UNIVERSITY OF CENTRAL OKLAHOMA

Jackson College of Graduate Studies

FINDING THE OPTIMAL SOLUTION FOR A SPECIFIC PLANNING PROBLEM USING SEARCH-GUIDED REINFORCEMENT LEARNING

By

Chi San Lau (Derrick Lau)

A Thesis Submitted to

Dr. Jicheng Fu

Dr. Gang Qian

Dr. Junghwan Rhee

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

December 2020

UNIVERSITY OF CENTRAL OKLAHOMA

Jackson College of Graduate Studies

Finding the optimal solution for a specific planning problem using search-guided reinforcement learning

Thesis Title

Chi San Lau (Derrick Lau)

Author's Name

12/02/2020

Date

A THESIS APPROVED FOR

THE DEGREE OF MASTER OF SCIENCE IN COMPUTER SCIENCE

By

Jicheng Fu Digitally signed by Jicheng Fu Date: 2020.12.01 21:27:55 -06'00'

(Dr. Jicheng Fu) Committee Chairperson

Qian Gang Digitally signed by Gang Qian Date: 2020.12.02 13:07:31 -06'00'

(Dr. Gang Qian) Committee Member

[illegible]

(Dr. Junghwan Rhee) Committee Member

THESIS ABSTRACT

Artificial Intelligent (AI) Planning is a very important research field, in which an AI planner strives to generate a plan (i.e., a series of actions) that can bring a system from its initial state to the goal state. Given a planning problem that is modeled with an initial state and a goal state, existing AI planning algorithms typically employ search-based methods to find a plan. However, search-based methods are subject to search space explosion, i.e., when planning problems scale up, the search space may experience exponential growth. This problem becomes even more serious when an action is non-deterministic, i.e., one action can generate multiple possible outcomes. In this study, we focus on Fully Observable Non-Deterministic (FOND) planning problems, where an action can generate non-deterministic, but observable effects.

Specifically, instead of searching for a plan, we aim to develop an artificial neural network that can predict the optimal action for a given state so that the action will bring the system to a new state closer to the goal. Then, we can recursively apply the neural network to the new state to predict the next action until the goal state is reached. The novelty of our research is that the neural network is trained by using a reinforcement learning approach, which enables self-learning without relying on prior knowledge and human intervention. Furthermore, we have explored the possibility of combining AI search and reinforcement learning to make the self-learning process more efficient.

Experimental studies showed that our proposed approach could learn how to optimally solve specific planning problems from scratch. In the next step of research, we will explore how to use reinforcement learning to achieve a more generalized capability of solving a class of planning problems.

KEYWORDS: *Artificial Intelligence (A.I.); Artificial Neural Network, Fully observable nondeterministic planning (FOND); Q-Learning; Reinforcement Learning; Temporal Difference;*

PART1: INTRODUCTION

1.1 AI Planning and blocks world problem

Artificial Intelligence (AI) is a sub-domain of Computer Science. Compared to the traditional approaches, AI can solve problems more wisely with a certain level of intelligence, including behaviors such as learning, reasoning, planning, problem-solving, etc.

AI Planning is a very important research field in Artificial Intelligence. Its goal is to use automation techniques to optimize the decision-making process. Specifically, a planning problem is modeled with an initial state and a goal state. A plan is a solution to a planning problem, consisting of a sequence of the actions that can bring the system from its initial state to the goal state.

In this study, we focus on the Fully-Observable Non-Deterministic (FOND) planning problem, which is an important and challenging research area (Jaramillo, Fu et al. 2014). In a FOND problem, an action may generate multiple non-deterministic, but observable effects. Specifically, we chose a well-known benchmark problem, blocks-world problem, as the domain in this study. The setting of a blocks-world problem is that there are a robotic arm and a set of blocks that are sitting on the table. The blocks can be piled up to form a vertical stack(s) on the table and the orders of these blocks in the stack(s) define different states. The robotic arm needs to move the blocks from a given initial state into a particular goal state. The robotic arm can pick up a block or a tower (two blocks in a vertical stack) and put them down on the table or another block. Due to mechanical constraints, an action of the robotic arm can be ended up in multiple non-deterministic outcomes. For example, if the robotic arm picks up a block, it may grip the block successfully or there is also a chance for it to slip out of the gripper and drop on the table. Therefore, in a state, when the robotic arm applies an action, it may result in more than one state. Therefore, if we use the traditional search-based approach to go from an initial state and recursively apply actions to expand the search space to find a path to the goal state, the search space may grow very fast due to the multiple possible effects generated by the actions. As a result, the larger is the search space, the longer time it needs to take to find a solution. The daunting complexity can lead to a computational difficulty, if the computational resources necessary to solve the problem become impossible to be fulfilled, then the problem cannot be solved either (Homer and Selman 2011), i.e., when a planner runs longer than expected, researchers simply do not know whether it needs more time to finish or it will never stop.

Moreover, even if a planner finds a path from the initial state to the goal state, it will be just a weak plan, because the path may only contain one of the possible outcomes of an action while other outcome(s) remain unhandled. Therefore, a complete plan should include a path for each possible outcome to reach the goal state. Such a plan is called a strong plan (with no cycles in the solution space) or strong cyclic plan (i.e., following the plan may run into indefinite loops, but once exiting the loops, it will guarantee to reach the goal) (Cimatti, Pistore et al. 2003).

In this study, not only we want to find a solution to a planning problem, but we also aim to find the optimal solution in terms of the size of the plan to a planning problem. Research has shown that to find an optimal strong or strong-cyclic solution to a blocks-world problem in general is an NP-Complete problem (Baral, Kreinovich et al. 2000). Imagine that a planning problem involves a large number of blocks, the distance to the goal state becomes much farther and the combinations of actions and their non-deterministic outcomes may grow exponentially. As a result, the blocks-world problem is a widely used benchmark planning problem in research and international AI planning competitions (Fu, Ng et al. 2011).

1.2 Machine Learning and Reinforcement Learning

Machine Learning (ML) is a subset of Artificial Intelligence. As its name suggests, it focuses on the learning part of AI. Machine learning lets computers to learn from some prior knowledge to find out how to perform tasks or to solve problems. Instead of directly writing code to program the logic, ML analyzes the patterns and structures from the training data. In real-world applications, there are many problems that we humans understand abstractly, but it is very difficult for us to generalize them into programming logic. In many cases, it is impossible to explicitly enumerate every characteristic of a problem due to the intractable number of possible situations. Therefore, machine learning is a very important technique that makes it possible for machines to understand a challenging problem without relying on human knowledge and/or human intervention.

As a typical example of classifying cats and dogs, it is extremely easy for humans to tell which one is a dog and which one is a cat. Nevertheless, it is very hard for us to accurately describing the differences between cats and dogs using computer logic. We can understand some concepts at an abstract level, but we are unable to explain them very explicitly. By using machine learning, given enough cat and dog images as training data, the computer can learn to distinguish between cats and dogs, and to some extent, a learning algorithm can even achieve superhuman performance.

The aforementioned cats-and-dogs example is a very classic example of supervised learning. In supervised learning, we label the training data to teach a learning algorithm which one is a cat and which one is a dog. Hence, the learning algorithm can learn from the data and correct labels to establish the relationships between them.

In contrast to supervised learning, unsupervised learning does not rely on labeled training data. Instead, a learning algorithm compares and group the data according to the data’s intrinsic characteristics. With a minimum level of human supervision, unsupervised learning works especially well in clustering and some bioinformatics problems, which sometimes are difficult for a human to manually categorize data into different groups. The downside is that as humans do not give direct guidance in the learning, it may generate unexpected or undesired categories (Kyan, Muneesawang et al. 2014).

Common to supervised and unsupervised learning approaches is that both of them need a large number of training samples. However, it is not always possible for a large set of training data. Sometimes, even if we can have a large number of samples, we should still ensure that the data set is well balanced to cover all the possible situations. The biased dataset will mislead the subsequent training process.

This is the place where the third category of ML techniques, reinforcement learning, comes into play. Unlike supervised and unsupervised learning, it may not require a large number of training data to learn. The characteristic of a reinforcement learning algorithm is that it employs the trial-and-error strategy to learning from its past experience. Reinforcement learning typically uses rewards to guide the learning process so that the correct decisions will be rewarded while poor decisions will be penalized. As the experiences and feedbacks on decision-making accumulate, a reinforcement learning algorithm will be able to learn without relying on prior knowledge or human intervention.

In this study, we employ the reinforcement learning technique to solve blocks-world planning problems. Our contributions are that we (1) have modeled a blocks-world planning problem in a way that is accessible to a learning algorithm; (2) designed an approach to mask out invalid actions so that the learning process is substantially accelerated; (3) designed a novel scheme to reward correct decisions and penalize wrong decisions during the learning process; (4) combined a heuristic search technique with the reinforcement learning so that the learning process is guided; and (5) conducted extensive experiments to validate the proposed approaches. Experimental results demonstrated that our approach could learn how to optimally solve a planning problem without relying on prior knowledge and human intervention. The knowledge and experience learned from this study will pave the way for future research to be able to generalize the learning process to solve a class of planning problems.

PART2: LITERATURE REVIEW

2.1 AI Planning and FOND planning problems

AI Planning is a sub-field of Artificial Intelligence and it started in the 1970s (Long and Fox 2002). Although it has been developing for several decades, it does not change the fact that AI planning is a difficult task. In this study, our focus is on fully observable non-deterministic (FOND) planning, especially one of the most well-known benchmark problems, the blocks-world problem. FOND is challenging because an action may result in multiple possible outcomes and the planning methods need to consider all the possibilities (Winterer, Wehrle et al. 2016).

Simply finding a non-optimal solution to a blocks-world problem is easy, but finding an optimal (shortest) plan is an NP-hard problem. An AI planner may run indefinitely without even stopping, not to mention finding an optimal plan (Gupta and Nau 1992).

AI Planning has made notable advances in the last few decades and the existing approaches typically use search-based techniques to look for a solution, e.g., the Stanford Research Institute Problem Solver (STRIPS) (Fikes and Nilsson 1971), Real-time Dynamic Programming(RTDP) (Barto, Bradtke et al. 1995), Pistore and Traverso 's planning algorithm (Pistore and Traverso 2001), Loops AO*(LAO*) (Hansen and Zilberstein 2001), FIND-and-REVISE, HDP, HDP(i) (Bonet and Geffner 2003), Goal Agenda Manager(GAM) (Koehler and Hoffmann 2000), Fast-Forward(FF) (Hoffmann and Nebel 2001), SGPlan (Chen, Hsu et al. 2004), Model Based Planner (MBP) (Bertoli, Cimatti et al. 2001), HTN-MAKER$^{\mathbf{ND}}$ (Hogg, Kuter et al. 2009), PRP (Muise, McIlraith et al. 2012), Fast Incremental Planner (FIP) (Fu, Ng et al. 2011) (Fu, Jaramillo et al. 2016), etc.

However, as a planning problem grows more complex, the search space may grow exponentially. For the blocks-world planning problem, there are 13 states in 3-block problems, 501 states in 5-block problems, 58941091 states in 10-block problems, 3276979278860856544441 states in 20-block problems, and so on. Besides the huge number of states, state transitions between the states make the blocks-world problem even more difficult to handle. Let us take the 3-block problem as an example again. The number of transitions between the 13 states is 30. As the base (the number of states) grows exponentially, the number of state transitions grows at an increasingly faster rate corresponding to the number of states. (Bădică, Bădică et al. 2020)

Therefore, the complexity of the large state space in planning problems is also an important problem that we need to handle in this study. In reality, depending on the task, if the planner cannot find a proper solution, the consequence can be critical. In 1997, the Mars Pathfinder rover had been estimated that it spent between 40% and 75% of its time doing nothing because of the failure of the planner, the plan could not tell the Mars Pathfinder how to deal with the situations it encountered (Bresina, Dearden et al. 2012). Hence, for non-deterministic planning, it is essential to solve the problems with strong or strong-cyclic plans and properly deal with the state explosion problem so that the planner will not get lost in the large state space and run forever.

2.2 Artificial Neural Networks and Reinforcement Learning

Artificial Neural Network (ANN) is a network of artificial neurons. The idea of “artificial neuron” was first proposed by McCulloch and Pitts in the 1940s (McCulloch and Pitts 1943), and then there was a second wave of extensive research activities occurring in the 1960s, yet at that time it was still far from the ANN we have today (Rosenblatt 1962). Until the late 1980s, after Hopfield’s energy approach (Hopfield 1982) and Werbos’ back-propagation learning algorithm for multilayer perceptron (multilayer feedforward networks) (Werbos 1974) were proposed, Artificial Neural Network was built up and started to develop (Jain, Mao et al. 1996).

