesi di progettazione ottimale sono tipicamente sconosciute all'inizio del processo, RL è utilizzato per risolvere efficacemente un MDP. L'obiettivo dell'agente di RL è massimizzare le ricompense cumulative, generando un progetto di struttura con prestazioni ottimizzate. Questa metodologia è stata applicata per la progettazione ottimale di sistemi reticolri in piano definiti su domini diversi, e con diverse condizioni di carico e vincoli. I risultati ottenuti mostrano che questo approccio ha il potenziale per essere esteso ad altri problemi di ottimizzazione strutturale. Dimostrando il potenziale di RL come algoritmo di ottimizzazione.

Parole chiave: sintesi computazionale, ottimizzazione di sistemi reticolari, machine learning per applicazioni ingegneristiche, apprendimento per rinforzo

Contents

List of Figures

List of Tables

Introduction

Computational Design Synthesis (CDS) is an area of research that aims to automate the design process by generating, evaluating, and selecting candidate design solutions [1–4]. The design problem is usually specified as a set of objectives and constraints, and the goal of CDS is to identify the design solution that best satisfies these objectives and constraints [5]. CDS combines the use of generative design grammar rules such topological, spatial and/or parametric along with different optimization search heuristics such as genetic algorithms [6, 7], simulated annealing [8–10], branch and bound [11] and particle swarm optimization [12] to synthesize efficient and/or optimized design solutions for a range of engineering design problems. In this context, optimization algorithms are crucial for achieving this goal.

Optimization algorithms are used to search for the optimal solution within a large design space. The design space can be continuous or discrete, and the search for the optimal solution can be constrained by various criteria, such as performance, safety, cost, and environmental impact. However, searching the entire design space is computationally infeasible, especially for complex design problems. Therefore, optimization algorithms need to be computationally efficient to find optimal solutions within a reasonable amount of time.

Optimization algorithms, such as Genetic Algorithms and Particle Swarm Optimization, have been used to address this challenge. However, recent advancements in Artificial Intelligence (AI) techniques, particularly Machine Learning (ML), have shown great promise in finding optimal solutions in CDS. ML techniques have been applied to various aspects of design, including generating and evaluating design alternatives [13–15], exploring design space [16], and optimizing structures [17–19]. A specific area of interest is discrete design optimization, where the goal is to find the best combination of discrete design variables that satisfy certain constraints. One example of this is the use of a machine learning-based method by Hayashi and Ohsaki [20] to optimize truss structures by minimizing volume under stress and displacement constraints. Their approach involves a neural network and graph embedding model to find near-optimal topological designs, demonstrating that machine learning can efficiently produce sub-optimal solutions.

This work explores the application of Reinforcement Learning (RL), a branch of ML in the context of optimal truss design, a common problem in civil engineering. The goal is to model the design synthesis process as an Markov Decision Process (MDP) using RL algorithms, specifically Q-learning[21, 22] to train an agent to learn the best design modifications that satisfy a set of objectives and constraints, and to use this learning to generate optimal truss designs in a computationally efficient manner.

A finite MDP is a stochastic control process in discrete time represented by a 4-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$ comprising finite sets of possible decision states $\mathcal{S}$, valid actions $\mathcal{A}$, state transition probabilities $\mathcal{P}$, and possible rewards $\mathcal{R}$. In the context of optimal design synthesis, the MDP framework is used to model the design process, where states represent specific design configurations, actions represent possible alterations based on design grammars, and immediate rewards correspond to the improvement in performance resulting from the alterations. The objective of the RL agent is to maximize the cumulative reward by learning the optimal alteration to a given configuration that results in improved performance and synthesis of an optimal design.

By combining RL with traditional optimization algorithms, this work aims to improve the efficiency and effectiveness of the truss design process, and to demonstrate the potential of AI techniques in CDS for civil engineering applications.

Chapter 1 of this manuscript delves into the topic of CDS, starting with an overview of its evolution over time. The chapter then explores in detail the various optimization algorithms commonly used in CDS, including genetic algorithms, particle swarm optimization, simulated annealing, gradient-based algorithms, and branch and bound methods. The theoretical foundations of each algorithm are discussed, along with their strengths and weaknesses and their typical applications in CDS. Finally, the chapter explores the role of AI in CDS, including how AI has been used to enhance existing CDS algorithms and facilitate the development of new CDS techniques.

Chapter 2 delves into the application of Reinforcement Learning (RL) in CDS. After introducing the fundamental concepts of RL, such as the Markov Decision Process and the Agent-Environment interface, the chapter covers the essentials of Returns and Episodes, Policies and Value Functions, and Exploration and Exploitation. It then goes on to provide an overview of the different types of RL algorithms, including Model-Based, Model-Free, and Actor-Critic algorithms. The chapter also explores the practical application of RL in CDS and also introduces the concept of Q-learning.

Chapter 3 focuses on the application of Q-learning to the optimization of truss design. The chapter first introduces the design problem and the linear finite element analysis used to evaluate the designs. It then discusses the learning environment for the Q-learning algorithm, including the initialization of Q-table, reward functions, parameter selection and convergence criteria.

Chapter 4 presents results of applying RL to truss design synthesis with certain case studies, including a planar truss with a single force, a planar truss with multiple loading conditions, and a bridge truss. Overall, the chapter shows the effectiveness of using RL methods to solve complex engineering design problems, particularly in truss design.

In the last chapter, an analysis and discussion of the obtained results is reported, highlighting the novelties and weaknesses of the proposed method; finally, some potential future developments based on this work are also suggested.

1 | Computational Design Synthesis

1.1. Introduction

Computational Design Synthesis (CDS) is a term used to describe the use of computer algorithms and software to assist in the creation of new designs. This approach involves employing various computational techniques to explore a vast range of design options, generate and evaluate multiple possible solutions, and identify the best design based on predefined criteria.

The primary objective of CDS is to automate the process of design generation and exploration, making it faster, more efficient, and less prone to errors. By utilizing computational techniques, designers can explore design alternatives and optimize their designs in ways that would be difficult or time-consuming to achieve manually.

CDS is a rapidly evolving field that combines computer science and engineering to create innovative design solutions. This chapter will explore the history and evolution of CDS and highlight some of the key algorithms used in the field.

1.2. Evolution of CDS

The roots of CDS can be traced back to the early days of computing. During the 1960s and 1970s, computer-aided design (CAD) systems, which help in automating the design process, were developed. Although, initially these systems focused on 2D drafting and simple geometric modeling, later developed into more sophisticated tools.

The Sketchpad system, developed by Ivan Sutherland [23] was one of the earliest examples of CDS. It allowed users to draw and manipulate geometric shapes using a light pen, which could be used to input commands directly into the computer. This system was groundbreaking and laid the foundation for many of the CAD tools that are in use today.

In the 1980s, researchers began developing more advanced CAD systems that could handle 3D modeling and simulation. One of the key contributions was the development of the Boundary Element Method (BEM), which allowed for more accurate and efficient simulation of complex engineering systems. This method was pioneered by C. L. P. Chen [24] and his team in the early 1980s.

In the 1990s, researchers began exploring the use of genetic algorithms [25] and artificial intelligence to aid in the design process. Genetic algorithms are a type of optimization algorithm that mimics the process of natural selection. These algorithms can be used to generate designs that meet specific criteria, and they have been applied to a wide range of design problems, from architectural design to product design.

During the 1990s, parametric design tools emerged as a key development in CAD systems [26]. These tools allowed for the creation of designs that could be easily adapted and modified to suit different contexts. Parametric design tools use algorithms to generate designs based on a set of parameters, such as size, shape, and material properties. Grasshopper and Dynamo being some of the most widely used tools.

In recent years, machine learning and deep learning algorithms have made significant breakthroughs in the field, allowing for the analysis of large datasets of existing designs and the generation of new designs that meet specific criteria [27]. These advancements have the potential to revolutionize the design process, enabling designers to create innovative solutions that would have been impossible to generate using traditional methods.

Furthermore, a range of CDS tools and techniques is available, including generative design, topology optimization, evolutionary algorithms, and artificial intelligence-based methods. These tools can be applied across a broad spectrum of design applications, such as architecture, product design, engineering, and urban planning.

1.3. Optimization Algorithms

The process of CDS generally involves three key stages: *ideation, exploration*, and *optimization*. During the *ideation* phase, designers define the problem, objectives, and constraints of the design task. In the *exploration* phase, they generate and evaluate a large number of design alternatives using computational algorithms and tools. Finally, in the *optimization* phase, they utilize optimization algorithms to identify the optimal design based on predefined criteria.

Optimization algorithms play a crucial role in CDS by enabling designers to identify the best solutions for complex design problems. In this section, we will discuss some of the optimization algorithms commonly used in CDS and highlight their strengths and weaknesses.

1.3.1. Genetic algorithms

Genetic algorithms (GA) are inspired by the process of natural selection and evolution. They operate by creating a population of potential solutions and using crossover and mutation operators to generate new solutions. These new solutions are evaluated based on a fitness function that measures how well they satisfy the design criteria. The fittest individuals are selected for the next generation, and the process continues until a satisfactory solution is found. GA is a popular optimization algorithm in CDS due to its ability to handle large search spaces and non-linear problems [25].

1.3.2. Particle Swarm Optimization

Particle Swarm Optimization (PSO) is another popular optimization algorithm in CDS that is inspired by the behavior of swarms. It operates by creating a swarm of particles that move around in search of the best solution. Each particle represents a potential solution, and its movement is influenced by its own best position and the best position of the swarm. PSO is often used in problems with multiple objectives, and it can converge quickly to a satisfactory solution. However, it may struggle with problems that have a large number of design variables [12].

1.3.3. Simulated Annealing

Simulated Annealing (SA) is a meta-heuristic optimization algorithm that is inspired by the process of annealing in metallurgy. It operates by starting with an initial solution and gradually decreasing the temperature of the system. As the temperature decreases, the algorithm becomes more likely to accept worse solutions in the search for a better solution. SA is often used in problems with complex search spaces and has the advantage of being able to escape local optima. However, it can be sensitive to the choice of parameters, and finding a good cooling schedule can be challenging [28].

1.3.4. Gradient-based Algorithms

Gradient-based optimization algorithms use the gradient of the objective function to find the optimal solution. They are at the basis of classical topology optimization. They operate by iteratively adjusting the design variables in the direction of the gradient until a satisfactory solution is found. Gradient-based optimization algorithms are often used in problems with smooth objective functions and a small number of design variables. They are computationally efficient and can converge quickly to a satisfactory solution. However, they may struggle with problems that have non-convex or discontinuous objective functions [29].

1.3.5. Branch and Bound

Branch and bound (B&B) is a commonly used algorithm in CDS that can solve combinatorial optimization problems. It is particularly useful in problems with discrete variables, where the search space can be large and cannot be easily explored by other optimization methods. The basic idea of the B&B algorithm is to recursively partition the search space into smaller sub-spaces, eliminating sub-spaces that cannot contain the optimal solution based on a lower bound estimate. The algorithm then explores the remaining sub-spaces until a satisfactory solution is found. The B&B algorithm has been widely used in CDS for problems such as facility location, network design, and scheduling. It has been shown to be effective in finding optimal solutions for problems with discrete and combinatorial variables, but its performance can be sensitive to the problem size and the quality of the lower bound estimate. The algorithm can also become computationally expensive for large problems due to its recursive nature [30].

In conclusion, optimization algorithms play a crucial role in CDS by enabling designers to identify the best solutions for complex design problems. Genetic algorithms, particle swarm optimization, simulated annealing, gradient-based optimization and branch and bound are some of the most commonly used optimization algorithms in CDS. Each algorithm has its own strengths and weaknesses, and the choice of algorithm will depend on the specific problem being solved.

1.4. Artificial Intelligence in CDS

Artificial Intelligence methods such as Machine learning (ML) has been increasingly used in CDS as it provides powerful tools for learning patterns and relationships from data. ML algorithms can be used to analyze design data, generate design alternatives, and optimize design solutions. ML is particularly effective in CDS problems where the search space is large and the design criteria are complex and often involve multiple conflicting objectives.

One popular ML algorithm used in CDS is artificial neural networks (ANNs), which are modeled after the structure of the human brain. ANNs can learn complex patterns and relationships from data and can be used to predict design outcomes based on a set of input parameters. ANNs have been used in various CDS applications such as predicting building energy consumption [31], optimizing building envelopes [32], and generating new building designs [33].

Another ML algorithm used in CDS is reinforcement learning (RL), which is a technique for learning optimal decision-making policies through interactions with the environment. RL has been used in CDS for various applications such as material design [34], urban design [35], and product design[36]. RL has shown promise in optimizing complex design solutions with multiple conflicting objectives, but its application in CDS is still in its early stages.

In addition to ANNs and RL, other ML algorithms such as decision trees, random forests, and support vector machines have also been used in CDS. These algorithms can be used for classification, clustering, and prediction tasks in various CDS applications.

As we can see ML provides powerful tools for analyzing, generating, and optimizing design solutions in CDS. RL is particularly effective in problems where the optimal solution cannot be easily defined or the search space is complex and dynamic. In the following section, we will discuss how RL works and its potential applications as an optimization algorithm in CDS.

2 | Reinforcement Learning

2.1. Introduction

The concept that we acquire knowledge by interacting with our surroundings is likely the first notion that comes to mind when considering the nature of learning. This connection yields a vast amount of information regarding causality, outcomes of actions, and strategies for attaining objectives. Throughout our lives, such interactions are undoubtedly a primary source of information about ourselves and our surroundings.

Reinforcement Learning (RL) is basically a type of machine learning that is based on the above principle. It involves an agent learning to make decisions in an environment through trial and error. The agent interacts with the environment and receives feedback in the form of rewards or penalties, based on its actions. The goal of this agent is to learn a policy that maximizes the cumulative reward it receives over time [22].

RL has been used to solve complex problems in various fields such as robotics, game AI, and autonomous vehicles. RL is particularly relevant to civil engineering because it can be used to optimize decision-making processes in complex and dynamic environments. Civil engineering involves many decision-making processes that can benefit from the use of RL, such as traffic control [37], energy management [38], and structural design [39]. RL can be used to optimize traffic signals to reduce congestion, to manage energy consumption in buildings, and to design more efficient and reliable structures.

Although RL has many applications in civil engineering, in this study we will focus mainly on its use in the CDS. In this chapter, we will provide an overview of the fundamental concepts of RL, including Markov Decision Processes, agents, states, actions, rewards, value functions, policies, and exploration vs. exploitation trade-offs. We will also review the most common RL algorithm, Q-learning and explore its application as an optimization algorithm.

2.2. Basic concepts of RL

A RL problem is defined using three main concepts *state, action*, and *reward.*

- *States* are a representation of the current environment of the task [22]. In structural optimization, we can think of states as the current conditions of the system we are trying to optimize.
- *Actions*, on the other hand, are the decisions that can be taken to modify the state of the system [22]. Like in structural design, the actions could be changing the dimensions of the structure or changing the material properties.
- *Reward* is the feedback that is received after taking an action in a given state [22]. Rewards can be positive, negative, or zero, and they reflect how desirable the outcome of the action was. In structural design problem, the reward could be positive if the structure can withstand higher loads after changing the dimensions.

By using RL, we learn to take the best action in a given state by maximizing the expected cumulative reward over time. This enables us to optimize complex decision-making processes in civil engineering.

2.3. Markov Decision Process

At the heart of RL lies the Markov Decision Process (MDP), which is a mathematical framework for modeling decision-making problems. An MDP consists of a set of states, actions, transition probabilities, rewards, and a discount factor [22]. MDPs provide a mathematically idealized framework for reinforcement learning, allowing precise theoretical statements. This section introduces key mathematical elements of MDPs such as Policies, Returns, Value functions, and Bellman equations.