The invention of ANN was inspired by the study of a biological system. It is a mathematical creation whose purpose is to map inputs into desired outputs (Priddy and Keller 2005). This is imitating the interconnections between the neurons in the biological system such as human brains. The operation of a neural network is managed by neuron dynamics, which has two parts: the dynamics of activation state and the dynamics of synaptic weights. These can help us map the input to the output although we may not fully understand the operations inside (Yegnanarayana 2009).

Artificial Neural Network is now a sub-area under Machine Learning (ML). Machine learning is a study aiming to teach machines how to learn from the data and perform some tasks based on the experience it learned. It can let the machines to understand the pattern and information from the data and make good use of it. (Dey and Technologies 2016)

There are three major types of learning methods: Supervised Learning, Unsupervised Learning, and Reinforcement Learning. Supervised learning is like having a “teacher” to supervise the learning process, telling the network what the desired results (labels) should be (Priddy and Keller 2005). Then the neural network, just like a student, by knowing the relations between the pattern of the data and the results, can learn what it should do when a similar task is given. In contrast, in Unsupervised learning, there is not such a “teacher”, so the machine has to find out the hidden structures in the data, trying to categorize the data according to some similarities in it (Dey and Technologies 2016).

While supervised learning and unsupervised learning both learn from the data, the difference is whether there is a teacher to tell which one is the answer. In contrast, reinforcement learning has no knowledge (data) about what to do until it has been given a situation to start with. The learning algorithm chooses the actions not based on any given training data, but its own experience (Priddy and Keller 2005). When the learning algorithm tries to do some action, it receives some numerical reward. That is the key to reinforcement learning because the learning algorithm will try to maximize the reward it gets to ensure that it is performing better and better. Moreover, the actions it chooses may not only affect the immediate reward, but also the future rewards it receives. Thus, trial-and-error and delayed reward are the two most important characteristics of reinforcement learning (Sutton and Barto 2018).

PART3: RESEARCH PURPOSE

Artificial Intelligence (AI) Planning represents a long-term dream in Computer Science, which aims to automatically generate a plan bringing a system from its initial state to a goal state. In previous studies, state-of-the-art AI planners typically employ search-based approaches to seek solutions. However, such planners are subject to space explosion when a planning problem scales up in complexity.

In our research, we want to explore the possibility of using reinforcement learning techniques to solve AI planning problems without human intervention, i.e., achieving self-learning. We have leveraged AI search and machine learning techniques to make the self-learning process more effective.

Machine learning, specifically reinforcement learning, offers an alternative direction for addressing AI planning problems. We aim to develop an AI planner that can learn all by itself about how to solve specific planning problems with optimal solutions. The combination of AI search and reinforcement learning techniques bring the efficiency of self-learning to the next level.

Experimental results showed that not only our approach was able to learn how to solve specific planning problems, but it could also result in optimal solutions in terms of the size of plans.

PART4: METHODS

In this study, we focused on a benchmark planning domain, blocks-world, in which a robotic arm attempts to move blocks from an initial configuration (i.e., state) to a goal state. Due to mechanical constraints, the robotic arm may drop a block onto the table during the process. The goal is to find a plan that can achieve the goal state although non-deterministic effects are possible.

4.1 Inputs

In AI planning domain, Planning Domain Definition Language (PDDL) is the standard language for the planning tasks (McDermott, Ghallab et al. 1998). Figure 1 is an example of a 5-block blocks-world problem in PDDL format representation.

```
(define (problem bw_5)
  (:domain blocks-domain)
  (:objects b1 b2 b3 b4 b5 - block)
  (:init (emptyhand) (on b1 b3) (on b2 b1) (on-table b3) (on-table b4) (on b5 b4)
(clear b2) (clear b5))
  (:goal (and (emptyhand) (on b1 b2) (on b2 b5) (on-table b3) (on-table b4) (on-
table b5) (clear b1) (clear b3) (clear b4)))
)
```

Figure 1. Example of a 5-block blocks-world problem in PDDL format

A PDDL problem file representation clearly shows what are the objects participating in the world that we are interested in. It also defines the initial state that we start with and the goal state that we want to achieve. Although it may not be very intuitive for a human to read, for a computer program, it is very clean and accurate in specifying the configuration of the objects (See Figure2).

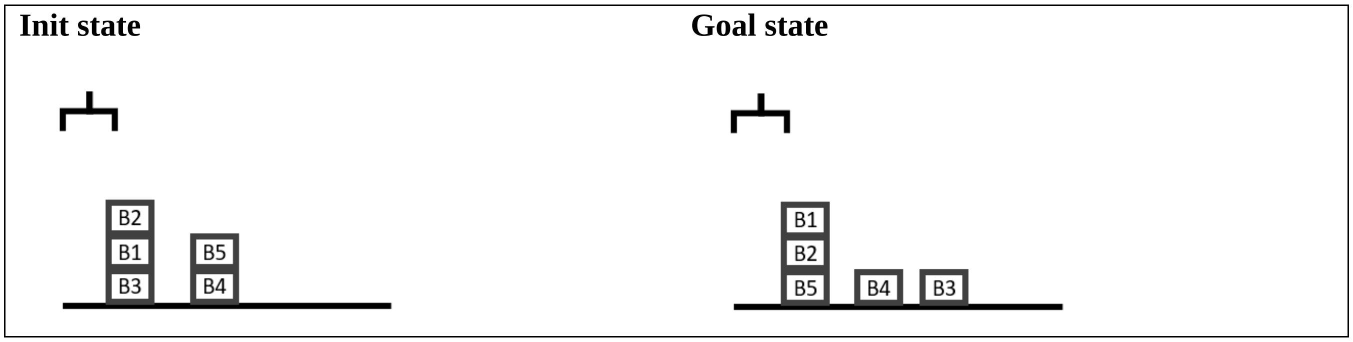


Figure 2. Visualization of the PDDL 5-block blocks-world problem in Figure1.

For the planning task specified in PDDL format, aside from the problem file(s), there is also a domain file. While a problem file defines what the problem is like – the objects, the initial state, and the goal specification, the domain file specifies the predicates and the actions, which a planner can choose to cause state transitions to happen. This type of input suits AI planners, yet for the machine learning model, it is not a good fit.

To convert a planning problem into a form that can be processed by a machine learning algorithm, we model a state (i.e., the positions of blocks and the status of the robotic arm) as a two-dimensional (2D) matrix. The motivation is that a machine learning algorithm, such as the artificial neural network, cannot directly use PDDL description for learning. Its inputs have to be formatted in the tabular form, with each column corresponding to a feature variable.

Then the next step is how we can transform a blocks-world problem into the tabular format. Specifically, a blocks-world problem should define what the blocks are in the problem (the objects), and where the objects are placed in the initial and goal specifications (the states). Hence, we come up with a novel idea to use a 2D matrix to denote a state.

As shown in Figure 3, the rows of the array represent the blocks, while the columns represent the blocks, the table, and the robot arm. The intersection of a row and a column (i.e., a cell) represents the relative position of two items, e.g., one block is on another block, or on the table, or a robotic arm is holding a block, etc. As a planning problem consists of an initial state and a goal state, we use two 2D matrices to represent the planning problem. Since the initial state and the goal state both affect what we are going to do next, i.e., if either the initial state or the goal state changes, the action we should choose next will be different. Both the initial and goal states are essential to a planning problem. Hence, we stack the initial and goal states (i.e., the converted matrices) together as an input to the learning algorithm.

After choosing what action to be applied to the initial state, state transition will happen, i.e., move from the initial state to another state meanwhile the goal state still stays unchanged. Thus, the first 2D matrix (i.e., the current state) is always changing while the second 2D matrix (i.e., the goal state) is always the same. When we start to solve the planning problem from the very beginning, the initial state is the current state, after an action is determined and applied, the current state will change to the next state. This process will repeat and the current state will continue to change triggered by the application of appropriate actions until the current state is identical to the goal state, i.e., has reached the goal state. In consequence, the input of the machine learning algorithm is a matrix pair, one representing the current state and the other representing the goal state.

4.1.1 2D array based on number of blocks

As we mentioned that the rows represent the blocks and the columns represent the table, blocks, and robot arms, it means that the blocks in the problem will appear on both the rows and columns. The number of blocks determines the size of the matrix. For instance, if we have a 5-block planning problem, then we will have $Objects = \{B1, B2, B3, B4, B5\}$. Accordingly, we have 5 rows (B1 to B5) and 7 columns (Table (T), B1 to B5, Robotic Arm (H)). That is, for a 5-block problem, the dimensions of the state arrays will be 5 x 7 (See Figure 3).

Init state

	T	B1	B2	B3	B4	B5	H
B1	0	0	0	1	0	0	0
B2	0	1	0	0	0	0	0
B3	1	0	0	0	0	0	0
B4	1	0	0	0	0	0	0
B5	0	0	0	0	1	0	0

Goal state

	T	B1	B2	B3	B4	B5	H
B1	0	0	1	0	0	0	0
B2	0	0	0	0	0	1	0
B3	1	0	0	0	0	0	0
B4	1	0	0	0	0	0	0
B5	1	0	0	0	0	0	0

Figure 3. The 2D matrix representation of the 5-block blocks-world problem in Figures 1 and 2.

Let us use the initial state in the 5-block problem above as an example. In row0 (B1), it represents the object B1 and the columns of this row represent where the object is currently located. The one (1) in the row indicates where the block is. Therefore, in row0, the location of block B1 is at B3, more specifically, B1 is on B3. Similarly, B2 is on B1, B3 is on T (the table), and so forth.

In this two-dimensional tabular format, we can easily draw a state as shown in Figure2. If we want to know where a block is currently located, we can just go directly to check the corresponding row to determine the location immediately.

According to the number of blocks of the problems we have, we can create the corresponding size of the table, i.e. a table of (the number of blocks) by (the number of blocks + 2). Hence, for a 10-block problem, we need a pair of 10 × 12 arrays, while we are dealing with a 3-block problem, we just need to create a pair of 3 × 5 arrays. The size of the arrays can just fit the problem we need to solve. The advantage is to save some space to store the state. However, there is a critical downside, which made us decide to switch to another improved format – i.e., the neural network cannot understand different shapes of inputs like we humans do.

For humans, if we have learned how to solve a difficult problem, when we see a simpler problem of the same type, the previous experience and knowledge will help us understand how to solve a problem that is simpler than what we have known about the difficult problem. Although it may not be guaranteed to help us solve the problem at once, usually it gives us a significant level of help rather than learning how to tackle a similar problem from scratch. Likewise, if a learning algorithm learned how to solve a 5-block problem before, it should be able to learn how to solve a 3-block problem. However, the input format of the 2D array based on the number of blocks is tightly bound with how many blocks we have in the problem. Imagine that we have a 10-block problem, the initial state is just one step away from the goal state, and what we need to do is just to put down B2 on B1. We train this problem in a machine learning model many times and it can solve this problem very well. Then we have another 5-block problem, which is very similar to the previous 10-block problem, it is also one step away and needs to do the same operation – to put down B2 on B1 (See Figure 4).

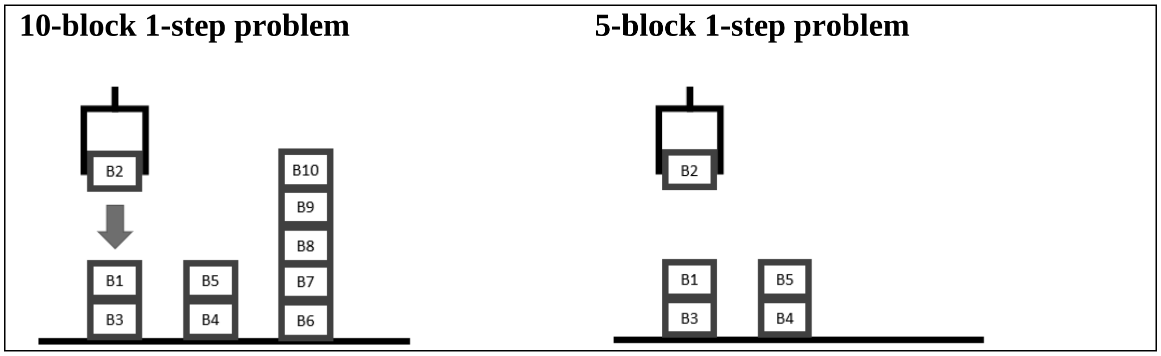


Figure 4. Visualization of comparing two similar 1-step problems of different number of blocks.