2.3.1. The Agent-Environment Interface

An MDP consists of a learner and a decision maker called *agent* and a surrounding with which it interacts called *environment.* The agent interacts with the environment by selecting actions from a set of possible actions, and the environment responds by transitioning to a new state and generating a reward. This process is repeated until the agent has maximized the rewards and obtained the optimal action-state pairs.

The finite Markov decision process involves a sequence of discrete time steps where the agent and environment interact. At each time step, represented by $t = 0, 1, 2, 3, ...$, the

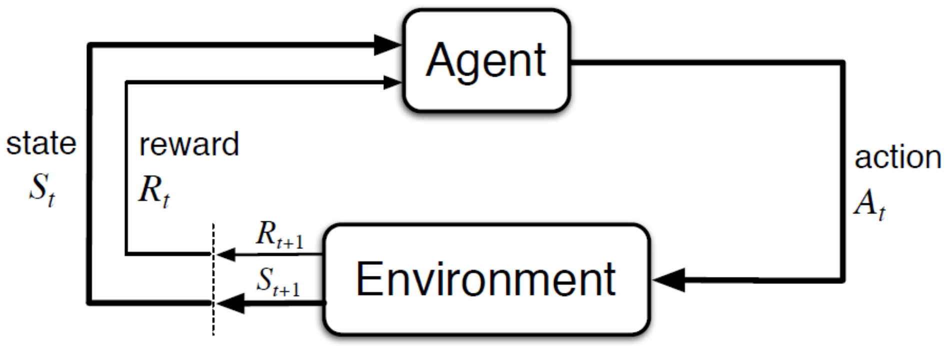


Figure 2.1: The agent–environment interaction in a Markov decision process.

agent receives information about the *state* of the environment, S_t, from a set of possible states, $S_t \in \mathcal{S}$. Based on this information, the agent selects an *action*, $A_t \in \mathcal{A}$. As a consequence of this action, the agent receives a numerical *reward*, $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ and the agent finds itself in a new state, S_{t+1}. This gives rise to a trajectory that progresses through the sequence of time steps.

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots \tag{2.1}$$

A decision process is Markovian if the random variables representing the current state and the immediate reward, i.e., S_t and R_t, respectively, depend only on the preceding state and action.

2.3.2. Finite Markov Decision Process

A finite MDP is characterized by having a finite number of states, actions, and rewards denoted as ($\mathcal{S}, \mathcal{A}$, and $\mathcal{R}$) respectively. Due to the Markovianity of the decision process, for a particular set of values of these variables, such as $s^{'} \in \mathcal{S}$ and $r \in \mathcal{R}$, the probability of their occurrence at a specific time t is determined by the preceding state and action.

$$p(s', r|s, a) = Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}, \tag{2.2}$$

for all $s', s \in \mathcal{S}, r \in \mathcal{R}$, and $a \in \mathcal{A}(s)$. The function p defines the dynamics of the MDP. The dynamics function p is a standard function with four inputs: the current state s, the action a taken by the agent, the resulting reward r, and the next state s'. It outputs a probability value between 0 and 1 that represents the probability of transitioning from state s to state s' and receiving reward r, given that action a was taken. The notation $p(s', r|s, a)$ represents this conditional probability distribution, which specifies the likelihood of transitioning to state s' and receiving reward r, given the current state s and action a, Consequently, it holds that: [22]

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) = 1, \text{for all} \quad s \in \mathcal{S}, a \in \mathcal{A}(s) \tag{2.3}$$

2.3.3. Markov Property

The aforementioned Markov property can be stated formally as follows: a state S_t is said to have the Markov property with respect to a sequence of environment *states, actions, and rewards* $S_t, A_t, R_t, S_{t+1}, A_{t+1}, R_{t+1}, ...$ if and only if

$$Pr[S_{t+1}, R_{t+1}|S_t, A_t, R_t, ..., S_0, A_0, R_0] = Pr[S_{t+1}, R_{t+1}|S_t, A_t]. \tag{2.4}$$

That is, the probability distribution of the next state and next reward depends only on the *current state* and *action*, and not on any past states, actions, or rewards beyond the immediately preceding one. In other words, the future is conditionally independent of the past given the present [22].

Once the dynamics function (p) given by 2.2 is defined for a Markov decision process, it becomes possible to calculate various other aspects of the environment, such as the *expected rewards* associated with different state-action pairs.

$$r(s, a) = \mathbb{E}[R_{t+1}|S_t = s, A_t = a] = \sum_{r \in \mathcal{R}} \sum_{s' \in \mathcal{S}} p(s', r|s, a), \tag{2.5}$$

the *state-transition probabilities*,

$$p(s'|s, a) = Pr\{S_{t+1} = s'|S_t = s, A_t = a\} = \sum_{r \in \mathcal{R}} p(s', r|s, a), \tag{2.6}$$

and the expected rewards for state-action-next-state triples,

$$r(s, a, s') = \mathbb{E}[R_{t+1}|S_t = s, A_t = a, S_{t+1} = s'] = \frac{\sum_{r \in \mathcal{R}} p(s', r|s, a)}{p(s'|s, a)} \tag{2.7}$$

2.4. Returns and Episodes

The goal of the agent is to maximize the cumulative reward it receives in the long run. To formalize this, we define the expected return, which is the expected total reward obtained by the agent over time. The return at time step t, denoted by G_t, is a specific function of the reward sequence $R_{t+1}, R_{t+2}, R_{t+3}, \ldots$ and in the simplest case, it is the sum of the rewards:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \ldots + R_T, \tag{2.8}$$

where T is the final step. The approach of defining the return as the sum of rewards makes sense in scenarios where the agent-environment interaction naturally divides into sub-sequences known as episodes. Each episode concludes in a specific state called the terminal state, followed by a reset to a standard starting state. These tasks are known as episodic tasks. As it will be shown, such a notion can be applied to the optimization of truss structures.

The concept of discounting is an important addition to the notion of maximizing the expected return. In the discounted return setting, the agent aims to select actions that maximize the sum of discounted rewards it receives in the future. Specifically, the agent chooses action A_t to maximize the expected discounted return:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.9}$$

where γ is the discount rate, a scalar value between 0 and 1 that determines the importance of immediate versus future rewards. It can be intuitively interpreted as the trust, that the decision process will go on. A discount rate of 0 means that the agent only cares about the immediate reward, while a discount rate of 1 means that the agent considers all future rewards equally important. The discount rate allows the agent to balance the trade-off between immediate and delayed rewards, and also ensures that the sum of rewards is finite even in infinite horizon tasks.

Returns at successive time steps are related to each other in a way that is important for the theory and algorithms of reinforcement learning:

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{2.10}$$

This formula works for all time steps $t < T$, even if termination occurs at time step $t+1$, if we define $G_T = 0$. This allows us to easily compute returns from reward sequences.

It's worth noting that although the return formula involves an infinite number of terms, it is still finite if the reward is nonzero and constant, and if $\gamma < 1$. For example, if the reward is a constant $+1$, then the return formula simplifies to:

$$G_t = \sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma} \tag{2.11}$$

2.5. Policies and Value Functions

In RL, value functions play a crucial role in almost all algorithms. These functions estimate the quality of a given state (or state-action pair) in terms of the future rewards that can be expected. The concept of "quality" is defined in terms of the expected return, which depends on the actions taken by the agent. Therefore, value functions are defined with respect to specific policies, which represent ways of acting in the environment.

A policy in RL can be defined as a function that maps states to probabilities of selecting each possible action. Formally, if the agent is following policy π at time t, then $\pi(a|s)$ represents the probability of taking action a when in state s and $A_t = a$ if $S_t = s$. The symbol "|" indicates that $\pi(a|s)$ defines a probability distribution over $a \in \mathcal{A}(s)$ for each $s \in S$. In RL, the agent's policy is modified based on its experience using specific algorithms.

The value function of a state s under a policy π, denoted $v_\pi(s)$, is the expected return when starting in s and following π thereafter. For MDPs, we can define v_π formally by

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right], \text{for all s} \in \mathcal{S}, \tag{2.12}$$

where $\mathbb{E}_\pi[\cdot]$ denotes the expected value of a random variable given that the agent follows policy π, and t is any time step. Note that the value of the terminal state, if any, is always zero. We call the function v_π the *state-value function for policy* π.

The value of taking an action a in a state s while following policy π is represented by the function $q_\pi(s, a)$. This value is the expected return that the agent will receive after taking the action a in state s and then following the policy π. To calculate this value, we consider the expected sum of future rewards that the agent will receive after taking the action a in state s at time t and then following the policy π:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a] \tag{2.13}$$

We call q_π the *action-value* function for policy π.

2.6. Optimal Policies and Optimal Value Functions

The goal of RL is to find a policy that can maximize the cumulative reward over time. In finite MDPs, an optimal policy can be precisely defined using value functions. A policy π is considered to be better than or equal to another policy π' if it achieves greater or equal expected return for all states. An *optimal policy* is one that is better than or equal to all other policies. There may be multiple optimal policies, denoted by π_* . All optimal policies share the same state-value function, called the *optimal state-value function*, which is denoted by v_* and is defined as the

$$v_*(s) = \max_\pi v_\pi(s) \tag{2.14}$$

for all $s \in \mathcal{S}$.

Optimal policies also share the same *optimal action-value function*, denoted q_* , and defined as

$$q_*(s,a) = \max_\pi q_\pi(s,a) \tag{2.15}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$. For the state–action pair (s,a), this function gives the expected return for taking action a in state s and thereafter following an optimal policy. Thus, we can write q_* in terms of v_* as follows:

$$q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a] \tag{2.16}$$

For finite MDPs, the Bellman optimality equation for the optimal state-value function, denoted as $v_*(s)$, can be written as:

$$v_*(s) = \max_a \sum_{s',r} p(s',r|s,a)\,[r + \gamma v_*(s')] \tag{2.17}$$

where $p(s',r|s,a)$ is the probability of transitioning to state s' and receiving reward r

when taking action a in state s, and γ is the discount factor. The Bellman optimality equation for the optimal action-value function, denoted as $q_*(s, a)$, can be written as:

$$q_*(s,a) = \sum_{s',r} p(s', r|s, a) \left[r + \gamma \max_{a'} q_*(s', a') \right] \tag{2.18}$$

The Bellman optimality equation for $v_*(s)$ is a system of n equations in n unknowns, where n is the number of states. If the dynamics of the environment are known, this system of equations can be solved for $v_*(s)$ using any method for solving systems of nonlinear equations. Once $v_*(s)$ is obtained, an optimal policy can be determined by selecting the action that maximizes the right-hand side of the Bellman optimality equation for each state.

The optimal action-value function $q_*(s, a)$ can also be used to determine an optimal policy without having to know anything about the environment's dynamics. For each state s, the action that maximizes $q_*(s, a)$ is an optimal action. Hence, an optimal policy can be obtained by selecting the optimal action for each state. The optimal action-value function effectively caches the results of all one-step-ahead searches, providing the optimal expected long-term return as a value that is locally and immediately available for each state-action pair.

In summary, while explicitly solving the Bellman optimality equation provides a route to finding an optimal policy, it is rarely directly useful due to the assumptions of accurate knowledge of the environment's dynamics and sufficient computational resources. As a result, approximate solutions must be settled for. Many decision-making methods, including heuristic search methods and dynamic programming, can be viewed as approximations to the Bellman optimality equation. Reinforcement learning methods can also be understood as approximations to the Bellman optimality equation, using actual experienced transitions in place of knowledge of the expected transitions.

2.7. Exploration and Exploitation

In RL, the agent learns through exploration and exploitation. Exploration involves trying different actions to learn which ones result in the highest rewards and Exploitation involves using the knowledge gained to take actions that are most likely to lead to high rewards. The trade-off between exploration and exploitation is a fundamental challenge that arises in RL, and it distinguishes RL from other types of learning. The exploration-exploitation dilemma refers to the fact that the RL agent must balance its need to try new actions (exploration) with its need to exploit what it has learned so far (exploitation) to maximize reward. The agent needs to explore to find new strategies that may lead to higher rewards, but it also needs to exploit what it already knows to achieve its immediate goals. Exploration is necessary to avoid getting stuck in sub-optimal strategies or local optima, but too much exploration can result in inefficient use of time and resources. Exploitation is necessary to maximize short-term reward, but relying too much on familiar strategies can result in missing out on better long-term options. To address the trade-off between exploration and exploitation, the RL agent needs to find the right balance between the two. It needs to try a variety of actions, including ones it has not selected before, while also favoring actions that have been effective in producing rewards in the past. One solution to balance the trade-off is by using an ϵ-greedy policy which has been described in Section 2.10.2.

2.8. RL algorithms

RL algorithms can be classified into three types

1. Model-Based
2. Model-Free, and
3. Actor-Critic

2.8.1. Model-Based algorithms

Model-based RL algorithms require the agent to have a model of the environment, which includes information about the transition probabilities and rewards associated with each state and action. These algorithms use the model to compute the optimal policy or value function. One example of a model-based algorithm is dynamic programming (DP), which is well-suited for problems with a small state space and discrete actions [22]

2.8.2. Model-Free algorithms

Model-free RL algorithms do not require a model of the environment. Instead, they learn the optimal policy or value function by interacting with the environment and updating their estimates based on the observed rewards. One example of a model-free algorithm is Q-learning, which estimates the optimal action-value function using the Bellman equations and the observed rewards [21].

2.8.3. Actor-Critic algorithms

Actor-critic RL algorithms combine the strengths of both model-based and model-free approaches. These algorithms use a policy network to generate actions and a value network to estimate the value of each state. One example of an actor-critic algorithm is the Actor-Critic with Experience Replay (ACER) algorithm, which has been shown to be effective in continuous action spaces [40].

2.9. RL for CDS

CDS is a field which requires algorithms that can efficiently explore a large space of design options and identify the best designs. RL algorithms offer a promising approach to this challenge, as they can learn to optimize a design objective by interacting with an environment and receiving feedback in the form of rewards or penalties.

In the context of CDS, model-free algorithms are advantageous over model-based algorithms. Model-free algorithms are more flexible and can handle complex and dynamic environments where the transition probabilities are difficult to model [41]. This is particularly relevant for design synthesis, where the space of design options is often large and complex. Additionally, model-free algorithms can learn from experience and adapt to changing design objectives, as described in [42]. This is important in design synthesis, where the design objective may change as the design progresses or new requirements are added. Model-free algorithms can learn to optimize the design objective in real-time, without requiring a complete redesign of the system.

Furthermore, model-free algorithms can scale to large design spaces and handle high-dimensional input and output spaces. This is essential in design synthesis, where the number of design options can quickly become overwhelming. Model-free RL algorithms can efficiently search the design space and identify the best designs without getting stuck in local optima [43].

Overall, model-free RL algorithms offer a powerful and flexible approach to CDS. In the following section we will look at one of the most common model-free algorithm, Q-learning.

2.10. Q-learning

Q-learning is a popular and effective RL algorithm that addresses the limitations of explicitly solving the Bellman optimality equation for finite MDPs. Instead of relying on complete knowledge of the dynamics of the environment and an exhaustive search, Q-learning learns an estimate of the optimal action-value function, Q*, through trial-and-error experience. This allows Q-learning to handle large state and action spaces and to operate in non-Markovian environments. In the following sections, we will dive deeper into the Q-learning algorithm and explore its implementation and variations.

A key concept in RL is the temporal-difference (TD) learning, which combines ideas from Monte Carlo [44] and dynamic programming [45] methods. TD learning allows an agent to learn directly from raw experience without a model of the environment's dynamics, like Monte Carlo methods, and update its estimates based on other learned estimates, without waiting for a final outcome, like dynamic programming methods.

2.10.1. Temporal Difference

TD learning was introduced by Richard Sutton in the late 1980s and is now considered one of the most important ideas in reinforcement learning. The TD algorithm estimates the value function of the state-action pairs by making updates based on the difference between the estimate of the current state-action pair and the estimate of the next state-action pair. The difference between these estimates is known as the temporal difference error, which is used to update the current state-action pair's estimate. The TD error is the difference between the current estimate and the updated estimate and is the basis for the algorithm's name.

One of the primary benefits of TD learning is that it can learn online, i.e., while the agent is interacting with the environment, rather than relying on a pre-collected data-set. This online learning enables the agent to adapt to changes in the environment and can lead to faster convergence than other methods. Additionally, TD learning requires less memory and computation than Monte Carlo methods since it does not need to wait until the end of the episode to update the value function.

TD learning is a recursive algorithm that updates the value function for each state-action pair. The algorithm uses an update rule that is similar to the Bellman equation used in dynamic programming methods. The update rule is given by:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right] \tag{2.19}$$

where $V(S_t)$ is the estimate of v_π in state S at time t, G_t is the actual return following time t, and α is a constant step-size parameter.

2.10.2. Off-policy TD Control

Off-policy TD learning is a reinforcement learning technique that allows an agent to learn from a behavior policy that is different from the target policy. The behavior policy is the policy that the agent follows to interact with the environment, while the target policy is the policy that the agent is trying to learn to optimize.

In off-policy TD learning, the agent learns from the experiences generated by the behavior policy and updates its estimates using the target policy. This approach allows the agent to learn from a wider range of experiences and potentially converge to a better policy.

One of the early breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as Q-learning [21], defined by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{2.20}$$

where α is the learning rate, ruling the strength of the update. The learned action-value function Q approximates the optimal action-value function q_* directly, regardless of the policy that is being followed. This simplifies the analysis of the algorithm and allows for early convergence proofs. Although the policy still determines which state-action pairs are updated, it is only necessary for all pairs to continue to be updated for correct convergence [22]. This requirement is minimal in the sense that any method that is guaranteed to find optimal behavior in the general case must fulfill it.

Q-learning is known to have strong convergence properties, ensuring convergence to an optimal action-value function in a stationary environment with discrete domains. For this to occur, two conditions must be met. Firstly, the learning rate must gradually approach zero, and secondly, all state-action pairs must be visited for an infinite number of episodes. To satisfy the latter condition and avoid the possibility of converging to local optima by always selecting the action with the highest immediate reward, an $epsilon-greedy$ policy can be adopted. This policy involves selecting the action with the highest estimated state action-value with probability $(1-\epsilon)$ and selecting a random action with probability ϵ. By choosing a random action, the agent is encouraged to explore and potentially find more optimal actions, thereby refining its estimate of the action-value function. However, visiting all state-action pairs infinite times is not feasible. Hence the convergence of the Q-learning is usually assessed observing the stabilization of the rewards obtained. Further explanation of this method is given in the section 3.4.5.

3 | Application to Optimal Truss Design Synthesis

3.1. Design Problem

The design goal is to create a truss structure that is stable and can withstand external forces with minimal displacement. To achieve this, the design must be optimized to minimize the compliance of the structure while staying within a specified volume constraint.

This is a common problem in truss design optimization, and there are established methods and algorithms that can be used to solve it, in particular topology optimization algorithms [29]. By applying mathematical modeling and simulation techniques, designers can create and evaluate different truss configurations to find the one that best meets the stability and volume constraints. In this study we will be using the Q-learning Algorithm and model the process as an MDP to obtain the optimal design formulation.

The problem can be formulated as,

$$\begin{aligned} &minimize \quad u \\ &subject\ to \sum_{i=1}^{n_m} A_i L_i \leq V_{cnst} \qquad (3.1) \\ & \qquad (3.2) \end{aligned}$$

where u is the displacement of the truss configuration at a specified node and direction. A_i represents the cross-sectional areas assigned to each of the n_m truss elements, However, in this work, the same cross sectional area will be assigned to each element, thus $A_i = A$. The volume constraint is denoted by V_{cnst}.

A positive reward is obtained by diminishing the displacement of the target node in the modified structure with respect to the initial configuration. The displacement of each truss configuration is evaluated using linear finite element analysis (FEA). Before FEA, each truss configuration's stability is checked by assessing the conditioning of the stiffness matrix of the system. The result of the FEA is the displacement at a specified node, which is used to determine the immediate rewards.

3.2. Optimal Design Synthesis Through Q-learning

We now proceed to implement the process of design synthesis as an MDP using Q-learning. An MDP is a 4 tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, which in the design synthesis context, at any time t, a given design configuration is represented as a state s_t, and the available actions $a \in \mathcal{A}$, are the available grammar rules that can be applied to the given configuration to transform it into a new configuration and hence a new state, s_{t+1}. The reward R_{t+1} is the improvement in a certain design criteria in the new configuration. And P being the transition probabilities. In the following sections we will look at steps involved in the algorithm.

3.2.1. Initialization

The process starts with an initial state, s_0, which is the initial configuration or guess, having an initial volume, v_0 and sparsely connecting a nodal domain. The nodal domain is described by a Cartesian co-ordinate system. The nodes being denoted by $n_i(x, y)$.The initial configuration is then altered sequentially by discrete actions $a \in \mathcal{A}$ chosen by the agent. Fig. 3.1 depicts an example nodal domain along with an initial configuration.

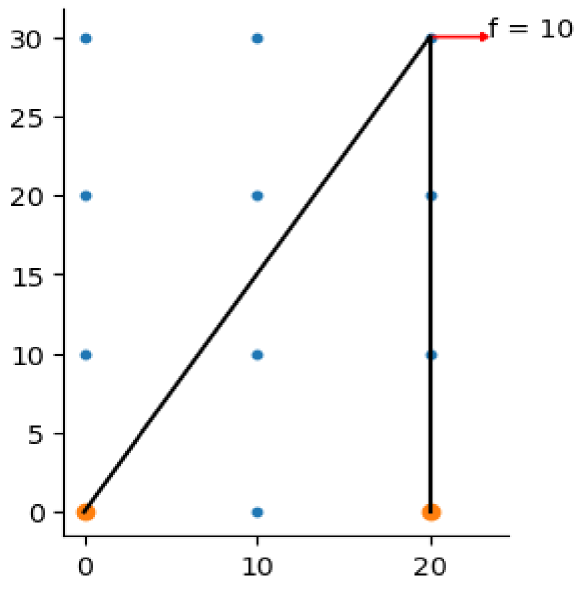


Figure 3.1: Example of a nodal domain with an initial configuration

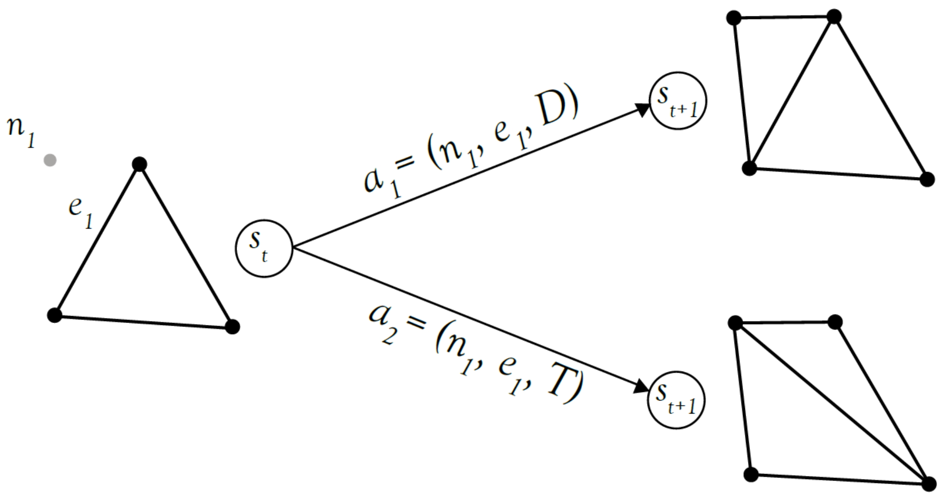


Figure 3.2: Example of a configuration altered by two actions, a1 and a2, and the associated new states. Operators D and T on element 1, e1, and node 1, n1, generate two new configurations, adapted from [47].

3.2.2. Action Space

The set of actions in this study is modelled after topological grammar rules. The set of actions that can alter the initial configuration is adapted from [46], and are represented as a decision tree.

Given a state s, at time t, the agent is presented with a set of actions based on two operators, denoted by **T** and **D**. "The D operator creates a new node and connects it to both the endpoints of a given link, essentially creating a rigid triangular component. The T operator replaces a given link with two links that pass through a newly created node." [46]

The fig 3.2, depicts the alteration of a configuration in state s, by two actions consisting of operators T and D leading to formation of new states. When drawing these operations in a schematic manner, the resulting overview resembles a tree as shown in fig 3.3

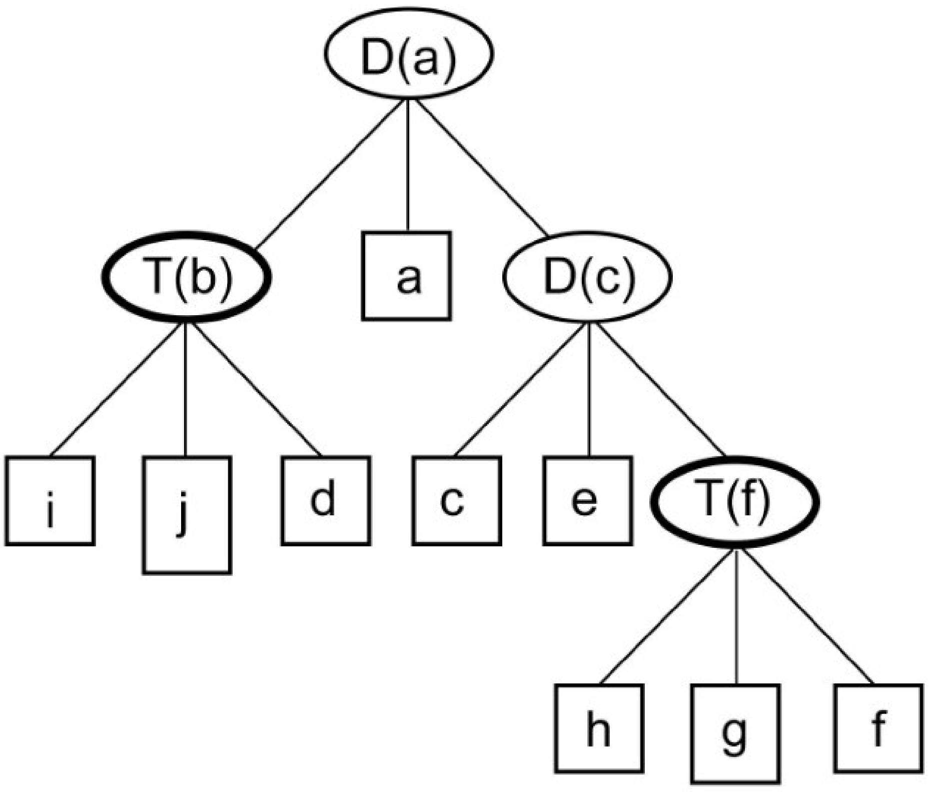


Figure 3.3: Example of a tree-like flow chart, edited from [46].

As described in [47], an action a consists of three components,

1. The selection of an inactive node
2. Selection of an eligible element for that node, and
3. The choice of of an operator (T or D) based on the location of the node relative to the selected element.

Based on this an action set is defined as in [47]:

$$\mathcal{A}(s_t) = \{(n_i, e_j, o_k) | n_i \in N_{avail}(s_t), e_j \in E_{allow}(s_t, n_i), o_k \in O_{legal}(n_i, e_j)\} \tag{3.3}$$

for which an agent is required to select a node n_i, an element e_j, and an operator o_k based on the given configuration at s_t. The selection process is limited to a set of available nodes N_{avail}, eligible elements E_{allow}, and legal operators O_{legal} that are determined by the location of an inactive node in the domain.

Rules for Operator selection

The selection of an Operator for a selected inactive node and a chosen element is based on their relative position and as such can be given as follows

1. If the selected inactive node passes through the chosen element, then only a T operator can be applied to that element.
2. If a selected inactive node passes through any of the other elements, then no operator can be applied to that element.
3. For any free node selected, both operator T and D can be applied to that element.
4. Elements will be linked to the chosen node and other active nodes, as long as they don't cross paths with any previously existing elements.
5. If any chosen node is "inside" the configuration, then its possible to connect it with a single D operator to any of the immediately surrounding active node.

3.2.3. Termination

Having defined the action space, we can now alter the given initial configuration sequentially by discrete actions until we reach a terminal state, s_T. The definition of a terminal state is problem specific and can be based on any design constraint. Generally, the algorithm is terminated when the number of inactive nodes becomes zero and/or any design criteria is violated. The run from the initialization to the termination is termed as one episode. The Q-learning algorithm can consist of multiple episodes. The exact number of episodes required for the Q-learning algorithm to converge and achieve optimal performance can vary depending on the specific problem being addressed. The initial states, transition states, and actions outlined can be used to optimize planar(or possibly spatial) trusses in general. However, the rewards, transition probabilities, and terminal states will differ depending on the design objective and constraints.

3.3. Linear Finite Element Analysis

The material behavior of the truss structure is assumed to be linear-elastic, and a linear FEA is used to determine the displacement of the structure.

3.4. Learning Environment

The problem of optimal truss design synthesis is formulated as an episodic, finite MDP with a discount factor, $\gamma = 1$, and is solved using a tabular Q-learning algorithm. The value of the discount factor depends on the fact that each modification matters for the final design. The outline of the tabular Q-learning algorithm taken from [47] is given in Algorithm 3.1. The terminal states for the given application include,

1. A state with no inactive nodes
2. An unstable state
3. A state exceeding the volume constraint

Algorithm 3.1 Tabular Q-learning for optimal truss topology synthesis, adapted from [47]

Data: Starting configuration s_0, discount factor γ, exploration probability ϵ, learning rate α, number of episodes N

```
1:  for $episodes = 1 : N$ do
2:      Initialize with the starting configuration, $s_0$
3:      repeat
4:          Choose $a_t$ from $Q(s_t, a)$ using $\epsilon$ - greedy policy
5:          Execute action $a_t$, transition to state $s_{t+1}$.
6:          if $s_{t+1}$ is a continuing state then
7:              compute reward $R_{t+1}$ through FEA
8:              update Q-table using Eq. 2.20
9:          else if $s_{t+1}$ is a terminal state then
10:             terminate
11:         end if
12:     until $volume \leq V_{cnst}$ and $N_{inactivenodes} > 0$
13: end for
```

3.4.1. Q-Table

The Q-learning algorithm employs a tabular representation termed Q-table, that stores the Q value for each state-action pair. The dimensions of the Q-table are dependent upon the number of states and corresponding actions in the given problem domain. Specifically, the rows of the Q-table signify the states that the agent can encounter, while the columns indicate the possible actions that the agent can execute in these states.

It is worth mentioning that the Q-table is initially set to zero, and the Q-values are iteratively updated based on the experiences of the agent using Eq. 2.20. The learning process aims to converge to optimal Q-values, which furnish the agent with the most optimal policy to accomplish its goals. Moreover, it should be noted that the number of states and corresponding actions are indeterminate a priori, and hence the size of the Q-table dynamically expands as more states and actions are included.