We can quickly find out that the two problems are nearly the same, the five blocks on the right side (B6 – B10) do not affect anything, we just have to do the same operation and on the left side what we have are the same 5 blocks (B1 – B5).

Nevertheless, from the perspective of the state matrix, because the whole shapes of the arrays are different, even they are in the same row and same column (B1, B3, B4, B5), to the machine learning model, they are neither the same nor relevant (See Figure 5).

10-block 1-step problem

	T	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	H
B1	0	0	0	1	0	0	0	0	0	0	0	0
B2	0	0	0	0	0	0	0	0	0	0	0	1
B3	1	0	0	0	0	0	0	0	0	0	0	0
B4	1	0	0	0	0	0	0	0	0	0	0	0
B5	0	0	0	0	1	0	0	0	0	0	0	0
B6	1	0	0	0	0	0	0	0	0	0	0	0
B7	0	0	0	0	0	0	1	0	0	0	0	0
B8	0	0	0	0	0	0	0	1	0	0	0	0
B9	0	0	0	0	0	0	0	0	1	0	0	0
B10	0	0	0	0	0	0	0	0	0	1	0	0

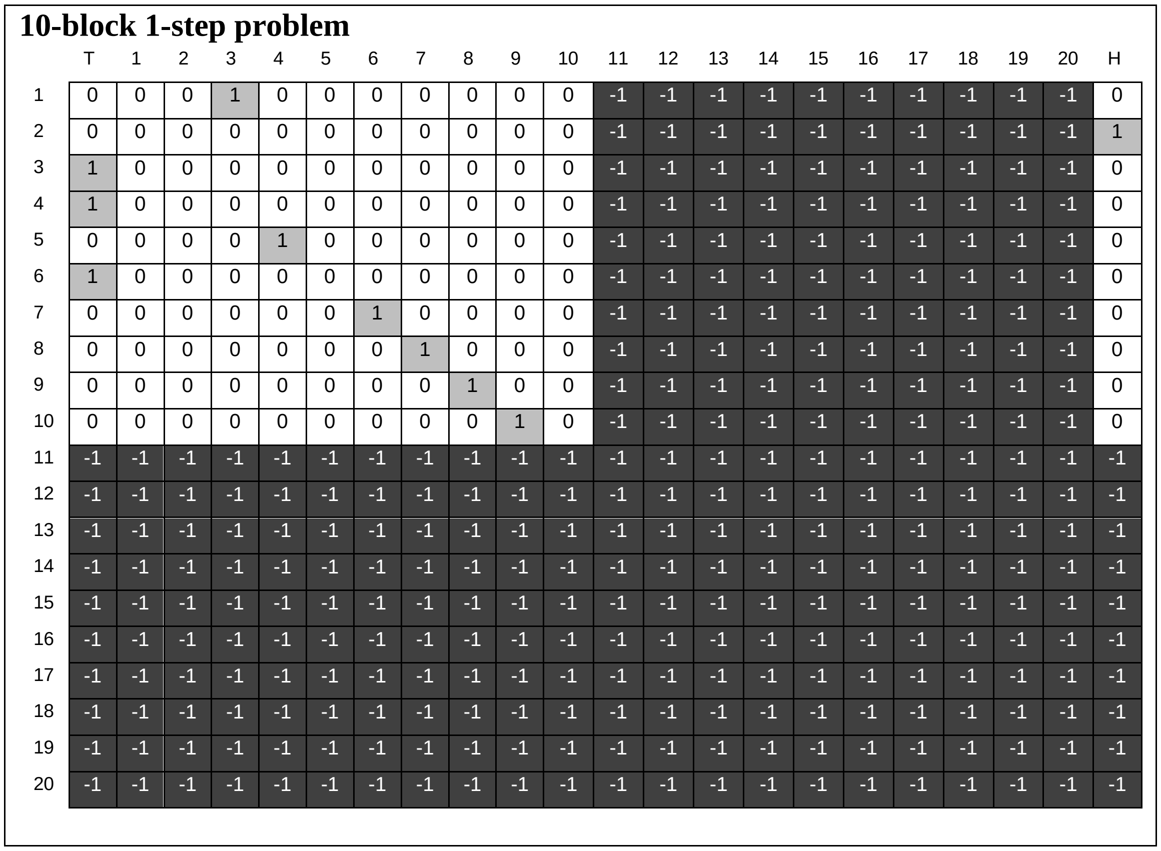
5-block 1-step problem

	T	B1	B2	B3	B4	B5	H
B1	0	0	0	1	0	0	0
B2	0	0	0	0	0	0	1
B3	1	0	0	0	0	0	0
B4	1	0	0	0	0	0	0
B5	0	0	0	0	1	0	0

Figure 5. The 2D matrix representation of comparing two similar 1-step problems of different numbers of blocks.

4.1.2 20×22 2D matrix for any problem up to 20 blocks

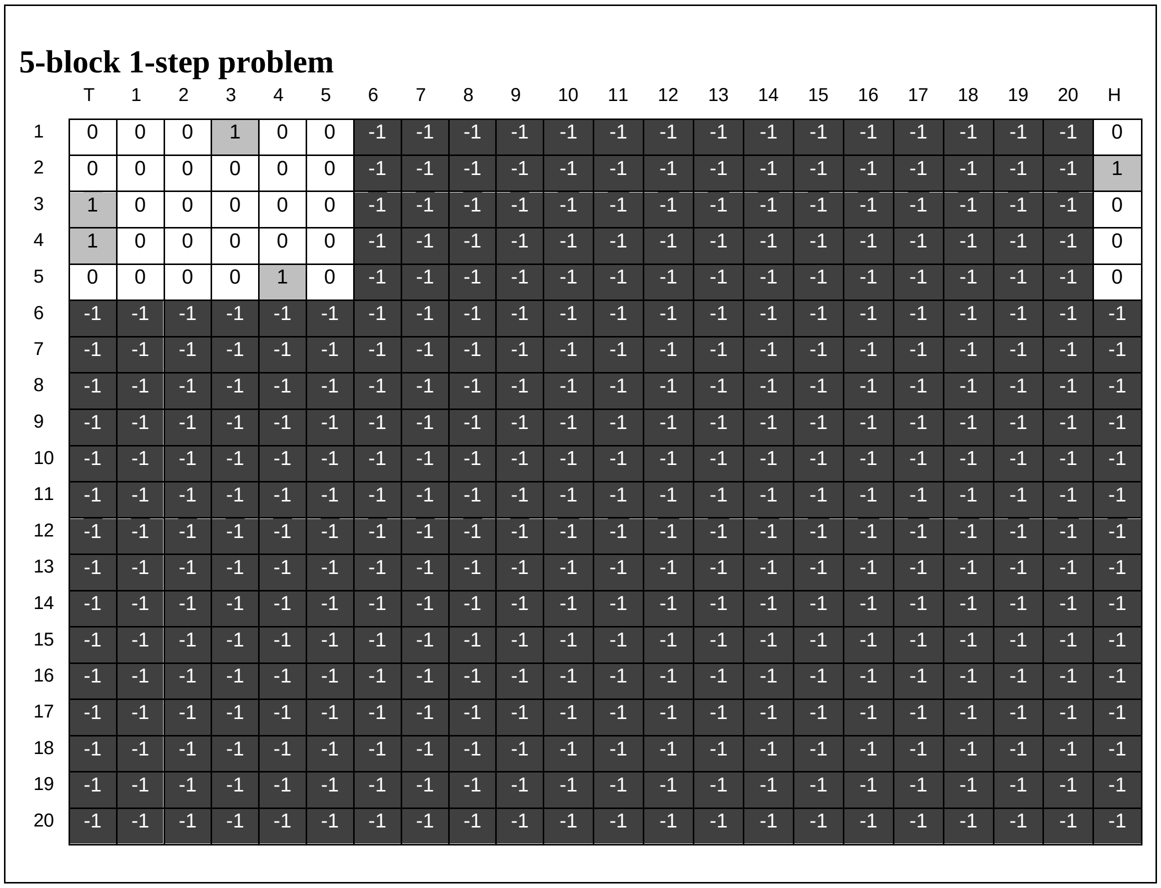
Although in this preliminary study we are still focusing on using reinforcement learning to solve specific planning problems, our ultimate goal is to be able to solve planning problems in general. For that reason, if the model can only learn and solve the problems of one particular number of blocks, i.e., after the model learns how to solve 5-block problems and later it can only solve 5-block problems, that is undesirable. We expect that the model can be compatible with problems of different numbers of blocks, therefore we updated the 2D matrix format in 4.1.1, i.e., we changed the shape of the array to a fixed size, 20 × 22, which means that it can store the states up to 20-block problems. Based on the number of blocks, for the unused rows and column, we filled them with negative ones (-1) so that the model can know that they are different from the 0 and 1. In spite of sacrificing more space to store the states in the input, this will help the model to recognize the relationship among the states in different numbers of blocks because they are all in the arrays of the same shape - 20 × 22 (See Figure 6.1 & 6.2).



10-block 1-step problem

	T	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	H
1	0	0	0	1	0	0	0	0	0	0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0
2	0	0	0	0	0	0	0	0	0	0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	1
3	1	0	0	0	0	0	0	0	0	0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0
4	1	0	0	0	0	0	0	0	0	0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0
5	0	0	0	0	1	0	0	0	0	0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0
6	1	0	0	0	0	0	0	0	0	0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0
7	0	0	0	0	0	0	1	0	0	0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0
8	0	0	0	0	0	0	0	1	0	0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0
9	0	0	0	0	0	0	0	0	1	0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0
10	0	0	0	0	0	0	0	0	0	1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0
11	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
12	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
13	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
14	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
15	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
16	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
17	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
18	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
19	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
20	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Figure 6.1 The 20 × 22 2D matrix representation of the 10-block state in Figure 5.



5-block 1-step problem

	T	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	H
1	0	0	0	1	0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0
2	0	0	0	0	0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	1
3	1	0	0	0	0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0
4	1	0	0	0	0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0
5	0	0	0	0	1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0
6	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
7	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
8	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
9	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
10	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
11	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
12	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
13	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
14	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
15	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
16	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
17	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
18	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
19	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
20	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Figure 6.2 The 20 × 22 2D array representation of the 10-block state in Figure 5.

We chose the size of 20 blocks because the complexity of the 20-block problem is already at a fairly high level. If in the future study the result shows that the model can master any general problem up to 20 blocks, we can increase the size of the input to be compatible with up to 50 blocks, 100 blocks, or even larger.

4.2 Artificial Neural Network

As the search space is huge, it is impossible to enumerate all the possible states. Instead, we build an artificial neural network that can predict the best action to choose based on a given state.

As the nature of this study is a preliminary study to show the feasibility of using reinforcement learning to solve AI planning problems, we will use the classical Artificial Neural Network (ANN), which is optimized by using the cutting-edge algorithm, i.e., Adam optimization (Kingma and Ba 2014).

4.2.1 Supervised Learning

From the beginning, we believed that reinforcement learning would be a better method because we do not have a large amount of state data. Even though we have collected a large volume of state data, the number of states in a blocks-world planning problem grows exponentially with the number of blocks. We will never have enough state data as inputs to train the ANN.

To verify this presumption, we used the supervised learning approach to train a neural network. It is a two-layer network, which has 128 hidden units in the hidden layer, and then 150 valid move combinations as the categories in the output layer. (For 5-block problems, there are about 150 valid moves, as the number of blocks increases, this number will increase exponentially, this will be further explained in Section 4.3.1)

We prepared ten 5-block problems and used the FIP planner (Fu, Jaramillo et al. 2016) to find out the legal plans for the ten problems. For instance, the plan to planning problem 1 consists of 10 states (including the leaf states generated by the non-deterministic outcomes): one of the states is the goal state itself, and nine non-goal states. Hence, we have 9 (non-goal state, goal state) pairs and 9 corresponding chosen actions from the FIP planner's result in problem 1.

We picked the first eight 5-block problems as the training set and the remaining two problems as the test set. We got 105 training examples (state matrix pair data) and labels (action indexes) and trained it for 1000 epochs. For the other two problems for testing, we had 22 testing examples and labels in total.

Experimental results showed that the model could perform 100% accurate predictions on the training set itself. But for the unseen testing sets, it only achieved 30~50% correctness. Obviously, overfitting has happened. For planning problems, this level of accuracy of finding the next move is not bearable, because it can lead to some very bad situations that are much harder to come back to the right track, let alone if the number of blocks increases, the accuracy of predicting the next move will drop dramatically because of the explosion of the number of possible valid moves.

Furthermore, supervised learning heavily relies on the training data it has seen before. If the problem we want it to solve looks very different from what it learned before, it is nearly impossible for it to choose a correct action. If we train on data specified in 5-block problems, the model can hardly predict other numbers of blocks correctly. It can only perform well unless we have training data of a wide range of problems of different numbers of blocks trained previously.

4.2.2 Reinforcement Learning

Due to the limit of supervised learning, we then use a reinforcement learning algorithm to learn how to solve a given planning problem.

We configure the neural network similar to the previous one, a 2-layer ANN, but we set the number of hidden units to a larger number 512. For the output layer, we arrange it into 440 categories, this will also be explained in Section 4.3.1.

Initially, the neural network knows nothing about AI planning except the rules. The reinforcement learning algorithm is responsible for training the neural network to predict the correct action based on the current state.

In our algorithm, there are two major parts: the environment and the agent, which are the key ideas in reinforcement learning. The environment determines the state transition and reward, i.e., if an action applies to the current state, the environment determines what the next state is (if the action is non-deterministic) and how much reward the agent would receive. The agent is the algorithm that receives information from the environment and makes the decision to choose an action, in order to move the system to a state closer to the goal state. The neural network is a part of the agent to help it to find out what is the best action to choose. After the agent picks an action, according to the action chosen by the agent, the environment will determine the next state and the reward, and then return the state and reward back to the agent. This process continues until the current state reaches a goal state or the time has run out (Sutton and Barto 2018) (See Figure 7).

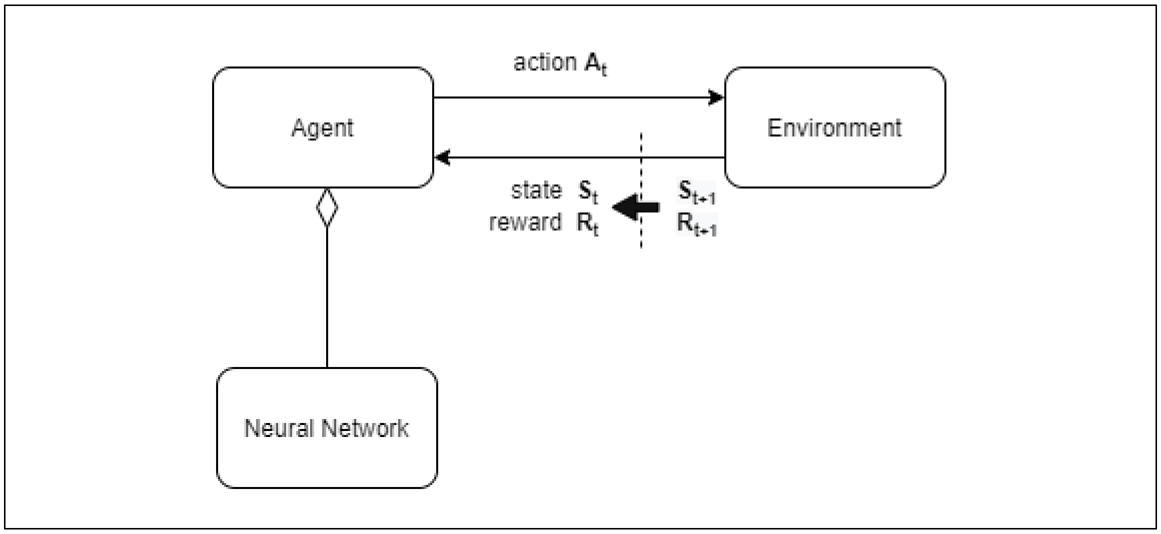


Figure 7. The relationship among our neural network, the agent, and the environment

4.3 Actions

In a blocks-world problem, at a specific state, as long as it is not yet the goal state, then we want to choose an action to move the system closer to the goal state. Considering different situations, the possible actions at a state can be classified into the following groups of action operations (See Table1).

Action operations can be performed at a state:

1. PICK-UP block1 block2 **Move** ROW[block1_index-1] **From** COL[block2_index] **To** COL[Hand_index]
2. PICK-UP-FROM-TABLE block1 **Move** ROW[block1_index-1] **From** COL[Table_index] **To** COL[Hand_index]
3. PUT-ON-BLOCK block1 block2 **Move** ROW[block1_index-1] **From** COL[Hand_index] **To** COL[block2_index]
4. PUT-DOWN block1 **Move** ROW[block1_index-1] **From** COL[Hand_index] **To** COL[Table_index]
5. PICK-TOWER block1 block2 block3 **Move** ROW[block2_index-1] **From** COL[block3_index] **To** COL[Hand_index]
6. PICK-TOWER-UP-FROM-TABLE block1 block2 **Move** ROW[block2_index-1] **From** COL[Table_index] **To** COL[Hand_index]
7. PUT-TOWER-ON-BLOCK block1 block2 block3 **Move** ROW[block2_index-1] **From** COL[Hand_index] **To** COL[block3_index]
8. PUT-TOWER-DOWN block1 block2 **Move** ROW[block2_index-1] **From** COL[Hand_index] **To** COL[Table_index]

Table 1. The possible action operations at a state and how these operations reflect on the state array

In the blocks-world problems, the robotic arm can hold at most a two-block tower at a time. This lets the robotic arm solve many cases faster by moving two blocks together. In a traditional planner, picking up a block and picking up a tower are two different moves. However, in our approach, by mapping the state into a two-dimensional array, the TOWER operations (Action operation groups 5~8) can be also simplified into normal block operations (Action operation groups 1~4). Hence, we actually only have 4 groups of different action operations. Suppose that we have a tower “block1 block2” (block1 is on top of block2) and if we want to move the tower to another location in the array, the location of block2 is the only thing we need to modify. This is because in the 2D array format, we do not need to change the location of block1, even though we move the tower “block1 block2”, block1 is still on block2. Hence, the location of block1 will remain unchanged. The nature of the 4 groups of action operations in the 2D matrix can all be abstracted into “Move [block] From [location1] To [location2]”.

4.3.1 Action combinations and indexes

Let us use a 5-block problem as an example again. As all actions can be summarized into the “Move [block] From [location1] To [location2]” form, we have 5 blocks (B1 to B5) in rows and 7 locations (Table + B1 to B5 + Robotic Arm) in columns, then we should have 5 × 7 × (7-1) = 210 combinations. However, there are some moves that are not valid in the array that we can remove. It is fine for having 5 blocks to choose to move. Imagine that we pick up B1, no matter where B1 is on, indeed there are not seven options but six, because B1 cannot be on itself. Assume B1 is currently on B3, and we want to move it to another place. There are seven columns, but we cannot move B1 onto B1 and we are not going to move B1 onto B3 either because that action is meaningless. Then we have only 5 options left.

Hence, we will have 5 × 6 × 5 = 150 action combinations that we can choose. If we perform an action in a 5-block problem, it must be 1 of the 150 actions. That is the reason in the early Section 4.2.1 we mentioned for a 5-block problem, the output layer will have 150 categories. The formula of the combinations for the possible actions in an N-block problem can be summarized as “N × (N + 1) × N”, i.e. for a 10-block problem, there will be 10 x 11 x 10 = 1100 action combinations.

Therefore, for 5-block problems, in the output layer of the neural network, there will be 150 units (category 0~149), and for 10-block problems, in the output layer of the network, there will be 1100 units (category 0~1099). There are two critical problems here. First, the shapes of the output layer of the neural network cannot match, which means, if a model trains for the 5-block problem, then it cannot be used to train for the 10-block problem. Second, if we label the action as an index in the categories and the sizes of the categories are different, then the indexes of the same actions may be different too. For instance, the action index of “Move B2 From T To B1” in 5-block problem is ‘30’, while in 10-block problems, because we also consider the moves on the locations B6~B10, the same move “Move B2 From T To B1”, its index in 10-block problem becomes 110. Even if the output layer shapes can match, the model cannot recognize the index 30 in a 5-block problem and the index 110 in a 10-block problem are indicating the same operation.

4.3.2 Action matrices

Because of the problem of incompatible format with different blocks problems, we re-modeled the action indexes to a new format – action matrix. After the state matrix format is changed into 20 × 22, the space of the 2D array is changed to fit up to 20-block problems.

We attempted to further simplify the action form “Move [block] From [location1] To [location2]”. Instead of storing location1 in the action, we can find out the location1 column at the time when we try to move to location2. As we have removed the location1, the actions are simplified into “Move [block] To [location]”. As a result, we merge (N+1) actions related to location1 into one action. Then we use a 20 × 22 2D matrix, which had been initialized to all zeroes (0), by changing the location we want to move to as one (1), to denote the movement (See Figure 8).

The action matrix of the move “Move B10 To B20”:

	T	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	H
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8. The representation of the 20 × 22 action matrix format

In this new action format, no matter 5 blocks, 12blocks, 20blocks… the actions are all mapped into a 20 × 22 matrix, thus the number of units in the output layer of the Artificial Neural Network can be fixed as 20 × 22 = 440 categories. This way we can solve the problem we raised in Section 4.3.1, the shapes of problems of different numbers of blocks are integrated into one, and for the same move in problems of different blocks, they now share the same index in the action matrices. More importantly, for problems of large numbers of blocks, we have simplified the action combinations. Suppose that we use the 20-block problem as an example, if we use the old format, we will have 20 × 21 × 20 = 8400 combinations, while in the new action matrix format, we just need to handle 20 × 22 = 440 combinations.

4.4 Q-values

In Section 4.2.2, we briefly introduced the idea of how the parts in reinforcement learning are related. After the agent chooses an action $\boldsymbol{A_t}$ and passes it to the environment, the environment will change itself to the new state and return the new state $\boldsymbol{S_{t+1}}$ and the reward of the new state $\boldsymbol{R_{t+1}}$ to the agent (The new state $\boldsymbol{S_{t+1}}$ and the reward of the new state $\boldsymbol{R_{t+1}}$ are then become the current state $\boldsymbol{S_t}$ and the reward $\boldsymbol{R_t}$ that the agent receives) (See Figure 7).

In other words, the process of trying to get to the goal state in reinforcement learning is indeed relying on the rewards, as the agent receives new rewards (feedbacks) from the environment so that it can continue to make new decisions trying to maximize its future rewards.

A state-action value function is called the Q function $\boldsymbol{Q(s,a)}$, so the state-action value is also called Q-value. The Q-value specifies the quality of the agent performing an action in a state, i.e. how good the action a is at the specific state s.

We have developed an Artificial Neural Network inside the agent, and we have the current state and goal state pair as the input of the network, then the input will go through the hidden layer and output as the 440 Q-values in the output layer, which correspond to the 440 action categories. (See Figure 9.)