3.4.2. Reward Function

The reward function for the design synthesis task is set up to give the agent immediate feedback on the effect of its actions. Specifically, the reward is based on how the altered configuration performs in comparison to the initial configuration. If the displacement of the target node in s_{t+1} is smaller than the displacement in the initial configuration, s_t the agent gets a positive reward equal to the difference. On the other hand, if the displacement in the altered configuration s_{t+1} is larger, the agent gets a negative reward. The goal of the agent is to maximize the cumulative reward over time.

3.4.3. Transition

The process of optimal truss design synthesis involves utilizing the actions discussed in Section 3.2.2. These actions are used to alter a given truss configuration in a particular state s_t to generate new truss designs. For instance, in [47], two actions a_1 and a_2 were employed, as depicted in Fig.3.4 Action $a_1 = (n_1, e_1, D)$ is executed by activating node n_1 and creating two new elements that connect to the truss in state s_t without removing e_1. On the other hand, action $a_2 = (n_1, e_1, T)$ is executed by activating node n_1 and generating new elements after removing element e_1. It's important to note that the number of available actions for any given state is dependent on that particular state. The immediate reward received by the agent is then determined by comparing the displacement of the altered truss configuration in the next state s_{t+1} to that of the current state s_t.

3.4.4. Hyper-parameters

The agent's action selection is based on an ϵ-greedy policy that balances exploration and exploitation. The exploration rate, ϵ_t, which determines the likelihood of selecting a random action, is gradually reduced over a specified number of episodes. As given in [47], The reduction in ϵ_t is controlled by a cosine function, $\epsilon_t = C_\epsilon \cos(B_\epsilon m)$, with parameters C_ϵ and B_ϵ that determine the amplitude and period of the decay, respectively and m denotes the episode number. In the case studies presented, the learning rate, α_t, is also decayed using the same cosine function, but with unique parameter values. Both

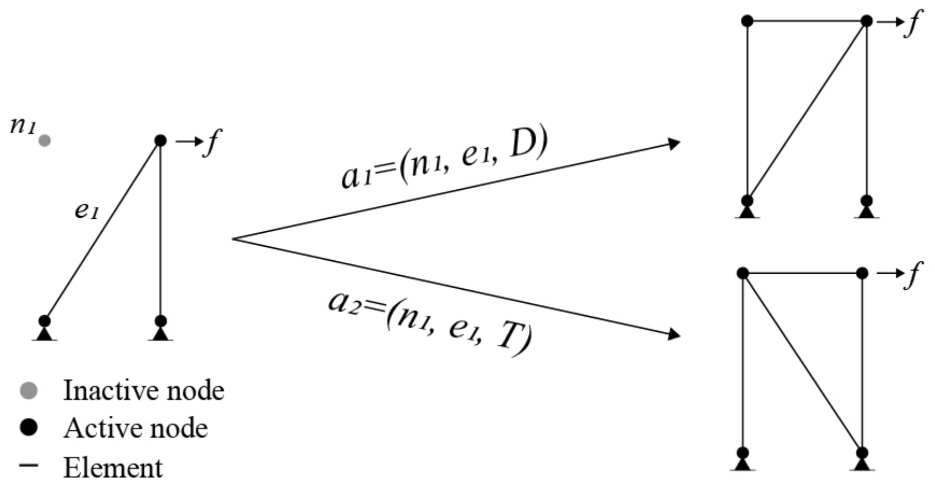


Figure 3.4: Example of a truss configuration altered by two actions, a1 and a2, and the associated new states. Specifically D and T operations on element 1, e_1, and node 1, n_1, generate two new truss configurations, adapted from [47].

the learning and exploration rates are initialized to $C_\alpha = 1$ and $C_\epsilon = 0.9$ and decayed to 0.1 and 0.01, respectively, over a set of episodes that constitute an experiment. The parameters are then held constant at their respective decayed values for the remaining episodes.

3.4.5. Convergence

The convergence of Q-learning algorithm is guaranteed under the condition that all state-action pairs are visited an infinite number of times and the learning rate, α_t, satisfies the Robbins-Monro conditions, which requires $\sum_{t=1}^{\infty} \alpha_t = \infty$ and $\sum_{t=1}^{\infty} \alpha_t^2 < \infty$. [22]

In practice, however, it is not always feasible to visit all state-action pairs an infinite number of times. Therefore, convergence of the Q-learning algorithm is typically assessed by monitoring the average reward obtained by the agent over a large number of episodes. If the average reward stabilizes or converges to a maximum value, it is assumed that the Q-learning algorithm has converged. Additionally, convergence can also be evaluated by monitoring the Q-values over time and checking if they converge to their true values. However, this is difficult to achieve in practice due to the large state and action spaces. Therefore, it is common practice to stop the Q-learning algorithm once the average reward has stabilized or after a fixed number of iterations.

In this study, the convergence is determined by letting the algorithm run for a large number of episodes and then finding the stabilization point. Indeed the number of episodes selected is problem specific and depends on the number of available actions.

Having understood the application of the Q-learning algorithm, in the next chapter we take a look at few case studies solved using this approach.

4 | Results

This chapter presents few case studies to demonstrate the effectiveness of the discussed MDP-RL framework for optimizing planar trusses. The first three case studies are the reproduction of the results given in [47]. They are needed to validate the in house python code written during this thesis project. Each one begins with a truss starting configuration, an externally applied force or forces, and boundary conditions. The structural parameter values used in the linear Finite Element Analysis (FEA) are made non dimensional. For all examples the linear FEA assumes an elastic modulus of $E_0 = 10^4$ and an area of $A = 1$. As fulfilling the Q table is in part a random process, due to the adopted exploration strategy, the training has been repeated 50 times. The average results from these experiments were used to assess the algorithm's performance.

4.1. Case Study 1: 4x3 Nodal Domain with a Single Force

In the first case study we will consider a 4x3 Nodal Domain with a single Concentrated External Force. The initial state s_0 consists of the starting configuration described in the Fig. 4.1. The starting configuration includes two pin supports at the bottom left and right nodes, denoted in orange and an external force of $f = 10$ is applied horizontally to the upper right node. A volume constraint of $V_{cnst} = 160$ is also specified. The starting configuration has a volume of $V_0 = 66.1$ and a displacement of $u_0 = 0.1847$ at the upper right node (the target node) under the given external force, as determined by a linear FEA analysis.

The learning rate and exploration rate are initialized and decayed for each experiment, as described previously in Sec. 3.4.4. The learning rate is decayed using $B_\alpha = 9.7 * 10^{-4}$, and the exploration rate is decayed using $B_\epsilon = 1.6 * 10^{-3}$. The balance between exploration and exploitation is task-dependent, and the values of these parameters were determined based on trial and error. The case study parameters are reported in Table 4.1.

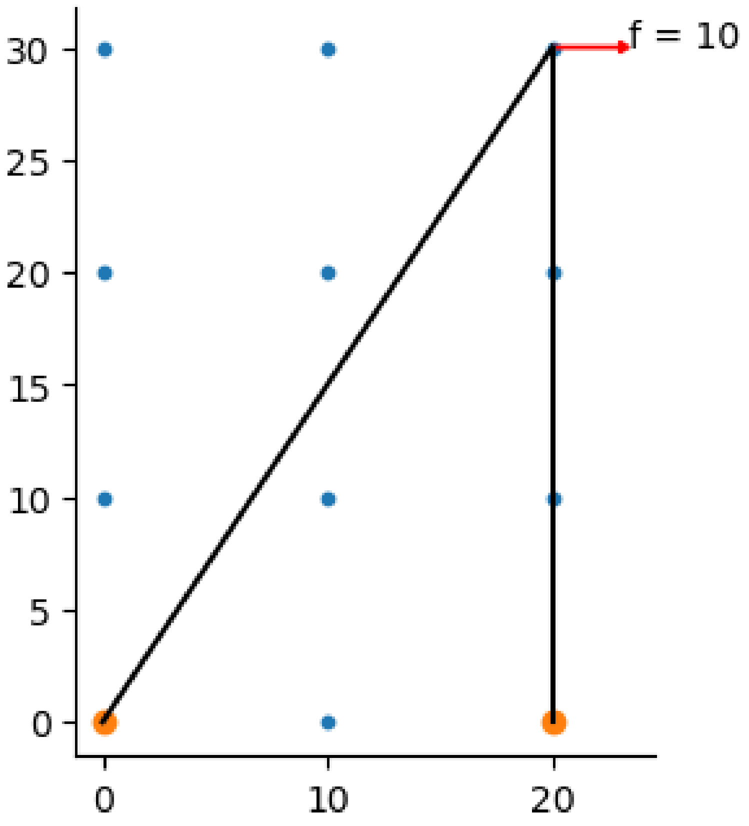
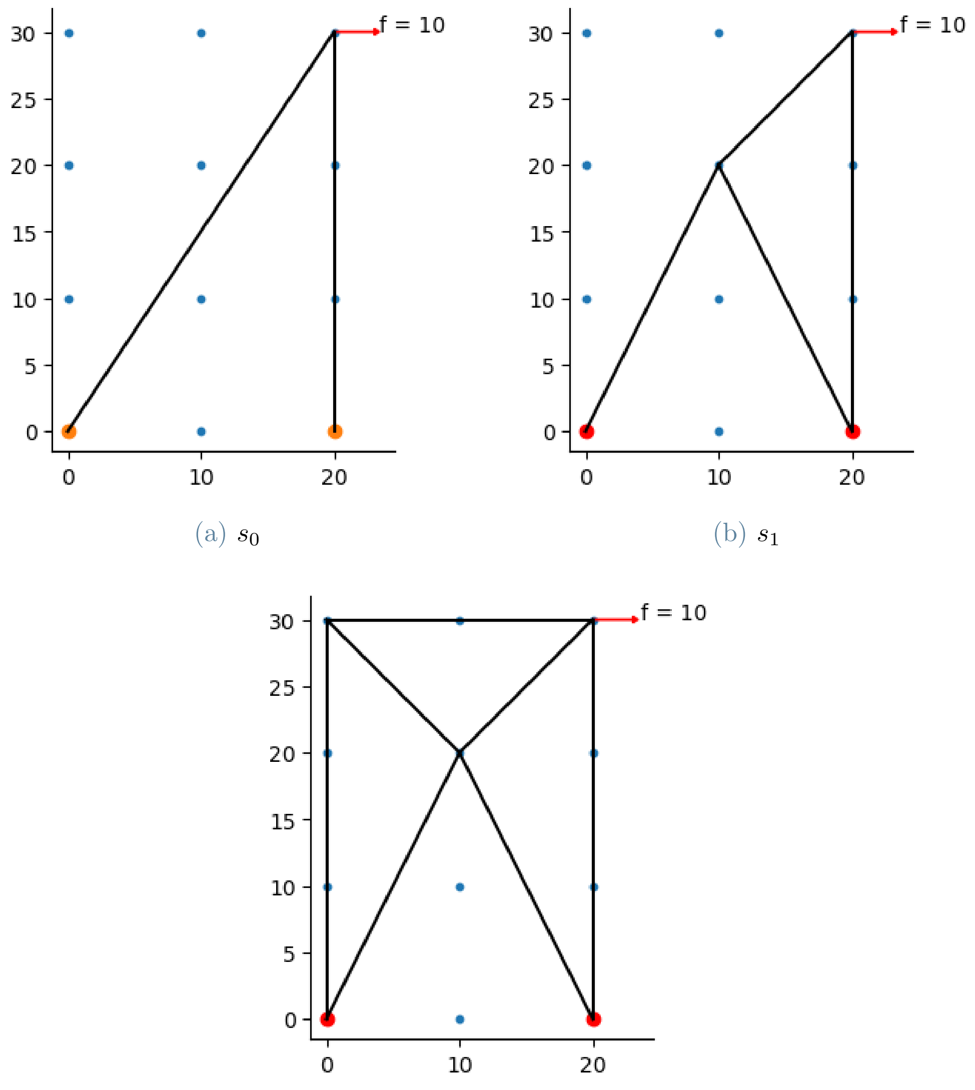


Figure 4.1: Case study 1, initial configuration, applied node and constrains. The target node is the where the force is applied.

	Values
f	10
V_{cnst}	160
u_0	0.1847
V_0	66.1
B_α	$9.7 * 10^{-4}$
B_ϵ	$1.6 * 10^{-3}$

Table 4.1: Attributes and Parameters for Case Study 1

The sequence of altered configurations that the agent makes from the learned optimal policy is illustrated in Fig. 4.2. The truss configuration shown for s_2 is the optimal truss configuration determined by the agent, for all experiments. The optimal truss has a volume, V_{opt}, of 153.0 and under the specified external force a displacement, u_{opt}, of 0.0895 at the upper right node in the domain.



(a) s_0 (b) s_1

(c) s_2

Figure 4.2: Optimal Design Sequence for Case Study 1

	Values
u_{opt}	0.0895
V_{opt}	153.0

Table 4.2: Optimal Attributes and Parameters for Case Study 1

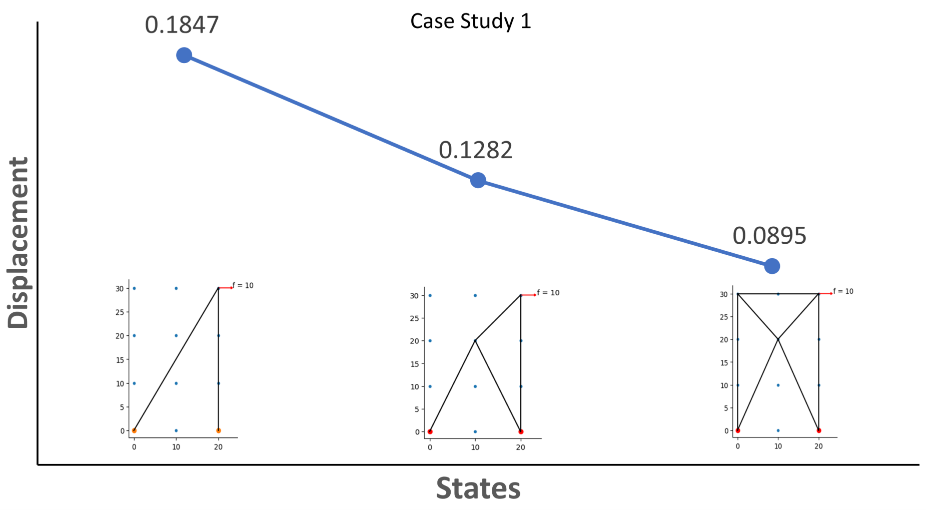


Figure 4.3: Case study 1, variation of displacement with optimal design sequence

The agent achieved a maximum reward of 0.0952, unknown at the beginning. However, it eventually learns the optimal policy that results in the best possible truss configuration with minimal displacement. The agent's average performance over the first 1000 episodes is shown in Fig. 4.4, and it converged to the maximum reward. This means that the agent has successfully learned an optimal policy. On average, the agent evaluated 183 truss configurations per experiment to learn this policy. The variation in the displacement in each state of the optimal design path is shown in Fig. 4.3.