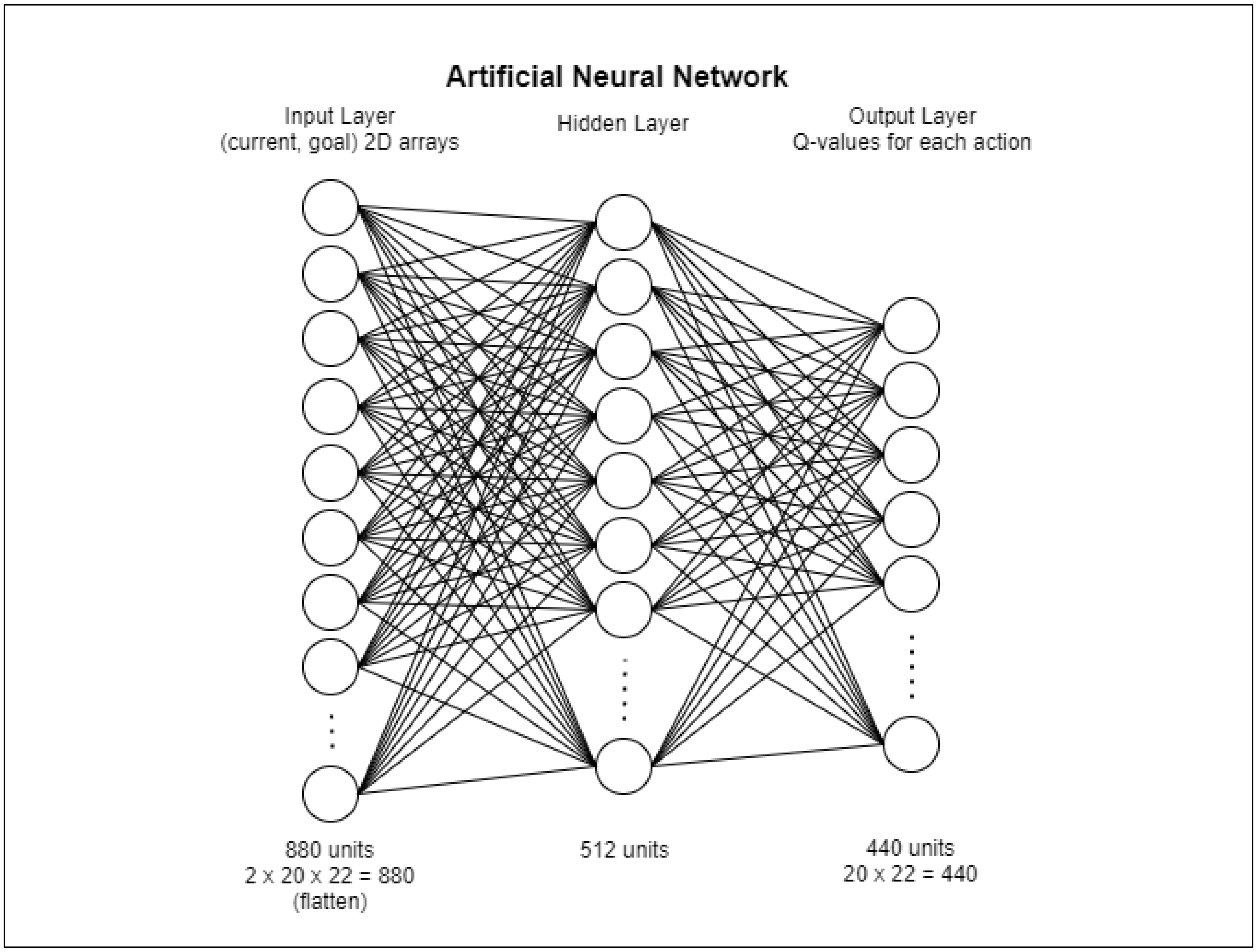


Figure 9. The Artificial Neural Network inside the agent

As for every planning problem, we have an initial state and a goal state. Our goal is solving the problem by reaching the ultimate goal state. Therefore, it is an episodic task, because we have a clear terminal state $\boldsymbol{T}$ (We also set a timeout so that the episode will end if it runs for too long, i.e. unreasonable long time).

Return for episodic task:

$$\boldsymbol{G_t} \doteq \boldsymbol{R_{t+1}} + \boldsymbol{R_{t+2}} + \boldsymbol{R_{t+3}} + \cdots \boldsymbol{R_T}$$

Figure 10. The Expected Return G_t for episodic task

The Expected Reward (or called Overall Reward) is the expected reward over an entire episode. For an episodic task, to calculate the Expected Return is easy. As it has a finite number of steps, we simply need to add all the future rewards $\boldsymbol{R_{t+1}}, \boldsymbol{R_{t+2}}, \boldsymbol{R_{t+3}} \ldots$ together (See Figure 10) (Sutton and Barto 2018).

However, what we need to find out is more than that. We want to find out the Q-values, which are also called the state-action values. The Q-values can reflect how good an agent chooses a particular action in a state with a policy $\boldsymbol{\pi}$ (In our study, we do not have a concrete policy, but we use the Q-values to convert to a policy that defines a set of probabilities to choose the action at a state). We use Q-values $\boldsymbol{Q}(\boldsymbol{s}, \boldsymbol{a})$ rather than the state values $\boldsymbol{V}(\boldsymbol{s})$ because it can give us additional information about how the action $\boldsymbol{a}$ works at the specific state $\boldsymbol{s}$.

Action-value (Q-value) function formula for episodic task:

$$\begin{aligned} \boldsymbol{Q}(\boldsymbol{s}, \boldsymbol{a}) &\doteq \mathbb{E}_{\boldsymbol{\pi}} \left[\boldsymbol{G_t} \mid \boldsymbol{S_t} = \boldsymbol{s}, \boldsymbol{A_t} = \boldsymbol{a} \right] \\ &= \mathbb{E}_{\boldsymbol{\pi}} \left[\sum_{\boldsymbol{k=t+1}}^{\boldsymbol{T}} \boldsymbol{R_k} \,\middle|\, \boldsymbol{S_t} = \boldsymbol{s}, \boldsymbol{A_t} = \boldsymbol{a} \right] \end{aligned}$$

Figure 11. Action-value (Q-value) function formula for episodic task

The Q-value is defined as the Expected Return $\boldsymbol{G_t}$ starting from state $\boldsymbol{s}$, taking action $\boldsymbol{a}$, and thereafter following policy $\boldsymbol{\pi}$. The Expected Return $\boldsymbol{G_t}$ is equal to what we discussed above in Figure 9. It is equal to the summation of all the future reward $\boldsymbol{R_{t+1}}, \boldsymbol{R_{t+2}}, \boldsymbol{R_{t+3}} \ldots \boldsymbol{R_T}$ till it reaches the Terminal state $\boldsymbol{T}$ (because it is an episodic task) (See Figure 11) (Sutton and Barto 2018).

Furthermore, not only we want to find out the Q-values for every episode in our research, but also, we want to know the Q-values every time after we move to the new state so that every new move picked by the agent is based on the latest Q-values. As this applies the idea of Dynamic Programming (DP), the core idea of reinforcement learning using in our research is the State-action value Bellman equation. (See Figure 12.)

State-action value (Q-value) Bellman equation:

$$\begin{aligned}\boldsymbol{Q_\pi(s,a)} &\doteq \mathbb{E}_\pi\,[\boldsymbol{G_t} \mid \boldsymbol{S_t = s, A_t = a}] \\ &= \sum_{s'}\sum_{r} \boldsymbol{prob(s', r \mid s, a)}\ \ [\boldsymbol{r} + \boldsymbol{\gamma}\mathbb{E}_\pi\,[\boldsymbol{G_{t+1}} \mid \boldsymbol{S_{t+1} = s'}]\,] \\ &= \sum_{s'}\sum_{r} \boldsymbol{prob(s', r \mid s, a)}\ \left[\boldsymbol{r} + \boldsymbol{\gamma}\sum_{a'} \boldsymbol{\pi(a' \mid s')}\ \mathbb{E}_\pi[\boldsymbol{G_{t+1}} \mid \boldsymbol{S_{t+1} = s', A_{t+1} = a'}]\right] \\ &= \sum_{s'}\sum_{r} \boldsymbol{prob(s', r \mid s, a)}\ \left[\boldsymbol{r} + \boldsymbol{\gamma}\sum_{a'} \boldsymbol{\pi(a' \mid s')}\ \boldsymbol{Q_\pi(s', a')}\right]\end{aligned}$$

Figure 12. State-action value (Q-value) Bellman equation

In this state-action value bellman equation, we define the $\boldsymbol{Q(s,a)}$ in terms of a recursive form. Given the state $\boldsymbol{s}$ and the action $\boldsymbol{a}$, we have a set of probabilities. Due to the blocks-world problem is non-deterministic, for each specific non-deterministic state $\boldsymbol{s'}$ and its reward, we sum up all the values for $\boldsymbol{s'}$ and $\boldsymbol{r}$. Moreover, according to our policy $\boldsymbol{\pi}$, given a state $\boldsymbol{s'}$, it chooses an action $\boldsymbol{a'}$. For each of the action $\boldsymbol{a'}$ we may choose, we add them all up in to the (state, action) pairs we choose, i.e. $(\boldsymbol{s'}, \boldsymbol{a'})$. Hence, we can convert the part into $\boldsymbol{Q_\pi}\left(\boldsymbol{s'}, \boldsymbol{a'}\right)$, so we can simplify the Bellman equation above into a recursive form.

4.5 Softmax and invalid moves masking

After the Artificial Neural Network in the agent finishes calculating the Q-values in the output layer, we implement the Softmax function to convert the Q-values into probabilities, i.e. we have 440 Q-values, so we will have 440 corresponding probabilities, which are interrelated with the action matrix discussed inside. The Softmax function applies the exponential function to each element $\boldsymbol{Q(s,a)}$ and divides them by the sum of all the exponentials. Since we compute the Softmax directly on the logits, so we set the temperature parameter $\boldsymbol{T}$ to equal one (See Figure 13) (Wiering and Van Otterlo 2012) .

Softmax Equation for Q-values
(Assuming temperature parameter $\boldsymbol{T}$ = 1)

$$\boldsymbol{Prob(a)} = \frac{\boldsymbol{e^{\frac{Q(s,a)}{T}}}}{\boldsymbol{\sum_i e^{\frac{Q(s,a_i)}{T}}}}$$

$$= \frac{\boldsymbol{e^{Q(s,a)}}}{\boldsymbol{\sum_i e^{Q(s,a_i)}}}$$

Figure 13. Softmax Equation for Q-values

In our Softmax function implementation, we pre-process the Q-values before we apply the exponential calculations to them. This is because we only want to assess the actions which are valid at the current state. In the previous Section 4.3, we discussed the valid action combinations, but those valid action combinations are just based upon whether they are allowed in the 2D matrix. The valid action we are talking about here is whether the action $\boldsymbol{a}$ is allowed to perform at the specific state $\boldsymbol{s}$. If an action should not be allowed at a specific state but the agent chooses it, then the whole plan is wrong. For this reason, even though the probability of performing an invalid move at a specific state is less than an extremely small number, as long as it is not 0% (impossible to be chosen), it is not tolerable.

To avoid choosing an invalid action, we made the following attempts.

First, we attempted to apply a mask to all the Q-values. The mask is an action matrix whose values are all initialized to zeros (0) and only the valid moves are labeled to one (1). We use another customized function to check whether a move is valid at a specific state. Then, we will find out the largest of all the Q-values, and let all Q-values subtract it to normalize all the Q-values. This is to avoid exceeding the maximum possible number that a programming language can handle. We make the invalid moves to have a value of 0 in the hope that it will not be chosen in the subsequent processing.

Nonetheless, we found that even we had applied one layer of mask to change all the Q-values of the invalid moves into 0, it was still not enough to avoid the invalid moves completely. The reason is that the $e^0 = 1$, although the invalid move action values may have been normalized by the subtraction by the max Q-values, after applying the exponential function, they may still have a possibility of being chosen.

Second, in order to make the exponential values of those invalid moves to become absolutely zero, we use the mask to spot out all the invalid move indexes, and then we change all their Q-values to -1e5, i.e. -1×10^5. In Python, the exponential value of -1e5 is zero (i.e., $e^{-10^5} \approx 0$). Accordingly, all the exponentials of the invalid move action-values become zeros, and the probabilities of the agent choosing these invalid moves also become 0%, i.e. impossible to be chosen.

There is still one extreme case found in our experiments. In the process when the environment keeps giving negative rewards in our learning process, there may be some Q-values of the valid moves turned out to be negative values that are too small (because of the subtraction we used above). They are too small that they reached the boundary that the exponential function would turn them into zeros. If all the values are too small and all their exponentials become zeros, then an exception comes – the summation of the exponentials is zero instead of the expected one. Since the summation of the exponentials is the denominator in the Softmax formula, so if it is zero, then it will trigger the “Divide by 0” exception and crash the program.

After we realized this problem, we found out the boundary value that would make this exception happen, and so we set a condition that if any of the Q-values are smaller than -7.440346e2, i.e. -744.0346, then we will divide all the small negative Q-values by ten (10) before it continues the Softmax calculation so that we can ensure that the Q-values will not exceed the boundary to trigger the exception to happen.

4.6 Update the weights

After we used the neural network to find out the Q-values of the (current, goal) state matrix pair, and then convert those values into a set of probabilities by applying Softmax and invalid move masking so that the agent can pick the next action. Then, the action is passed to the environment, which changes to a new state (one of the non-deterministic outcomes) and also generates a reward. Then the new state and reward are passed back to the agent, which would update the weights of the neural network. In our research, we used the semi-gradient Temporal Difference (TD) algorithm to tune the weights of the neural network.

4.6.1 Monte Carlo method

Before we discuss the Temporal Difference update method, we want to briefly introduce the Monte Carlo Method. The Monte Carlo method is the first algorithm that made reinforcement learning practical to solve real-world problems. It is a model-free reinforcement learning technique, which means that it does not rely on transition probability information like in the Markov Decision Process (MDP). It uses repeated random sampling to acquire numerical results (Sutton and Barto 2018) (See Figure 14).

```
Monte Carlo Method:

Input: a policy $\pi$ to be evaluated
        (in our case, $\pi$ is derived from the Q-values)

Initialize:
    $Q(s,a) \in \mathbb{R}$, arbitrary, for all $s \in S$, and all $a \in A$
    $Returns(s,a) \leftarrow$ an empty list, for all $s \in S$, and all $a \in A$

Loop forever (for each episode):
    Generate an episode following $\pi$ : $S_0, A_0, R_1, S_1, A_1, R_2, \ldots\ S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$ :
        $G \leftarrow \gamma G + R_{t+1}$
        Unless $S_t$ appears in $S_0, S_1, \ldots\ S_{t-1}$, $A_t$ appears in $A_0, A_1, \ldots\ A_{t-1}$
            Append $G$ to $Returns(S_t, A_t)$
            $Q(S_t, A_t) \leftarrow$ average($Returns(S_t, A_t)$)
```

Figure 14. Monte Carlo Method

Suppose that we have a policy $\boldsymbol{\pi}$ (π is derived from Q-values, it can be randomly generated because it will be optimized later), for each action-value $\boldsymbol{Q(s,a)}$, we initialize it to an arbitrary value and then return an empty list for all the (state, action) pairs to $\boldsymbol{Returns(s,a)}$.

For each episode, we follow π to generate a sequence of ($\boldsymbol{S_t, A_t, R_{t+1}}$) groups, where $= 0, 1, \ldots T-1$. Subsequently, we calculate the Expected Return $\boldsymbol{G}$ by going backwardly from $t = T-1, T-2, \ldots 0$. Thus, for each sequence, we obtain an Expected Return gain $\boldsymbol{G}$. Depending on the number of episodes, we may have many sequences and the corresponding $\boldsymbol{G}$s. We append each $\boldsymbol{G}$ to each (state, action) pair $\boldsymbol{Returns(S_t, A_t)}$, which we had initialized to an empty list. To calculate the expected state-action value $\boldsymbol{Q(S_t, A_t)}$, we take the average of the $\boldsymbol{Returns(S_t, A_t)}$ (the $\boldsymbol{G}$s we had collected).

4.6.2 Semi-gradient Temporal Difference (TD) update

In our research, instead of using the traditional Monte Carlo, we use the semi-gradient Temporal Difference (TD) method to help us tune the weights of the neural network. The problem of the Monte Carlo method is, we have to wait until each episode ends, then the learning starts. This is very inefficient. Imagine that we have a blocks-world problem and we use reinforcement learning to solve it, we try numerous steps in one episode, but we only update the Q-values once (our policy $\boldsymbol{\pi}$, how we choose an action $\boldsymbol{a}$ at a specific state $\boldsymbol{s}$, is derived from the Q-values). That is to say, during the episode, we cannot improve the way of the agent picking an action.

By using the Temporal Difference learning, we can update the weights after every step we try so that it can help us to learn much faster. It is the essential approach that lets us be able to do stochastic learning.

4.6.2.1 Temporal Difference (TD) error

In Section 4.4 and Figure 10, we discussed how the Expected Return Gain $\boldsymbol{G_t}$ is calculated from an episodic task. In the Monte Carlo method, we update the Q-value once only after every episode has completed,

i.e. $\boldsymbol{Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[G_t - Q(S_t, A_t)]}$, where $\boldsymbol{\alpha}$ is a hyperparameter to control how fast the model can learn.

On the contrary, the Temporal Difference (TD) error indicates how far the current estimated reward at the time step deviates from the actual reward that it received, and the algorithm acts to reduce this error. After every step, it will compare the actual reward received from the environment and compare it with the current estimated reward at the time step immediately so that the agent does not wait until the end of the episode, but update it at every time step (See Figure 15).

Unlike episodic task, return for continuing task is:

$$\boldsymbol{G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots}$$

$$\boldsymbol{= R_{t+1} + \gamma G_{t+1}}$$

$$\boldsymbol{Q(s,a) \doteq \mathbb{E}_\pi\,[G_t \mid S_t = s, A_t = a]}$$

$$\boldsymbol{= \mathbb{E}_\pi\,[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a]}$$

$$\boldsymbol{= R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})}$$

Therefore,

$$\boldsymbol{G_t \approx R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})}$$

Then,

$$\boldsymbol{Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[G_t - Q(S_t, A_t)]}$$

$$\boldsymbol{Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]}$$

$$\boldsymbol{Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[\delta_t]}$$

TD error $\boldsymbol{\delta_t}$:

$$\boldsymbol{\delta_t = R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)}$$

Figure 15. Temporal Difference (TD) error

Temporal Difference learning is actually one type of Monte Carlo method. It is an extreme Monte Carlo algorithm combining the Monte Carlo with the concept of Dynamic Programming so that when the task is ongoing in the episode but at every time step it can still optimize the weights. While Monte Carlo has to wait until the episode ends, TD can just wait for one more step and update the weights once it obtains the next state.

4.6.2.2 Q-learning

TD learning is a general method, which is not confined to Q-values only. That is why in the previous explanation sometimes we mentioned the policy $\boldsymbol{\pi}$ but we indeed do not have one in the learning process, because our focus is on Q-learning, which is also called off-policy TD control. Off-policy means that we do not follow an existing policy, but in Q-learning we can imagine we have a policy $\boldsymbol{\pi}$ that is derived from the Q-values. We do not have a 'real' policy in the beginning because we just arbitrarily initialize all the Q-values, nonetheless, when we start the iteration, we gradually learn the optimal policy. We keep updating our policy based on the Q-values, and based on the new policy we update the Q-values, until it converges. (Sutton and Barto 2018) (See Figure 16).

```
Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Input: a policy $\pi$ to be evaluated
    (in our case, $\pi$ is derived from the Q-values)

Initialize:
    $Q(s, a) \in \mathbb{R}$, for all $s \in S$, and all $a \in A$,
            arbitrary except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Loop for each step of episode :
        $A \leftarrow$ action given by $\pi$ derived from the Q-values for $S$
        Take action $A$, observe $R, S'$
        $Q(S, A) \leftarrow Q(S, A) + \alpha\left[R + \gamma \max_a Q(S', a) - Q(S, A)\right]$
        $S \leftarrow S'$
    Until $S$ is terminal
```

Figure 16. Temporal Difference (TD) learning for estimating $Q(s, a)$

After we initialized the Q-values to arbitrary values, we iterate the process for every episode, we initialize our current state $\boldsymbol{S}$ to the initial state of the problem, then for each step of the episode, we will use the Artificial Neural Network in the agent to predict an action $\boldsymbol{A}$. Taking action $\boldsymbol{A}$ and the environment will give the information of the next state $\boldsymbol{S'}$ and the corresponding reward $\boldsymbol{R}$. In the TD method, as long as we receive the next state $\boldsymbol{S'}$, then we can do the update process.

As we discussed in Section 4.4, we have a neural network that takes as input the (current state, goal state) pair, with the help of the weights in the network, it can calculate the Q-values for all actions. Then the agent will choose the action which leads to the maximal Q-value to be the next move. The 440 Q-values that we have are just for one specific state. As we mentioned in Section 2.1, the state space will grow exponentially when the number of blocks increases. Therefore, in Q-learning, we do not use a table to store the Q-values, in that case, we will have an astronomical number of rows and 440 columns. It will occupy too much of the memory and it is totally not manageable. That is also the reason why we adopt Artificial Neural Network to conduct the function approximation, i.e., at every time step when we have a specific state, we can use the neural network to predict the Q-values for all the actions applicable to that specific state.

4.6.2.3 Semi-gradient TD weights update

When training a machine learning model, gradient descent is an optimization algorithm that we usually use. The gradient descent algorithm tries to minimize the error between the actual output of the network and the true value we expected (i.e. the cost function $\boldsymbol{J(w)}$). If the error is minimized (i.e., 0), that means the weights we have are optimal. The weights will be updated by adding or subtracting the multiplication of the learning rate $\boldsymbol{\alpha}$ and the derivative of the cost function. The gradient descent algorithm keeps updating the weights until it is converged.

In error evaluation, we use the Mean Squared Error of the difference between the true value $\boldsymbol{Q(S_t, A_t)}$ and the predicted value $\boldsymbol{\widehat{Q}(S_t, A_t, w_t)}$. To this end, we find out the derivative of the error in order to minimize it, and then we can find out the weight updating rules for Monte Carlo (See Figure 17).

Gradient Descent

$$\boldsymbol{w_{t+1} \leftarrow w_t - \alpha \frac{\partial}{\partial w} J(w_t)}$$

Cost function:

Mean Square Error of the (True value – Predicted value)

$$\boldsymbol{J(w_t) = \frac{1}{2}\left(Q(S_t, A_t) - \widehat{Q}(S_t, A_t, w_t)\right)^2}$$

Derivative of Cost function (Minimize the Error)

$$\boldsymbol{\frac{\partial}{\partial w} J(w_t) = 2 \times \frac{1}{2}\left[Q(S_t, A_t) - \widehat{Q}(S_t, A_t, w_t)\right] \times \frac{\partial}{\partial w}\left(Q(S_t, A_t) - \widehat{Q}(S_t, A_t, w_t)\right)}$$

$$\boldsymbol{= \left[Q(S_t, A_t) - \widehat{Q}(S_t, A_t, w_t)\right]\left(0 - \frac{\partial}{\partial w}\widehat{Q}(S_t, A_t, w_t)\right)}$$

$$\boldsymbol{= -\left[Q(S_t, A_t) - \widehat{Q}(S_t, A_t, w_t)\right] \frac{\partial}{\partial w}\widehat{Q}(S_t, A_t, w_t)}$$

Thus,

the weight updating rule for Monte Carlo:

$$\boldsymbol{w_{t+1} \leftarrow w_t + \alpha\left[Q(S_t, A_t) - \widehat{Q}(S_t, A_t, w_t)\right]\left[\frac{\partial}{\partial w}\widehat{Q}(S_t, A_t, w_t)\right]}$$

We can replace the true value (Expected Return) with $\boldsymbol{G_t}$:

$$\boldsymbol{w_{t+1} \leftarrow w_t + \alpha\left[G_t - \widehat{Q}(S_t, A_t, w_t)\right]\left[\frac{\partial}{\partial w}\widehat{Q}(S_t, A_t, w_t)\right]}$$

Figure 17. The weight updating rule for Monte Carlo

In a similar way, the Temporal Difference method weight updating rule is nearly the same. Nevertheless, because in TD learning, rather than updating the weights after the episode ends, we do stochastic update after every time step. Hence, the Expected Gain $\boldsymbol{G_t}$ are different.

In the Temporal Difference (TD) method, we do it similarly as Monte Carlo.

The TD Update rule for Function Approximation:

$$\boldsymbol{w_{t+1} \leftarrow w_t + \alpha \left[G_t - \widehat{Q}(S_t, A_t, w_t) \right] \left[\frac{\partial}{\partial w} \widehat{Q}(S_t, A_t, w_t) \right]}$$

Referring to Figure 15, in TD:

$$\boldsymbol{G_t \doteq R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})}$$

If we check the Cost function…
Cost function:

Mean Square Error of the (True value – Predicted value)

$$\boldsymbol{J(w_t) = \frac{1}{2} \left(G_t - \widehat{Q}(S_t, A_t, w_t) \right)^2}$$

Derivative of Cost function (Minimize the Error)

$$\boldsymbol{\frac{\partial}{\partial w} J(w_t) = 2 \times \frac{1}{2} \left[G_t - \widehat{Q}(S_t, A_t, w_t) \right] \times \frac{\partial}{\partial w} \left(G_t - \widehat{Q}(S_t, A_t, w_t) \right)}$$

$$\boldsymbol{= \left[G_t - \widehat{Q}(S_t, A_t, w_t) \right] \left(\frac{\partial}{\partial w} G_t - \frac{\partial}{\partial w} \widehat{Q}(S_t, A_t, w_t) \right)}$$

Unlike in Figure17. Monte Carlo:

$$\boldsymbol{G_t \doteq R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})}$$

The true value here $\boldsymbol{G_t}$ is not a constant, so:

$$\boldsymbol{\frac{\partial}{\partial w} G_t \doteq \frac{\partial}{\partial w} R_{t+1} + \gamma \frac{\partial}{\partial w} Q(S_{t+1}, A_{t+1}) \neq 0}$$

However, in the TD method, we still assume:

$$\boldsymbol{\frac{\partial}{\partial w} J(w_t) = \left[G_t - \widehat{Q}(S_t, A_t, w_t) \right] \left(\frac{\partial}{\partial w} G_t - \frac{\partial}{\partial w} \widehat{Q}(S_t, A_t, w_t) \right)}$$

$$\boldsymbol{= \left[G_t - \widehat{Q}(S_t, A_t, w_t) \right] \left(0 - \frac{\partial}{\partial w} \widehat{Q}(S_t, A_t, w_t) \right)}$$

$$\boldsymbol{= -\left[G_t - \widehat{Q}(S_t, A_t, w_t) \right] \frac{\partial}{\partial w} \widehat{Q}(S_t, A_t, w_t)}$$

In other words, $\boldsymbol{\frac{\partial}{\partial w} G_t \neq 0}$, but we assume $\boldsymbol{\frac{\partial}{\partial w} G_t}$ to be 0.