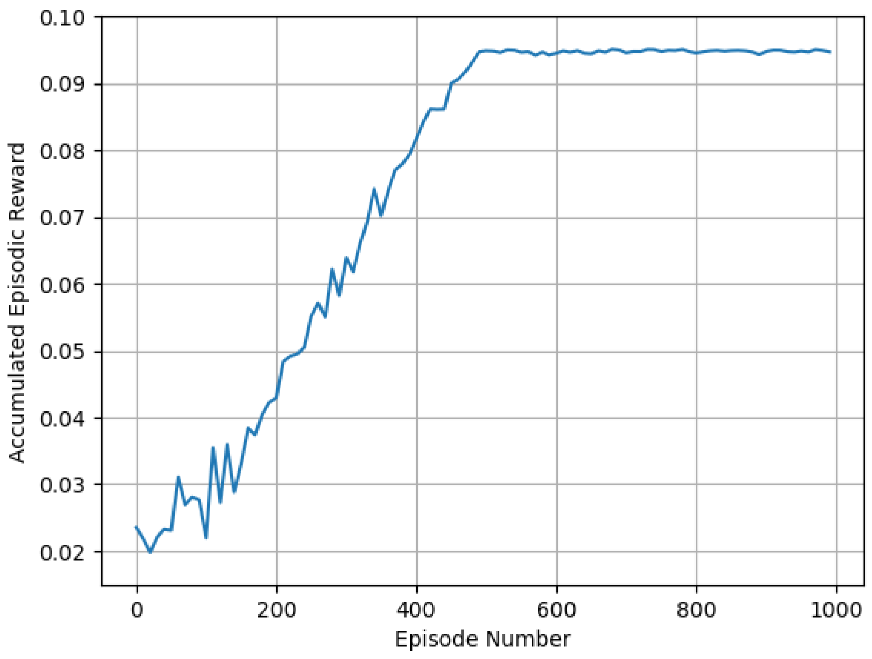


Figure 4.4: Average performance of the agent at every ten episodes for Case Study 1.

4.2. Case Study 2: 5x3 Nodal Domain with Multiple loading conditions

In the second case study we will consider a 5x3 Nodal Domain with two concentrated external forces. The initial state s_0 consisting of the nodal domain and starting configuration is described in the Fig. 4.5. The attributes and parameters are given in the Table 4.4. The learning rate and exploration rate are initialized and decayed for each experiment, as described previously.

In all the experiments, the best policy resulted in the optimal configuration of the truss, which was represented in the fourth state, s_4. This optimal configuration met the volume constraint and had a volume, V_{opt} of 238.8, with a displacement, u_{opt} of 0.1859 at the upper right edge node in the given domain under the specified external forces. The maximum reward that the agent could achieve was unknown initially, but it turned out to be 0.2377. The agent's average accumulated reward over episodes kept improving until it converged to the maximum value, indicating that the agent learned the optimal policy. Fig. 4.8 shows the average performance of the agent during the first 3000 episodes. On average, the agent evaluated 1334 truss configurations per experiment to learn the best policy.

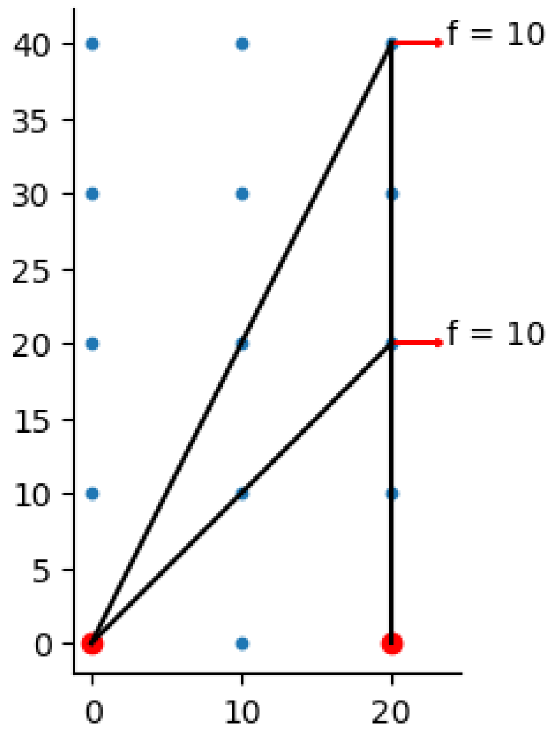
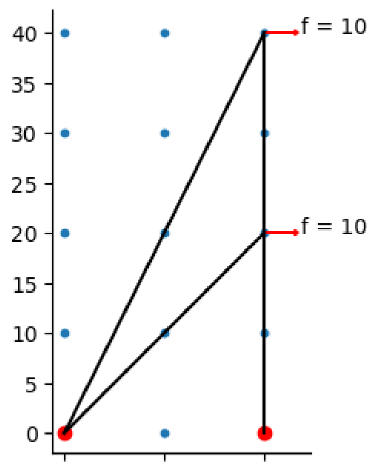
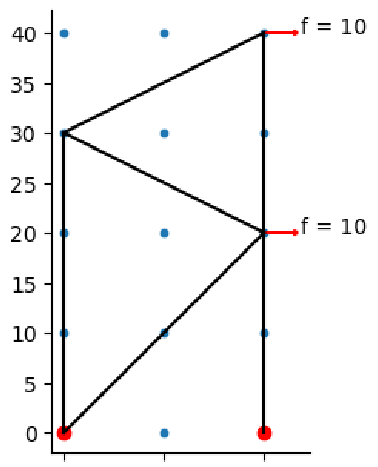


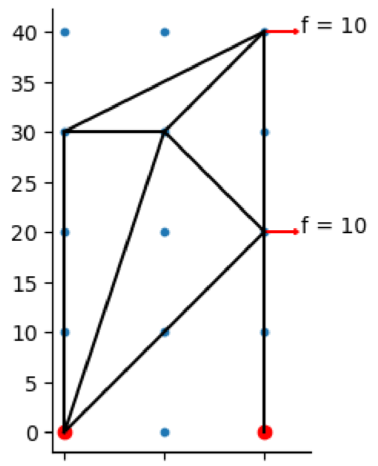
Figure 4.5: Case Study 2. initial configuration, applied node and constraints. The target node is the where the force is applied.



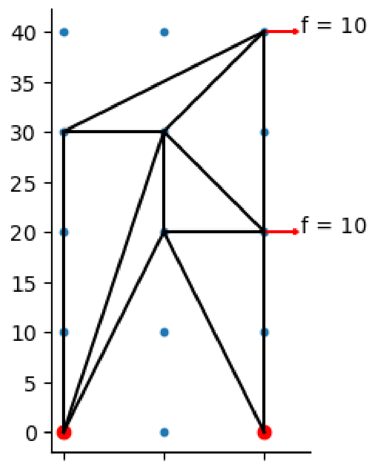
(a) s_0



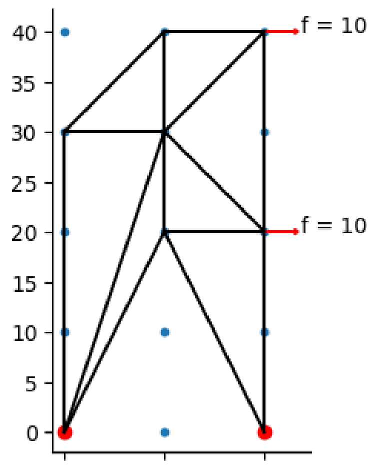
(b) s_1



(c) s_2



(d) s_3



(e) s_4

Figure 4.6: Optimal Design Sequence for Case Study 2

	Values
f	10
V_{cnst}	240
u_0	0.4236
V_0	113.0
B_α	$2.65 * 10^{-4}$
B_ϵ	$3.1 * 10^{-4}$

Table 4.3: Attributes and Parameters for Case Study 2

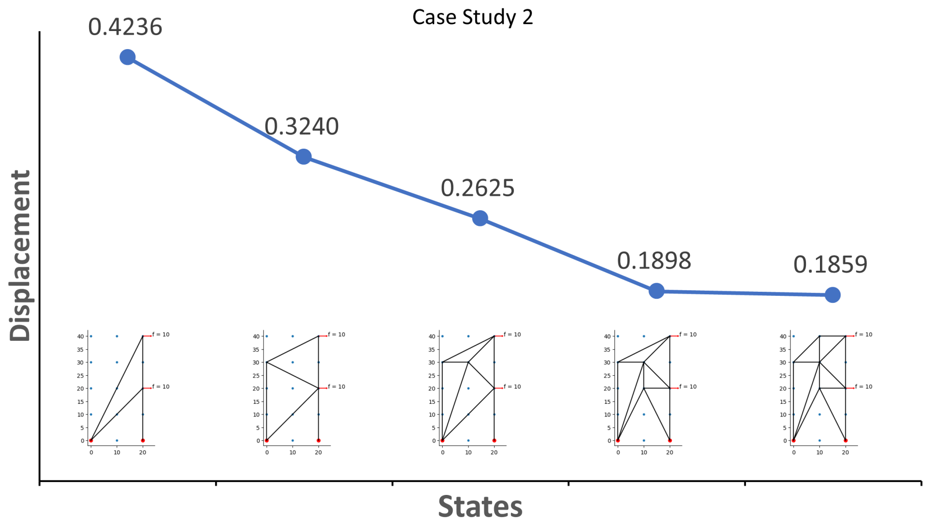


Figure 4.7: Case study 2, variation of displacement with optimal design sequence

	Values
u_{opt}	0.1859
V_{opt}	238.8

Table 4.4: Optimal Attributes and Parameters for Case Study 2

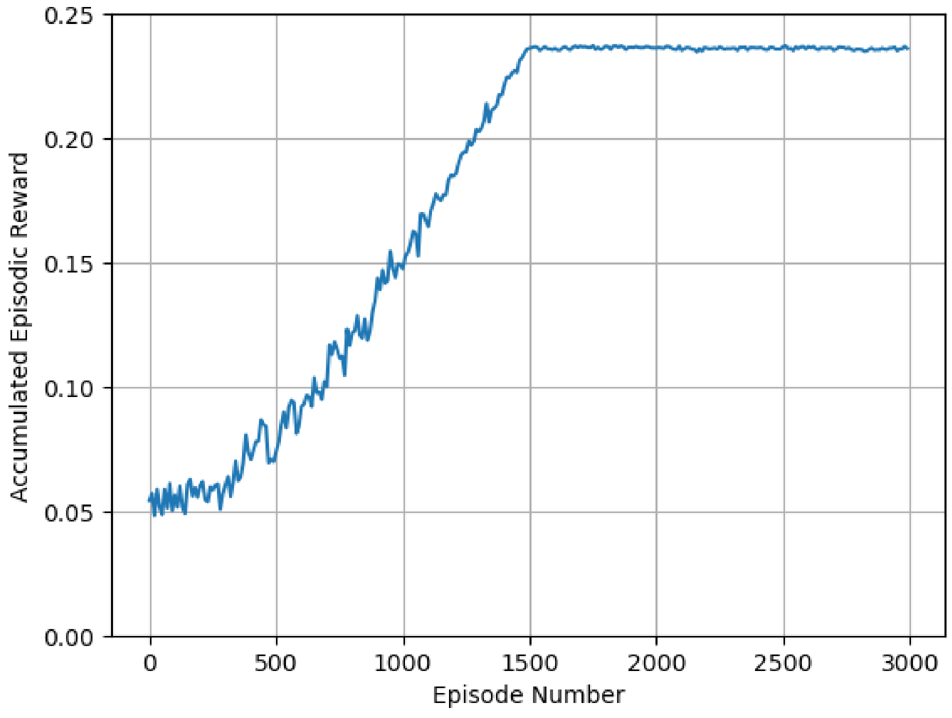


Figure 4.8: Average Episodic Reward, Case Study 2.

4.3. Case Study 3: 4x7 Nodal Domain with a Bridge Structure

In the third case study the MDP-RL synthesis framework was applied to a bridge design domain with a single external force to demonstrate its versatility. The initial state s_0 consisting of the nodal domain and starting configuration is described in the Fig. 4.9. Table 4.6 shows the attributes and parameters of this case study. The sequence of altered configurations resulting from a learned optimal policy is presented in the Fig. 4.9 The optimal truss configuration for s_2 is shown in the Fig. 4.10 and has a volume, V_{opt} of 236.0. Under the specified external force, it has a displacement, u_{opt} of 0.0425 at the bottom middle node of the domain.

The maximum reward that the agent could achieve in this case study was initially unknown, but it turned out to be 0.0299. The agent's average accumulated reward improved over episodes and converged to the maximum value, as shown in Fig. 4.12. This indicates that the agent learned an optimal policy that produces the best truss configuration. On average, the agent evaluated 324 truss configurations per experiment to learn the optimal policy.

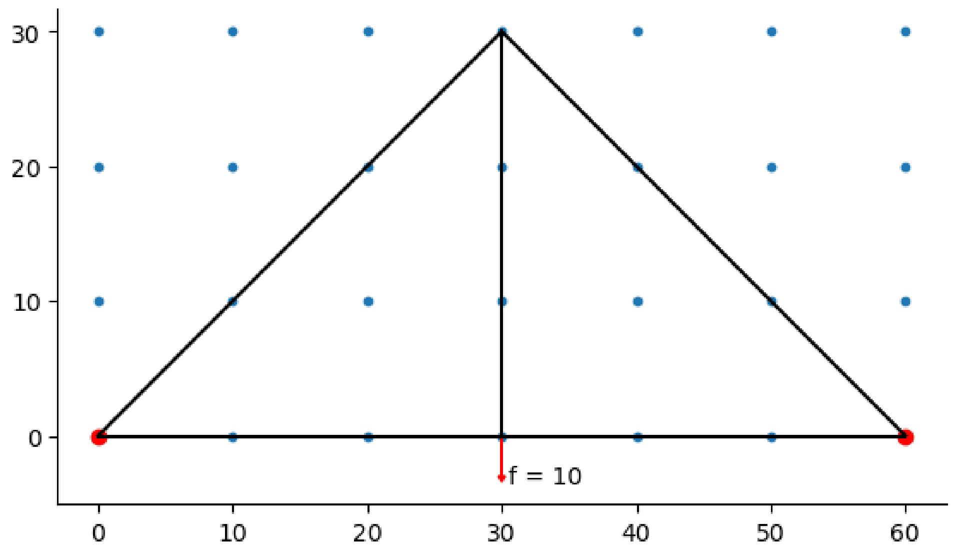
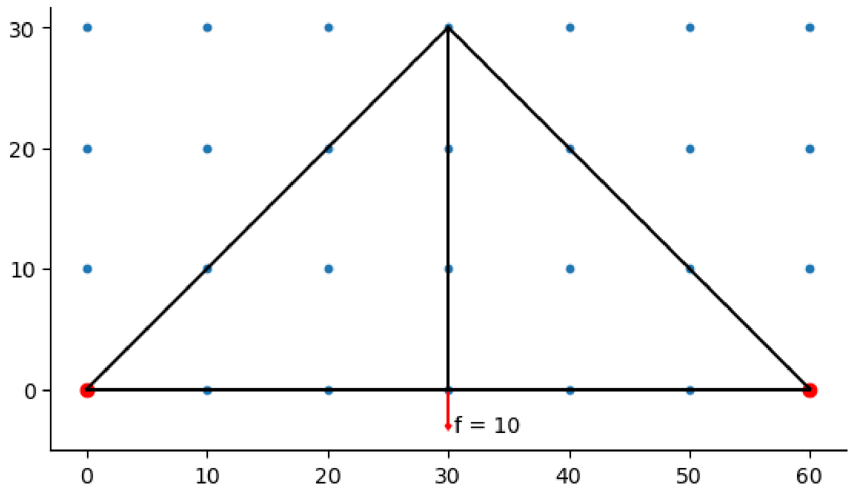


Figure 4.9: Case Study 3. initial configuration, applied node and constraints. The target node is the where the force is applied.

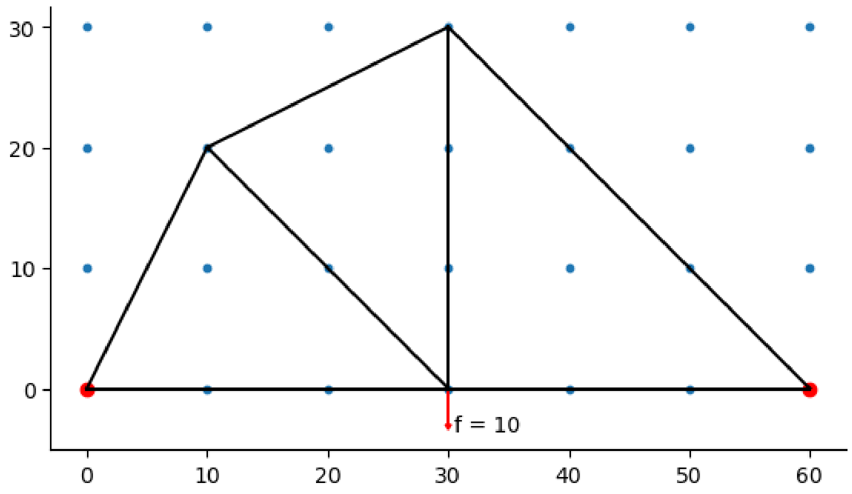
	Values
f	10
V_{cnst}	250
u_0	0.0724
V_0	174.9
B_α	$7.5 * 10^{-5}$
B_ϵ	$8.21 * 10^{-5}$

Table 4.5: Attributes and Parameters for Case Study 3

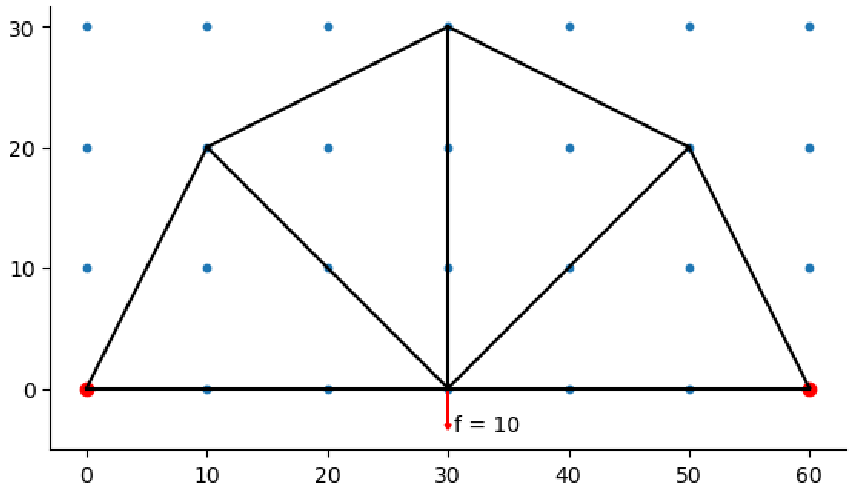
The sequence of altered configurations that the agent makes from the learned optimal policy is illustrated in Fig. 4.10. The truss configuration shown for s_2 is the optimal truss configuration determined by the agent, for all experiments. The optimal truss has a volume, V_{opt}, of 236.0 and under the specified external force a displacement, u_{opt}, of 0.0425 at the bottom middle node in the domain.



(a) s_0



(b) s_1



(c) s_2

Figure 4.10: Optimal Design Sequence for Case Study 3

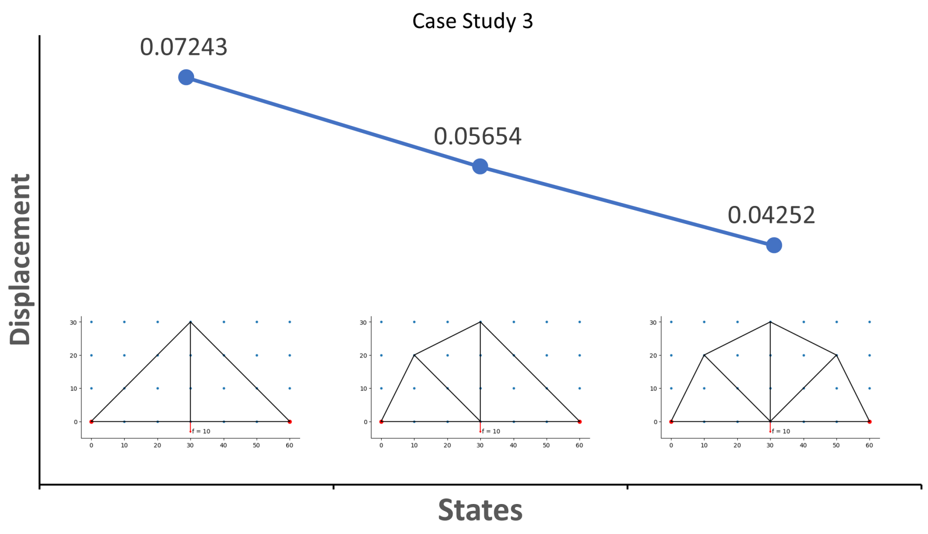


Figure 4.11: Case study 3, variation of displacement with optimal design sequence

	Values
u_{opt}	0.0425
V_{opt}	236.0

Table 4.6: Optimal Attributes and Parameters for Case Study 3

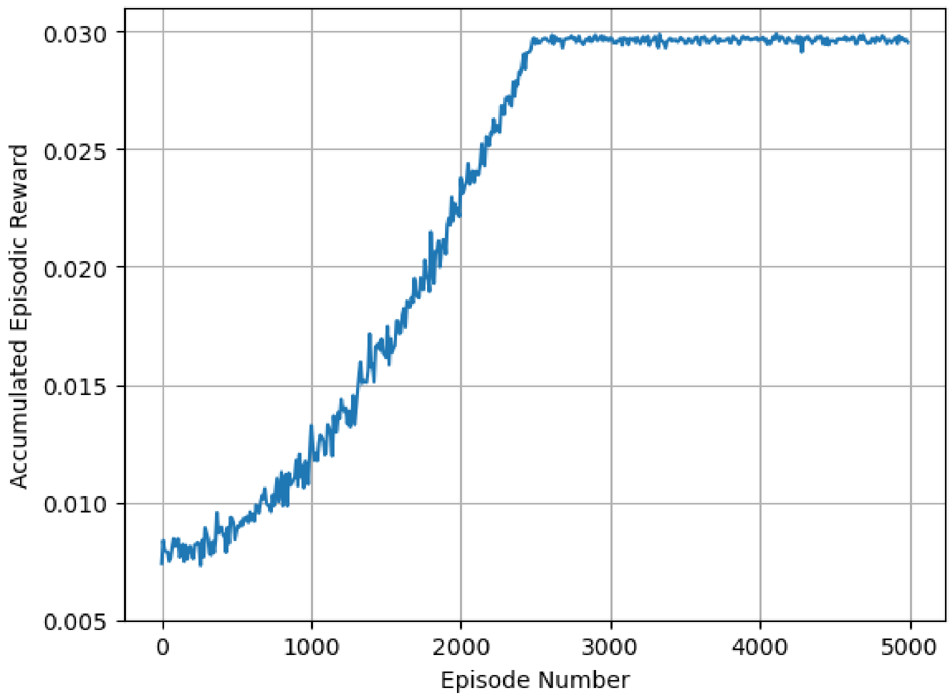


Figure 4.12: Average Episodic Reward, Case Study 3.

4.4. Comparison with Exhaustive Approach

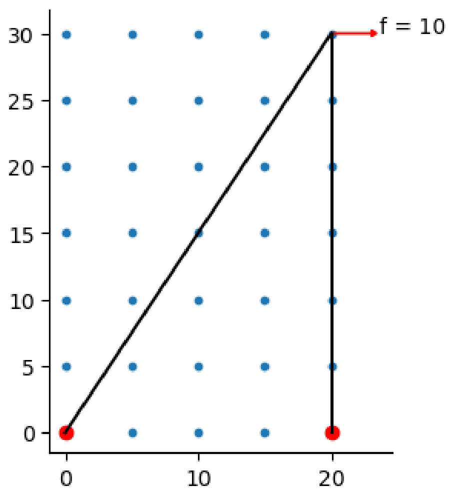
Comparing all three case studies with the results of [47], we can conclude that the in house python code is able to exactly reproduce the results. Table 4.7 provides a comparison of the results obtained from the RL approach and an exhaustive approach where all possible states are evaluated for the first three case studies. The given computational time was obtained from a system running a Intel Xeon W-2275 CPU @ 3.30GHz with 128 GB RAM. The results show that for case study 1, the RL approach achieved the optimal solution with only 183 model evaluations, compared to 1058 model evaluations required by the exhaustive approach. This translates to an 83% savings in terms of model evaluations and computational time. Similarly, for case study 2 and 3, the RL approach was able to converge to the optimal solution much faster than the exhaustive approach, resulting in savings of 89.93% and 65%, respectively. The study by Ororbia and Warn [47] has also shown that the proposed RL approach outperformed Genetic Algorithms in terms of computational efficiency, achieving over 50% savings in model evaluations and computational time for all case studies while finding the global optimal solution.

Case Study	**Approach**	**Average model evaluations**	**Average time, s**	u_{opt}	**% time savings**
1	RL	183	4.2	0.0895	83%
	Exhaustive	1058	25.2	0.0895	
2	RL	1334	48.3	0.1859	89.93%
	Exhaustive	9836	479.7	0.1859	
3	RL	324	12.8	0.0425	65%
	Exhaustive	903	36.6	0.0425	

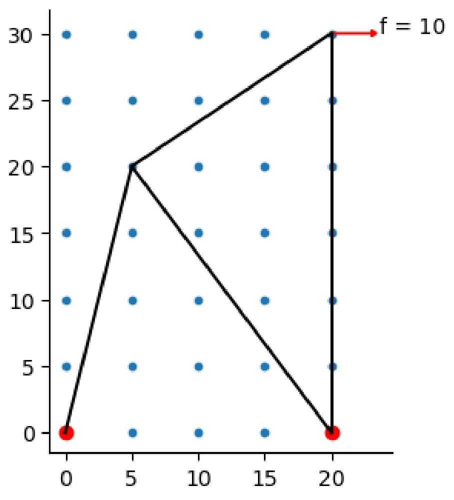
Table 4.7: Comparison of results obtained from RL synthesis method and exhaustive evaluation

4.5. Case Study 4: 7x5 Nodal Domain

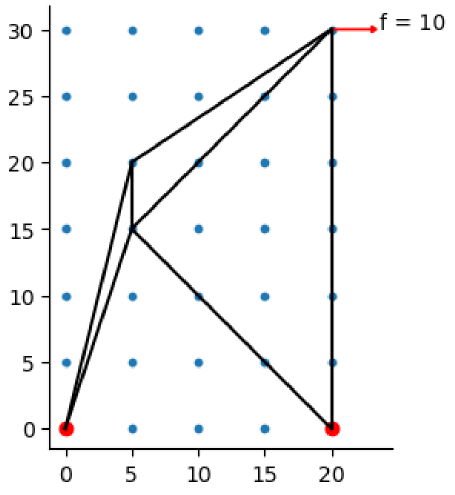
In this case study we repeat the case study 1 increasing the nodal density by using a 7x5 grid. while maintaining the same volume constraint. We run the algorithm for 50 experiments each with 2000 episodes. The initialed and the final optimized values are given in the Table. 4.8. While the sequence of modified design is given in the Fig. 4.13. we can observe that the algorithm produced a higher reward by increasing the nodal density while maintaining the same volume.



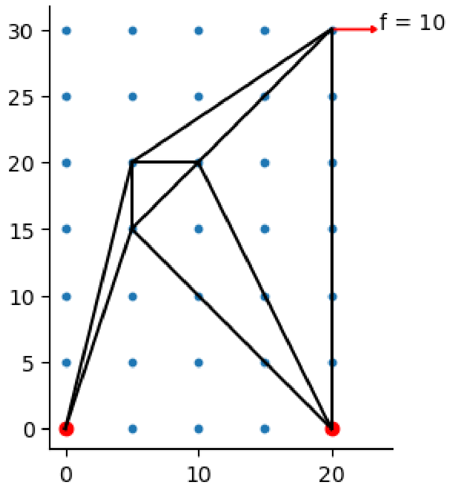
(a) s_0



(b) s_1



(c) s_2



(d) s_3

Figure 4.13: Optimal Design Sequence for Case Study 4

	Initial Values	**Optimized Values**
f	10	
V_{cnst}	160	
u	0.1847	0.0721
V	66.1	159.24
B_α	$9.8 * 10^{-4}$	
B_ϵ	$1.56 * 10^{-3}$	

Table 4.8: Attributes and Parameters for Case Study 4

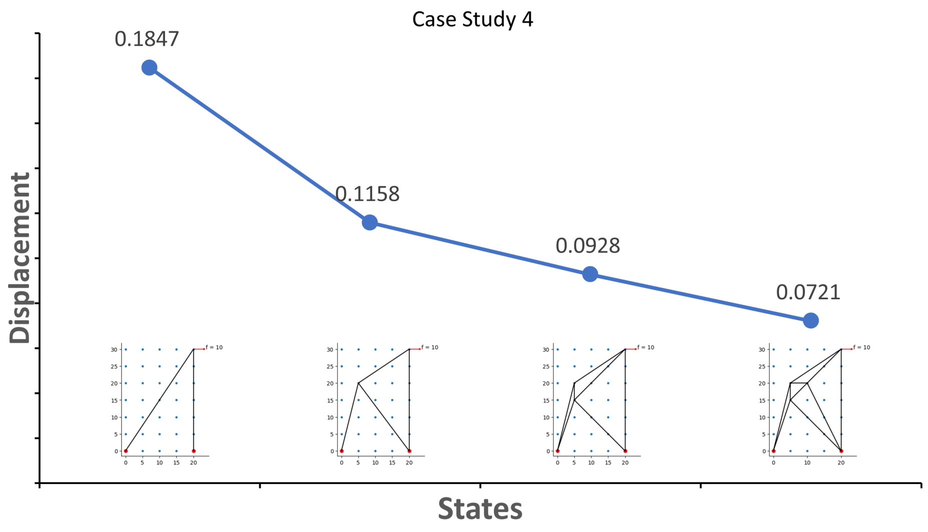


Figure 4.14: Case study 4, variation of displacement with optimal design sequence

4.6. Case Study 5: 3x9 Nodal Domain

Case Study 5 consists of a Beam like truss structure, supported by two hinges shown in red dots, while a concentrated force of $f = 10$ is applied at the centre as shown in the Fig. 4.15. The algorithm is run for 50 experiments each consisting of 5000 episodes. The results obtained show that the algorithm is able to exploit the symmetry of the structure and produces a optimal structure which is a symmetric one.

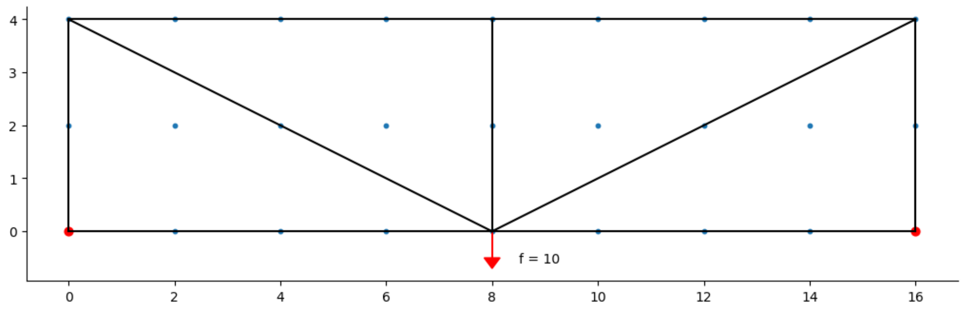
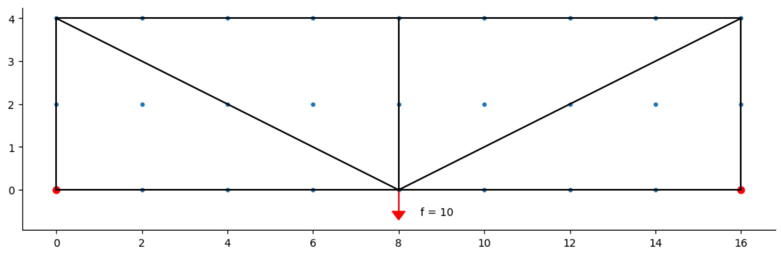


Figure 4.15: Case Study 5. initial configuration, applied node and constraints. The target node is the where the force is applied.