Hence, we say that TD is a semi-gradient method.

Figure 18. The weight updating rule for Semi gradient TD

The weight updating rule in TD is still the same as Monte Carlo method, but the Expected Return $\boldsymbol{G_t}$ are different, if we substitute the $\boldsymbol{G_t}$ to the cost function, we will see the main difference – in Monte Carlo method, since we have completed the whole episode and we compute the Expected Gain $\boldsymbol{G_t}$ backwardly from the last reward to the first reward, so the $\boldsymbol{G_t}$ is a constant value. However, in the Temporal Difference (TD) method, the $\boldsymbol{G_t}$ value relies on the next state, it is not a constant value.

If we take the derivative of the cost function, in Monte Carlo method, we will get 0 for $\boldsymbol{G_t}$. However, in the TD method the derivative of the Expected Return $\frac{\partial}{\partial w}\boldsymbol{G_t}$ is not 0. Nevertheless, in practice, we assume this non-zero term equal to zero (0). Therefore, Temporal Difference is a semi-gradient method, and we can call it Semi-TD. (See Figure 18)

4.7 Attempts in Reinforcement Learning

After we set up the agent and environment for training the blocks-world problems, the model did not just learn the problem right away. We made many attempts to let the model learn how to solve the problems. In this section, we will discuss the process of how the model improved and what we have practiced during the learning process.

4.7.1 Learn from scratch without guidance

In the beginning, we tried to let the model learn from scratch without any guidance, simply trying by itself. We conducted the tests mainly using the 5-block problems because the complexity is neither too high nor too simple without any challenges. For a normal 5-block problem, it needs to take multiple actions to get to a specific state. Furthermore, we also need to consider the non-deterministic outcomes, i.e. if the robotic arm grabs a block, it may end up holding it successfully, or dropping it on the table.

Due to the randomly generated Q-values, the model did not have a good policy to deal with the problem in the beginning. It just kept trying but was very inefficient. After many hours of trying, we thought that it was too hard for the model to learn a multi-step problem by itself because this meant that it had to get to the goal state with a sequence of correct choices of actions before timeout. Therefore, we let the model solve a 1-step 5-block problem, i.e. just 1 step away from the goal state. After a while, it could solve the 1-step problem because the valid moves at a specific state of a 5-block problem were very limited. It could get it right by just blindly performing different actions. The initial goal we set for the model was just to be able to find a plan to solve the problem, no matter whether the quality of the plan was good or bad because after the model had the capability to solve a problem, we could always optimize it later.

For the timeout, we set a timeout of 2000 time steps, i.e., it could try 2000 times. If it had tried 2000 times but it had not reached the goal state, the current episode ended. If there were more episodes to run, the next episode would reset the system to the initial state. The intention of doing this was that the model might predict actions that would go farther and farther away from the goal state. Hence, we reset it back to the initial state so that it might choose different sets of actions to reach the goal state.

For the reward, initially we just had a simple reward scheme. If the next state was the goal state, then the environment would give it a positive one (+1) reward; if it was a non-goal state, then it would receive a 0 reward (i.e. no harm for trying). The advantage of this was the model would continue trying different actions without any penalties, but this was too slow. Thus, we also tried another way of rewards – if the new state was the goal state, then it would receive a positive 100 reward, but if the new state was a non-goal state, then it would receive a negative one (-1) reward, or we can call it a penalty. This would make the model try to choose actions that would achieve the goal state faster to avoid penalty.

4.7.2 Epsilon Greedy

In the previous sections, we mentioned that in Q-learning we always pick the action with the maximum action-value. It is called greedy policy and it is quite intuitive, we always attempt to get the best result with the maximum rewards we can achieve. However, we initialized the Q-values to randomized values. Even if we have trained our model for a while, we may still not be able to guarantee that the action it chooses is the best choice. Hence, in order to increase the chance of exploring the search space differently, occasionally, we may want to try some actions other than the one chosen by the Q-Learning algorithm. For example, as what we show in 4.7.1, the agent might choose a bad move but believed that it was a good one due to the highest Q-value. As another example, the model found that at state $\boldsymbol{S_1}$ it should choose action "PICK UP block1 FROM TABLE", which yields the next state $\boldsymbol{S_2}$. Then at $\boldsymbol{S_2}$, the action of "PUT DOWN block1 ON TABLE" has the highest Q-value. In that case, the model will end up in a dead loop in repeating these two actions. More importantly, the model learns nothing by repeating these two actions.

To avoid such cases, we used an approach called Epsilon Greedy policy. The idea is very straightforward, i.e., we set a parameter epsilon $\boldsymbol{\varepsilon}$ to a small value $(0 \leq \varepsilon \leq 1)$, e.g., 0.01. Then for a probability of $(1 - \varepsilon)$ (i.e., 0.99), it picks the action with the largest Q-value as normal. But if it falls into the $\boldsymbol{\varepsilon}$ case (i.e., 0.01), then the agent will choose a random action. In our research, that means picking one random valid action applicable to that specific state. Hence, with the Epsilon Greedy policy, even the model gets stuck in a dead loop, it will not get stuck forever as there is a small chance that it will try some other moves.

By adding Epsilon Greedy policy, it is very easy to solve a 1-step 5-block problem. For 2-step or 3-step problems, after running for a few hours, they are still solvable as the model kept trying continuously. Nevertheless, it still came to a bottleneck that if the complexity of the problem increases, it can be nearly impossible for the model to get to the goal state, at least within a reasonable time frame.

The reason is, for a normal 5-block problem, depending on how the state looks like, it may take 6~10 steps to go from the initial state to the goal state. It means that no matter what path it takes, it has to take multiple good actions in the plan, and especially at the time when it is one-step away from the goal state, it has to pick that very action to get to the goal state.

No matter whether we used 0 rewards or -1 rewards for the non-goal states, as long as it did not reach the goal state, it could not receive any positive reward from the environment. Without any human guidance, and just randomly go through a sequence of multiple actions, it is still possible to happen for simple problems with one to three blocks. However, for more complex problems (e.g., 6~10-step problems), the chance of getting to the goal state by randomly picking actions is very slim.

4.8 Distance Heuristic search guide

As the learning process is very time-consuming, and the most important thing - we may not be able to solve the problem just by blindly making attempts, we employed a search algorithm to guide the learning process. As the search itself is expensive, our reinforcement learning approach only runs the search algorithm occasionally during the learning process. Instead of just blindly trying actions without knowing any direction, with the search guidance, the reinforcement learning model can gradually learn how to get closer towards the goal state.

The algorithm we use to help to guide the model is based on the distance from the current state to the goal state. We called it the Distance Heuristic search.

The idea of the Distance Heuristic search is to compare the two state matrices (the current state and the goal state) row by row, and calculate an estimated distance between the two states (The shape of the state arrays is 20×22). For each row, it will calculate how many steps it needs to take to be placed at the correct column. The distances to the goal state of each row will be added up to the total distance, and then the function will return the total distance to the agent for it to compare which action will lead to a state that has the shortest distance to the goal state.

For the row comparisons, the algorithm will check the number of blocks of the problems. If there are 20 blocks in the problem, it will check all 20 rows (row 0 to row 19). If there are only 5 blocks in the problem, it will check the 5 rows only (row 0 to row 4). When it reaches row 5 (B6) and find the values of the row are ‘-1’, then the following rows will be skipped so that it will not waste any extra time to compare the remaining unused rows.

The distance to the goal state is estimated by comparing each row of the two matrices, if the two matrices are identical, i.e. the current state matrix is equal to the goal state matrix, then the comparison results of all rows will be 0. Hence, the total distance return will also be 0. However, if the two matrices are not identical, then we have to look into them carefully, to find out the situation why they are not identical, depending on how the blocks are placed in a state, there are totally 6 different sub-cases that we need to treat individually, because otherwise the distance estimation may not be accurate. We will first introduce how the algorithm operates, then explain what are the sub-cases and how we evaluate them (See Table 2).

Distance Heuristic Evaluation Algorithm:

1. Check whether H (the robotic arm) is holding any block / tower
if H is holding something, then check which block it is holding for later use.

2. The initial total distance to the goal state is 0.

3. Start comparing the two matrices row by row, compare the current state column to the goal state column.
(It will only compare the rows of the number of blocks in the problem.)

4.1 If the current state column and the goal state column are the same, then it means the block is at the right place, add 0 to the total distance, and continue checking other rows (blocks).

4.2 If the current state column and the goal state column are not the same, then it means the block is not yet at the right place, then it will go further to find out the distance in the row compared to the goal state condition.

There are two cases that needed to be considered:
- 4.2.1 H is holding something (the H column of the row is 1)
- 4.2.2 H is not holding anything (The H column of the row is 0)

For case 4.2.1 'H is holding something', it has two sub-cases:
- 4.2.1.1 If the block on H is the block of the current checking row
- 4.2.1.2 If the block on H is not the block of the current checking row

For cases 4.2.2 'H is not holding anything' (Empty hand), it has no sub-cases.

5. Thus, from step 4, there are 3 groups of conditions:

- 5.1 'H is holding something' and
'the block on H is the block of the current checking row)

- 5.2 'H is holding something' and
'the block on H is not the block of the current checking row)

- 5.3 'H is not holding anything (Empty hand)'

For each of these cases, we still need to further expand them into two sub-cases:
- (1) the current checking block can move to the destination column directly
(the current checking block is movable **<u>and</u>** the destination block can be moved to)

- (2) the current checking block cannot move to the destination column directly
(the current checking block is not movable **<u>or</u>** the destination block cannot be moved to)

6. If the current column and the goal column of the row are different. According to the 6 groups of conditions, the distances will be estimated differently. Return total distance.

Table 2. Distance Heuristic evaluation idea

In Step 4.2, if we find that the current column and the goal column are not the same, the first thing we want to know is: whether the robotic arm is holding something (or it is in empty-hand status). The reason for considering this is that if the robotic arm is at the status of holding some block/tower, then we have to deal with the block/tower it is holding first, i.e. before putting down the block/tower, the robotic arm will not be free to do any other operations. If the robotic arm is in the empty hand status, then it can move a block/tower to get closer to the goal state directly because the hand is currently free. That is why we separate Step 4.2 into two sub-cases.

Then in the case 4.2.1 ‘H is holding something’, we further decompose it into two subcases. This time our concern is whether the block on the robotic arm is the same row that the algorithm is currently checking. This is also necessary because if the row we are checking is exactly the block held by the robotic arm, this implies that at this row we can directly move the block closer to the goal state. However, if the block that the robotic arm is holding is not the row we are currently checking (but the row we are checking is not perfect yet, still need to be moved), this means the robotic arm has to deal with the block it is holding first, after finishing that operation, then it can come back to handle the block which is of the row we are checking, so these two cases are different in one step away.

In Step 5 in Table2, we mentioned that the 3 groups of conditions that we just decomposed still had to break down into 2 sub-cases, i.e. totally 6 groups of conditions. The reason we need to further expand these conditions is that we need to find out if the block of the row we are currently checking can be moved to the destination column (location) directly or not. If it can, we do not need to add extra steps for it to move. However, if there are some obstacles preventing the robotic arm to move the block to the destination (i.e. the destination is not clear or the destination block is on top of the block we are currently checking), then we also need to evaluate how many steps we need to remove those obstacles so that we can move the block of the row we are checking. We can understand this into two sub-conditions but connecting by a logical-and: the block of the current checking row is movable **and** the destination location can be moved to. If both of these cases are true, we can say that the block of the current checking row can be moved to the destination column directly. Otherwise, if either case is false, then it means that the block cannot be moved to the destination in one step. Thus, we have the following 6 groups of conditions that need to be evaluated differently (See Table 3).