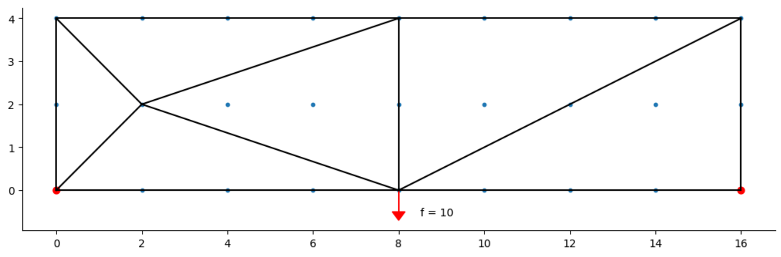
	Initial Values	**Optimized Values**
f	10	
V_{cnst}	100	
u	0.0404	0.0146
V	61.89	98.80
B_α	$3.92 * 10^{-4}$	
B_ϵ	$6.24 * 10^{-4}$	

Table 4.9: Attributes and Parameters for Case Study 5

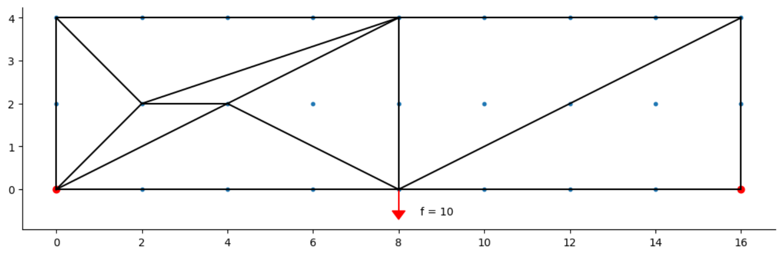
The sequence of optimization given in the Fig. 4.16 and the parameters are given in the Table. 4.9



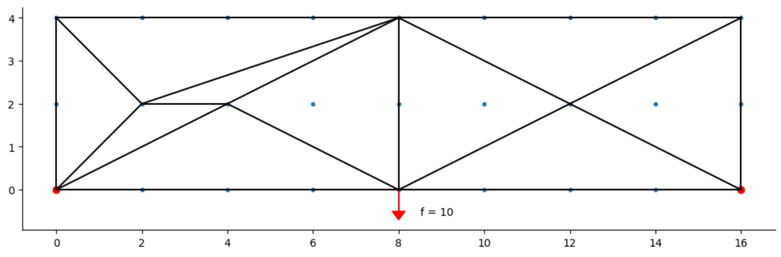
(a) s_0



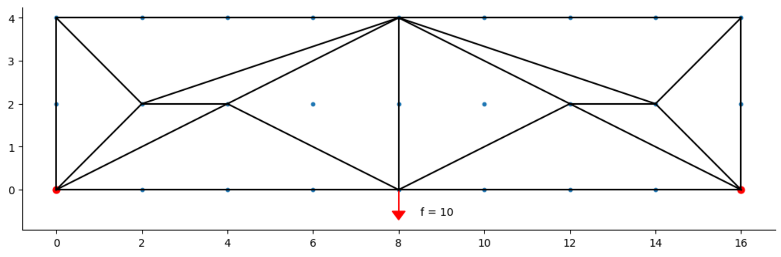
(b) s_1



(c) s_2



(d) s_3



(e) s_3

Figure 4.16: Optimal Design Sequence for Case Study 5

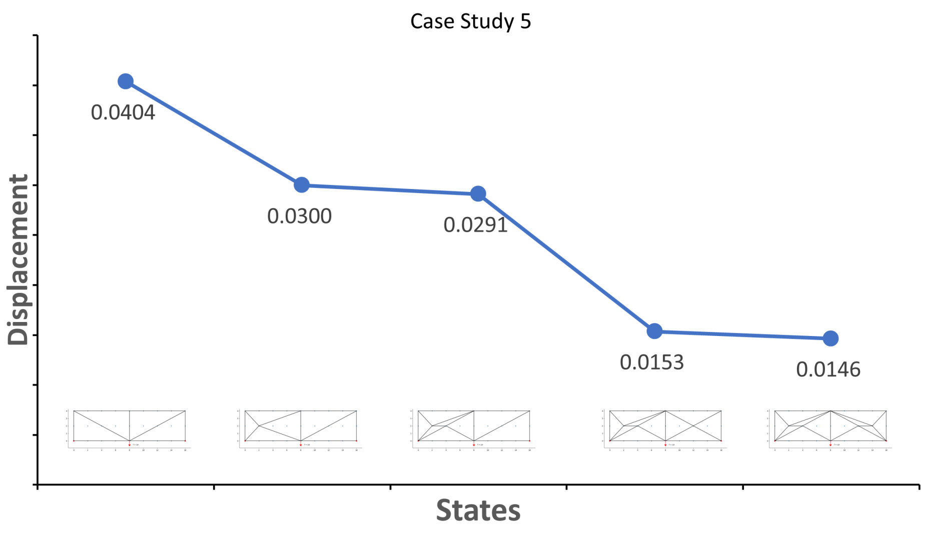


Figure 4.17: Case study 5, variation of displacement with optimal design sequence

4.7. Case Study 6: 3x9 Nodal Domain

Case Study 6 consists of a truss structure as shown in the Fig. 4.18. The structure is supported by two hinges, represented by two red dots. A concentrated force of $f = 10$ is also applied at the centre node. Similarly to the previous case study, the algorithm is run for 50 experiments of 5000 episodes each. And every time the algorithm produces the same optimal design shown in Fig. 4.19.

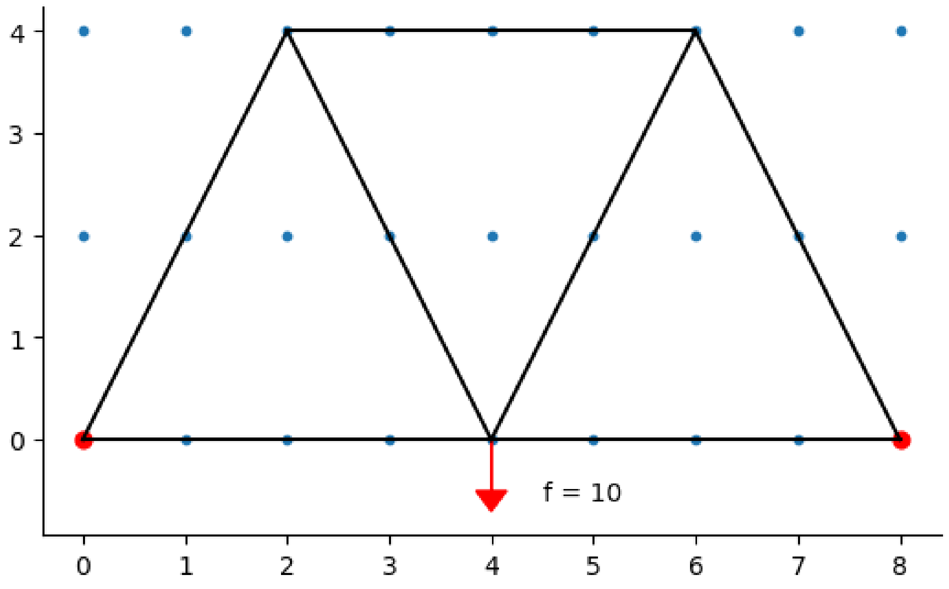
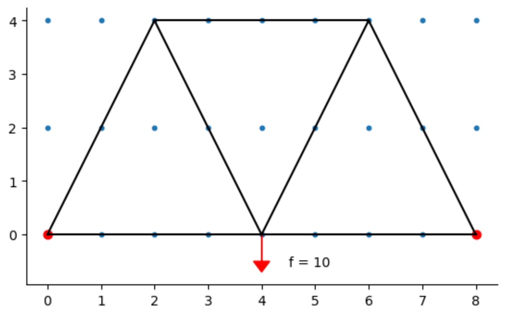
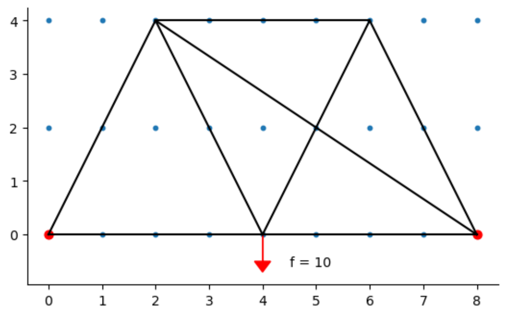


Figure 4.18: Case Study 6. initial configuration, applied node and constraints. The target node is the where the force is applied.

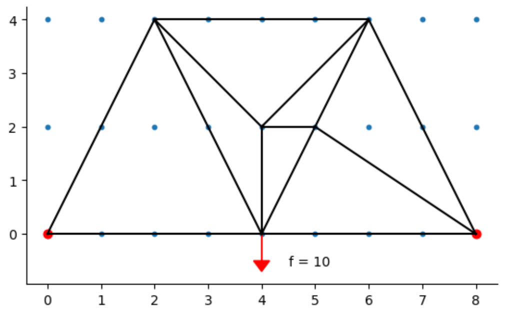
The initial and optimized parameters for the case study are given in Table. 4.10. And the variation of the displacement is presented in Fig. 4.20.



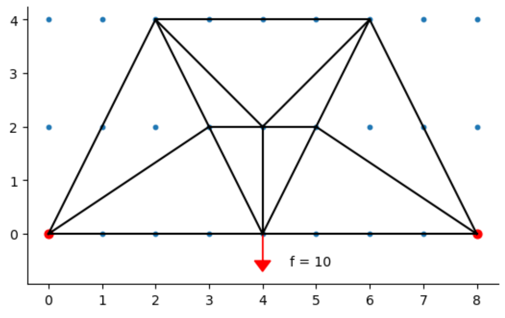
(a) s_0



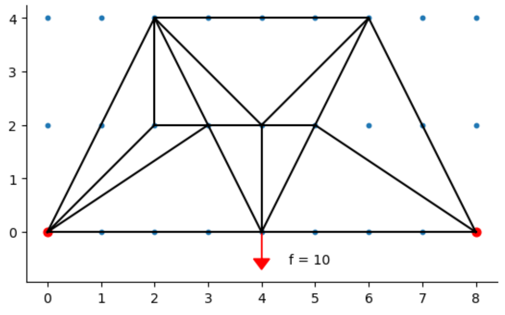
(b) s_1



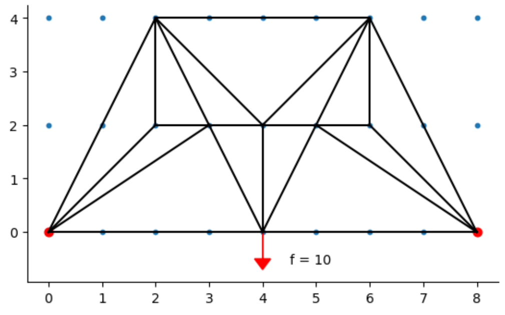
(c) s_2



(d) s_3



(e) s_3



(f) s_3

Figure 4.19: Optimal Design Sequence for Case Study 6

	Initial Values	Optimized Values
f	10	
V_{cnst}	60	
u	0.0066	0.0036
V	29.89	59.41
B_α	$3.92 * 10^{-4}$	
B_ϵ	$6.24 * 10^{-4}$	

Table 4.10: Attributes and Parameters for Case Study 6

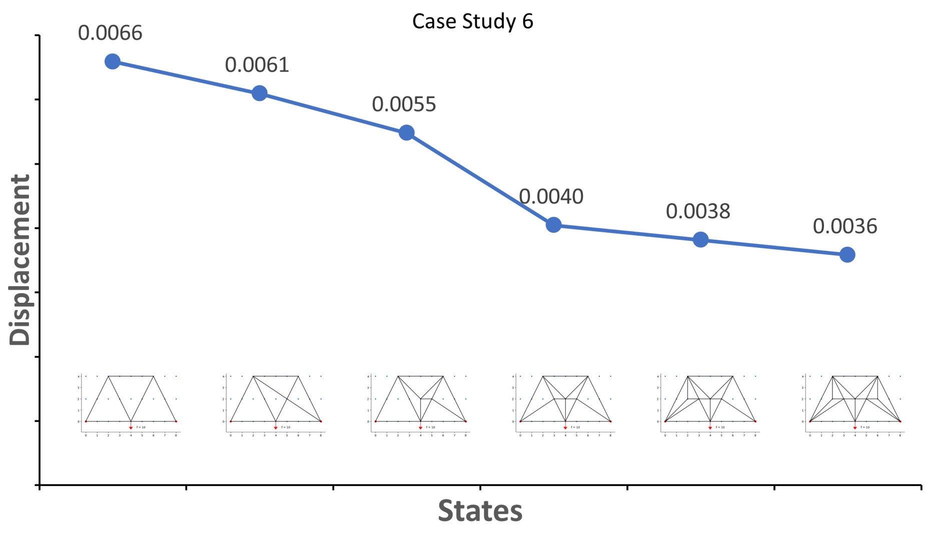


Figure 4.20: Case study 6, variation of displacement with optimal design sequence

4.8. Case Study 7: 3x9 Nodal Domain

Case Study 7 consists of a truss structure as shown in the Fig. 4.21. The structure is supported by two hinges, represented by two red dots. A concentrated force of $f = 10$ is also applied at the top node. The algorithm is run for 50 experiments of 5000 episodes each. And every time the algorithm produces the same optimal design shown in Fig. 4.22.

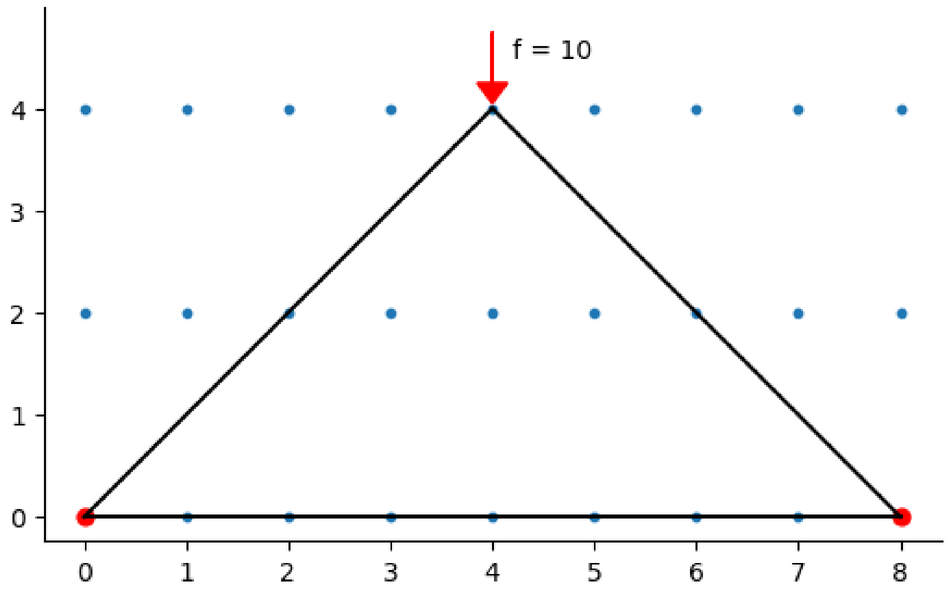


Figure 4.21: Case Study 7. initial configuration, applied node and constraints. The target node is the where the force is applied.

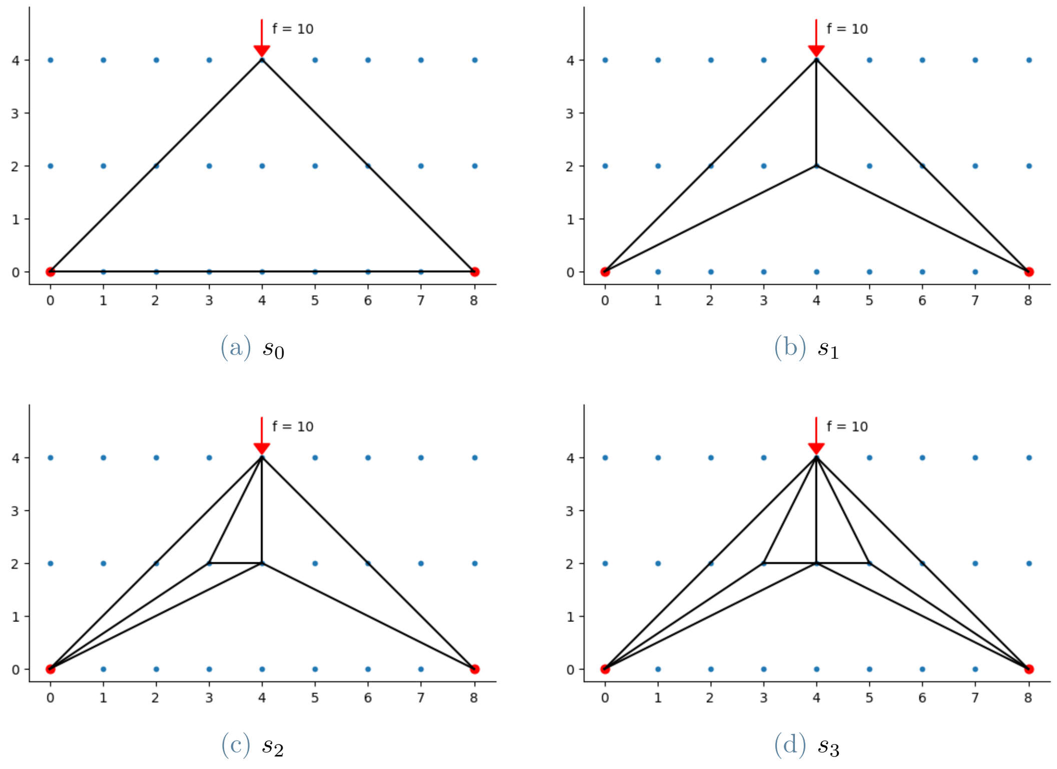


Figure 4.22: Optimal Design Sequence for Case Study 7

Similar to the previous case studies, the current case study also produces a symmetric optimal design. It can also be observed that the horizontal member, which wasn't carrying any load, gets substituted by the algorithm. The variation of the displacement is given in the Fig. 4.23. Table. 4.11 contains the parameters used for the case study.

	Initial Values	**Optimized Values**
f	10	
V_{cnst}	36	
u	0.0057	0.0026
V	19.31	35.94
B_α	$3.92 * 10^{-4}$	
B_ϵ	$6.24 * 10^{-4}$	

Table 4.11: Attributes and Parameters for Case Study 7

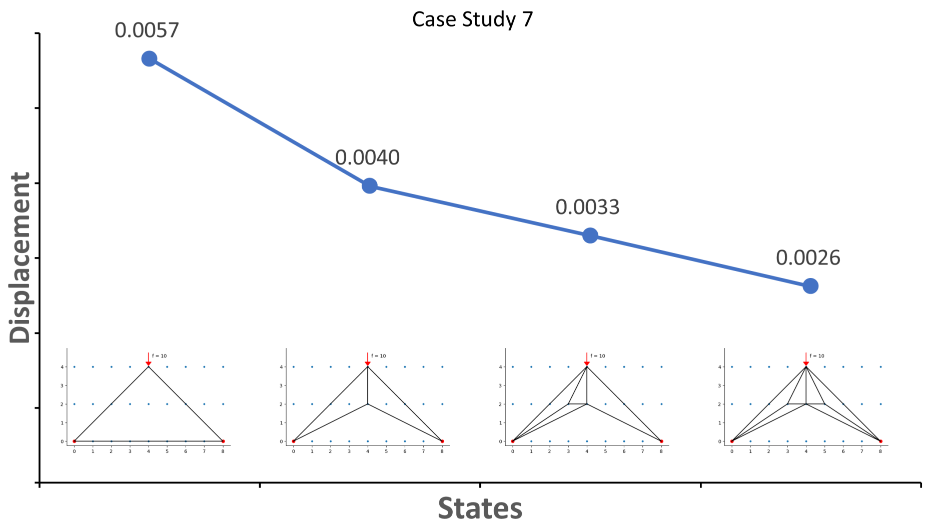


Figure 4.23: Case study 7, variation of displacement with optimal design sequence

5 | Conclusions and Future Developments

5.1. Conclusions

This work presented a framework for achieving optimal design synthesis using Q-learning, by mathematically formulating the design synthesis problem as a finite Markov decision process. According to the Markov decision formalization, new designs are achieved by taking a sequence of decisions. The framework is demonstrated by applying it to a planar truss optimization problem, where the objective is to minimize displacement under volume constraints. Seven Case Studies are presented to show the framework's versatility and ability to identify optimal solutions for problems with different characteristics. The work concludes that the MDP-RL synthesis framework can be applied to a range of design problems using general rules to modify a given configuration.

Case Studies 1,2 and 3, have shown that the proposed approach is computationally efficient compared to the Brute force approach of evaluating all the designs. The given approach has also proven to provide at least 50% of savings in computational time than the existing Genetic algorithms [47]. In Case Study 4, it was observed that increasing the nodal density of the design solution resulted in a corresponding increase in the achieved reward, while still maintaining the same volume. However, the augmented nodal density also corresponded to an increase in memory storage requirements. To accommodate for this, more sophisticated data structures may be necessary for efficiently storing the Q-table as the density of nodes increases. In Case Study 5, 6 and 7, where the starting configuration possessed some form of symmetry. The generated optimal design solutions were also found to be symmetric.

Furthermore, the extension of the proposed approach to 3D truss design necessitated additional rules beyond those utilized in the 2D case. The application of these additional rules is crucial for achieving optimal results in the 3D context.

5.2. Future Developments

The framework can be extended to truss optimization problems that involve multiple discrete values for cross-sectional areas. Other possible extensions, such as large nodal domains, three-dimensional design problems, and large volume allocations, may require the use of alternative RL algorithms like policy and function approximation. Ongoing research is currently exploring these topics.

The size of the Q-table depends on the number of possible states and actions in the environment. As the number of states and actions increase, the size of the Q-table grows exponentially, making it more challenging to store and update efficiently. This challenge can be solved by the development of alternative approaches such as function approximation and deep reinforcement learning that can handle larger and more complex state and action spaces.

Bibliography

[1] E. Antonsson and J. Cagan, *Formal Engineering Design Synthesis.* Cambridge, UK: Cambridge University Press, 2005.

[2] A. Chakrabarti, *Engineering Design Synthesis: Understanding, Approaches and Tools.* New York: Springer Science & Business Media, 2013.

[3] M. Campbell and K. Shea, "Computational design synthesis," *AI EDAM*, vol. 28, no. 3, pp. 207–208, 2014.

[4] A. Hooshmand and M. Campbell, "Truss layout design and optimization using a generative synthesis approach," *Comput. Struct.*, vol. 163, pp. 1–28, 2016.

[5] J. Cagan, M. Campbell, S. Finger, and T. Tomiyama, "A framework for computational design synthesis: Model and applications," *ASME J. Comput. Inf. Sci. Eng.*, vol. 5, no. 3, pp. 171–181, 2005.

[6] K. Tai, G. Cui, and T. Ray, "Design synthesis of path generating compliant mechanisms by evolutionary optimization of topology and shape," *ASME J. Mech. Des.*, vol. 124, no. 3, pp. 492–500, 2002.

[7] C. Königseder, K. Shea, and M. Campbell, "Comparing a graph-grammar approach to genetic algorithms for computational synthesis of pv arrays," in *CIRP Design 2012*, (Bangalore, India), pp. 105–114, Springer, March 2013.

[8] K. Shea and J. Cagan, "Innovative dome design: Applying geodesic patterns with shape annealing," *AI EDAM*, vol. 11, no. 5, pp. 379–394, 1997.

[9] Y.-C. Lin, K. Shea, A. Johnson, J. Coultate, and J. Pears, "A method and software tool for automated gearbox synthesis," in *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, vol. 49026, (San Diego, CA), pp. 111–121, August 2009.

[10] L. Puentes, J. Cagan, and C. McComb, "Heuristic-guided solution search through a two-tiered design grammar," *ASME J. Comput. Inf. Sci. Eng.*, vol. 20, no. 1, p. 011008, 2020.

[11] A. Swantner and M. Campbell, “Topological and parametric optimization of gear trains,” *Eng. Optim.*, vol. 44, no. 11, pp. 1351–1368, 2012.

[12] J. Kennedy and R. Eberhart, “Particle swarm optimization,” in *Proceedings of the IEEE International Conference on Neural Networks*, vol. 4, pp. 1942–1948, IEEE, 1995.

[13] A. Burnap, Y. Liu, Y. Pan, H. Lee, R. Gonzalez, and P. Papalambros, “Estimating and exploring the product form design space using deep generative models,” in *ASME 2016 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, (Charlotte, NC), pp. –, August 2016.

[14] M. L. Dering and C. S. Tucker, “Generative adversarial networks for increasing the veracity of big data,” in *2017 IEEE International Conference on Big Data (Big Data)*, pp. 2595–2602, IEEE, 2017.

[15] M. L. Dering and C. S. Tucker, “Implications of generative models in government,” in *AAAI Fall Symposium Series*, 2017.

[16] D. Shu, J. Cunningham, G. Stump, S. W. Miller, M. A. Yukish, T. W. Simpson, and C. S. Tucker, “3d design using generative adversarial networks and physics-based validation,” *Journal of Mechanical Design*, vol. 142, no. 7, p. 071701, 2020.

[17] Y. Yu, T.-H. Hur, J. Jung, and I.-G. Jang, “Deep learning for determining a near-optimal topological design without any iteration,” *Structural and Multidisciplinary Optimization*, vol. 59, no. 3, pp. 787–799, 2019.

[18] S. Jang, S. Yoo, and N. Kang, “Generative design by reinforcement learning: Enhancing the diversity of topology optimization designs,” *arXiv preprint arXiv:2008.07119*, 2020.

[19] H. Sun and L. Ma, “Generative design by using exploration approaches of reinforcement learning in density-based structural topology optimization,” *Designs*, vol. 4, no. 2, p. 10, 2020.

[20] K. Hayashi and M. Ohsaki, “Reinforcement learning and graph embedding for binary truss topology optimization under stress and displacement constraints,” *Frontiers in Built Environment*, vol. 6, p. 59, 2020.

[21] C. J. Watkins and P. Dayan, “Q-learning,” *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[22] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction.* MIT press, 2018.

[23] I. E. Sutherland, "Sketchpad: A man-machine graphical communication system," in *Proceedings of the Spring Joint Computer Conference*, pp. 329–346, ACM, 1963.

[24] C. L. P. Chen and D.-y. Huang, *Boundary element methods in engineering science.* Springer-Verlag, 1989.

[25] J. H. Holland, *Adaptation in natural and artificial systems.* University of Michigan Press, 1975.

[26] S. Datta and S. Sarkar, "Parametric design tools: A comparative study," *International Journal of Innovative Technology and Exploring Engineering*, vol. 7, no. 11, pp. 34–39, 2018.

[27] J. Huang and J. Li, "Machine learning in engineering design: A comprehensive review," *Journal of Mechanical Design*, vol. 143, no. 5, p. 051701, 2021.

[28] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[29] J. Nocedal and S. J. Wright, *Numerical optimization.* Springer Science & Business Media, 2006.

[30] A. H. Land and A. G. Doig, "An automatic method of solving discrete programming problems," *Econometrica*, vol. 28, no. 3, pp. 497–520, 1960.

[31] H. Yan, X. Wu, Z. Lu, and X. Luo, "A neural network-based energy consumption prediction model for commercial buildings," *Energy and Buildings*, vol. 239, p. 110820, 2021.

[32] Z. Dong, J. Li, J. Li, X. Sun, and R. Li, "A deep reinforcement learning approach for building envelope optimization," *Automation in Construction*, vol. 119, p. 103355, 2020.

[33] Y. Chen, Y. Yang, and Y. Zhang, "Generative adversarial networks for computer-aided architectural design," *Advanced Engineering Informatics*, vol. 41, p. 100915, 2019.

[34] J. Cheng, R. Xiao, H. Ma, and X. Cheng, "Machine learning-based optimization for nonlinear elastic materials," *Computer-Aided Design*, vol. 122, p. 102884, 2020.

[35] W. Li, Y. Jia, X. Zhang, and S. Chen, "Reinforcement learning-based multi-objective

optimization for urban design," *Frontiers of Architectural Research*, vol. 8, no. 2, pp. 202–216, 2019.

[36] R. Zhong, H. Liu, H. Dong, X. Wang, and J. Zhang, "Deep reinforcement learning for product design generation," *Journal of Mechanical Design*, vol. 140, no. 10, p. 101401, 2018.

[37] K. Shingate, K. Jagdale, and Y. Dias, "Adaptive traffic control system using reinforcement learning," *International Journal of Engineering Research and*, vol. V9, 02 2020.

[38] Q. Fu, Z. Han, J. Chen, Y. Lu, H. Wu, and Y. Wang, "Applications of reinforcement learning for building energy efficiency control: A review," *Journal of Building Engineering*, vol. 50, p. 104165, 2022.

[39] J. Yang, Z. Cheng, G. Xiao, X. Xu, Y. Wang, H. Ding, and D. Zhou, "Engineering design optimisation using reinforcement learning with episodic controllers," *Cognitive Computation and Systems*, vol. 4, no. 4, pp. 340–350, 2022.

[40] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas, "Sample-efficient actor-critic with experience replay," in *Advances in neural information processing systems*, pp. 3192–3202, 2017.

[41] O. A. Osoba, C. J. Baker, and D. Manocha, "Autonomous systems and the urban environment: A review of current and future research directions," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 7, pp. 4323–4343, 2020.

[42] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[43] R. Luo, Y. Chen, L. Xie, and Y. Li, "Recent advances in deep reinforcement learning for industrial control and robotics," *Journal of Manufacturing Systems*, vol. 50, pp. 25–34, 2019.

[44] R. Y. Rubinstein and D. P. Kroese, *Simulation and the Monte Carlo method.* John Wiley & Sons, 2016.

[45] R. Bellman, *Dynamic programming.* Princeton university press, 1957.

[46] H. Lipson, "Evolutionary synthesis of kinematic mechanisms," *AI EDAM*, vol. 22, no. 3, pp. 195–205, 2008.

[47] M. E. Ororbia and G. P. Warn, "Structural design synthesis through a sequential de-

cision process," in *ASME 2020 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, American Society of Mechanical Engineers, 2020.