Distance Heuristic conditions of different columns in the checking rows and their evaluations:

Case 1 'H is holding something' and
'the block on H is the block of the current checking row' and
'the current checking block can be moved to the destination column directly'

➔ Add 1 to the total distance.

Case 2 'H is holding something' and
'the block on H is the block of the current checking row' and
'the current checking block cannot be moved to the destination column directly'

➔ Add (number of the blocks on top of the destination block) * 2 to the total distance.

Case 3 'H is holding something' and
'the block on H is not the block of the current checking row' and
'the current checking block can be moved to the destination column directly'

➔ Add 2 to the total distance.
(Add another 1 if the block on top of the one on H is the current checking block)

Case 4 'H is holding something' and
'the block on H is not the block of the current checking row' and
'the current checking block cannot be moved to the destination column directly'

➔ Add (number of the blocks on top of the current checking block
+ number of the blocks on top of the destination block + 1) * 2 + 1
to the total distance.
(Add another 1 if the block on top of the one on H is the current checking block)

Case 5 'H is not holding anything' and
'the current checking block can be moved to the destination column directly'

➔ Add 2 to the total distance.

Case 6 'H is not holding anything' and
'the current checking block cannot be moved to the destination column directly'

➔ Add (number of the blocks on top of the current checking block
+ number of the blocks on top of the destination block + 1) * 2 to the total distance.

Table 3. The 6 conditions of the block not identical to the goal state in the Distance Heuristic evaluation

In Case 1, if H is holding the block we are currently checking and we know that the current block can be moved to the destination block directly without any barrier, then the distance of this row will be 1, because the block is held by the robotic arm and it can be taken care of immediately.

In Case 2, if the robotic arm is holding the block we are currently checking, but we know that the current block cannot be moved to the destination block directly as it is being held on the robotic arm, this implies that we need to consider how many blocks are on top of the destination block first. For each block on top of the destination block, we need two steps to move it away (i.e. pick up and put it somewhere else). Therefore, the number of blocks on top of the destination block needs to be multiplied by 2.

In Case 3, if the robotic arm is holding a block but it is not the one which we are checking, this means we need to put down the current holding block first. Then it can take care of this current checking block. Hence, this takes an extra one step. Still, we know the current checking block can be moved to the destination directly because it does not have any barrier on the destination block. We just need 1 more step to place the block at the right location. Therefore, the distance to the goal state is 2. There is a special case, if the destination block is on top of the block which the arm is holding, i.e. the robotic arm is holding a tower, but the destination is the top block of the tower, in this case, we have to put the tower down and take an extra step to make the destination block available.

In Case 4, as the robotic arm is holding a block which is not the one that we are checking, it takes one step to put down the holding block. After putting down the block, we know that we cannot move the block to the destination directly. In this case, some block(s) that might be on top of the block we want to move, or on top of our destination block, or both cases can be true. Hence, we have to consider both the obstacles and the block we want to move. As a result, the row distance is the sum of the two types of obstacles and the current one block, then multiply by 2 and plus the one step we mentioned in the beginning. The special case in Case 3 can take place in this condition, we also need to take care of that case as well.

In Case 5, this condition is very simple and straightforward: the robotic arm is free and we can move the block we are currently checking to the destination without any barriers. Hence, we can pick it up and place it at the destination, 2 steps in total.

In Case 6, the robotic arm is free, yet the destination is not clear to allow a block to be moved onto. Hence, we need to consider the blocks that may be on top of the current checking block, and the blocks that may be on top of the destination block, and the one we are going to move. Each of these blocks will take 2 steps to move.

With the Distance Heuristic evaluation function implemented, whenever given a state as input, it can provide an estimation of the distance from that state to the goal state. This is very useful because we can evaluate the current state we are at, and then when we travel to the new state, we can compare the new distance value with the previous one so that we can know whether we are getting closer to the goal state.

Moreover, the Distance Heuristic evaluation function is the core part of our Distance Heuristic Search algorithm. With the Distance Heuristic evaluation function, we can compare multiple resulting states of a valid move and pick the best action among all valid ones that lead to a state closest to the goal state. Our search algorithm is first to expand all the valid actions into their resulting states. As a blocks-world problem is non-deterministic, we considered the cases where the operations may result in multiple outcomes. Hence, all the resulting states of the valid moves at the current state will be added to a list, which are paired with their corresponding actions. We then compare all the resulting states by applying the Distance Heuristic evaluation function. The algorithm will determine the resulting state that has the shortest distance to the goal state and check which action can lead to that resulting state. That action is thus the best one among all the valid actions that can lead to the optimal next state, which is the closest to the goal state. (If there are multiple resulting states that have the same distance value to the goal state, then all the corresponding actions will all be taken into consideration, and one of them will be chosen by using a random function.

```
Distance Heuristic Search Algorithm

1. Use the valid action mask to find out all the valid moves at the current state

2. For each valid move in the possible valid move set
   check the action has 1 / 2 outcome(s) (because blocks world is non-deterministic)
   generate all the possible resulting state(s) of the action
   bind the resulting states with the corresponding action as a pair to add to a list

3. Initialize the minimum distance value to 999
   Initialize the minimum distance index to -1

4. For each resulting state in the (resulting state, action) list
   use the Distance Heuristic evaluation function to check the distance to the goal state
   if the distance to the goal state of that state is less than the minimum distance value
   then update the minimum distance value and the minimum distance index

5. Find out the best resulting state that has the shortest distance to the goal state
   Check from the (resulting state, action) list to see which action leads to the state
   That is to say, that action can lead to the best state,
   so the algorithm will return this action as the search result.

( If there are multiple states which have the same distance value to the goal state, they are all
added to a list to be candidates, and one of them will be chosen by Numpy random choice
function )
```

Figure 19. Distance Heuristic Search Algorithm

4.8.1 Epsilon Greedy trigger

At first, we put the Distance Heuristic search guide in the same place to replace Epsilon Greedy. Instead of randomly picking one valid move, we use the search guide algorithm to determine the action that can lead to the best resulting state from the current state. It works better than the Epsilon Greedy because the previous version just randomly picks one valid move, and the one it picks may be a very poor choice. But with the help of the Distance Heuristic search guide, whenever it is triggered, it is guaranteed to pick the best action among all actions applicable to the current state.

Although the Distance Heuristic can pick the optimal action among all at a specific state, we still keep the epsilon parameter $\boldsymbol{\varepsilon}$ to be 1%. We do not want to set the probability too high. The reason is that if it happens too often, e.g. 90%, the result will become the Distance Heuristic search algorithm itself, not the reinforcement learning. Compared to the reinforcement learning model, the Distance Heuristic search algorithm is too expensive. Therefore, we just want it to occur occasionally in the learning process. Even it just happens occasionally, it is very helpful and it makes the learning process much faster than before.

4.8.2 Repeated moves trigger

The Distance Heuristic search guide was triggered by Epsilon Greedy, which was randomly triggered. Hence, it was not stable, i.e., it might not be triggered at the time when it is really needed. We wanted to design another way to replace the Epsilon Greedy trigger method, to make better use of the Distance Heuristic search guide, i.e., we only use it whenever it is necessary. Although it is still occasionally triggered, it is triggered just in time as needed.

As we mentioned earlier, the reinforcement learning model may run into dead loops. That is the reason why we added the Epsilon Greedy approach. Whenever a dead loop occurs, it is necessary to help the model to get out of the dead loop. For this reason, we came up with an approach to trigger the Distance Heuristic search guide rather than depending on that unstable 1% chance.

At the beginning, we use a Boolean flag and an integer to keep track of the consecutive actions. In general, in a blocks-world problem, if we apply different actions, the environment will result in different new states. We have discussed the state space complexity of the blocks-world problem in the previous sections. The number of states is large and during the problem-solving process in an episode, it is very unlikely to repeatedly stay in the same state because it should continuously travel to the next states. While in different states, the set of the valid moves are not the same, and the Q-values for different states are also distinct. Therefore, if the agent keeps choosing the specific few actions repeatedly, it is abnormal. It can imply that the environment is getting stuck in a few particular states, and then in those particular states, the agent repeats to choose the few corresponding actions. As a consequence, it is neither moving closer to the goal state nor learning any meaningful feedback from the states and actions. That is the time we want the Distance Heuristic search guide to intervene.

The condition we set in our program was, if there are no more than 3 moves in consecutive 10 attempts, we judge that the model has fallen into a dead loop, or at least into an unhealthy situation. We use this repeated move detection to trigger the Distance Heuristic search guide. We do not need to Distance Heuristic when the model is learning on the right track, and whenever there is a problem, the search guide shows up to help at once. Therefore, this is a fairly good improvement over the previous versions.

4.8.3 Determine the non-goal state rewards

Another aspect that the Distance Heuristic guide contributed was the reward scheme. In the previous sections, we briefly introduced our early version of the reward scheme. When the environment successfully reaches the goal state, it will receive +100 reward, but for the non-goal states, it just receives 0 rewards or -1 rewards. If we just have to solve 1 or 2 – step problems, then it is fine.

However, if our goal is to solve more complicated problems, then this reward scheme is undesirable. Imagine that we have a 10-block problem that needs nearly 30 steps to get to the goal state, the model can try thousands or even over a million time steps. Without any guidance, it is likely that the algorithm cannot reach the goal state. If it does not reach the goal state, it will not receive any positive rewards. For the large number of non-goal states, if we give -1 rewards, it will keep penalizing all the possible moves it can try as long as it has not successfully reached the goal state. Or if we give 0 rewards for the non-goal states, although it receives no penalties for trying, it does not receive any positive reward either. Without future rewards and expected gains, the reinforcement learning model can hardly learn anything. This is a critical problem.

At the beginning of the Distance Heuristic section, we concisely discussed that what the Distance Heuristic evaluation could be used for and this was one game-changing point - the Distance Heuristic evaluation function can be used to evaluate the distance from a state to the goal state. As a consequence, we can find out the current distance to the goal state, and the previous distance to the goal state. By this comparison, we can know whether the last move made the state closer or farther to the goal state.

Previously, the problem was, in a non-goal state, the model did not know whether a move was good or bad, it had no idea so it could not give any meaningful reward according to the chosen action. After we have the help from the Distance Heuristic evaluation function, we can compare the distance change between the current state and the previous state to have an estimate of how good or bad the last action we picked. In theory, we should be able to give a good reward when the agent performs a good move and give a penalty when the agent makes a wrong decision. We implemented the reward scheme, but the result was somehow not very successful. It was interesting to find out that the model was very happy to receive good rewards. Hence, it intentionally performs a bad move to go farther so that the next time it can receive a good reward by moving back. Based on such experience, we designed a new version of reward giving system: If the agent chooses a move that is closer to the goal state (i.e. a good move) or a move that is acceptable (i.e. no change or one step farther from the last state, this is because we consider the effect of non-deterministic outcomes, even a good action may end up in dropping the block on the table), we give 0 rewards for these cases. However, if the action chosen by the agent made the current state even farther away from the goal state compared to the previous distance, then we give it -1, -2, -3, -4, -5 rewards depending on how much distance farther away compared to the last state.

PART5: EXPERIMENT

5.1 Train for specific problems and find the optimal solutions

In reinforcement learning, e.g., the classical Cart-Pole balancing, playing a chess game, training a robot or virtual animal to walk, or training a model to play a game, such kind of learning problem is to give it a specific task to learn, i.e., let it try and learn from the previous experience, and then gradually master the task. Therefore, in this study, although our ultimate goal is to solve all the blocks-world problems, we start from solving specific blocks-world problems.

We designed 3 distinct problems for each group of the N-block problems: 3-block problems, 5-block problems, 10-block problems, 15-block problems, 20-block problems.

For each question, because of the non-deterministic outcome effect, the generated plans may look different every time. In the learning process, we keep track of the lengths of the plans found by the model. If a shortest plan is found, and then in the next 10 plans there are no plans shorter than it, then we stop the training and assume the weights are nearly converged. (Because the non-deterministic outcome may make a good move result in a bad state (and receive a penalty), so instead of setting a range to assume whether the weights are converged, we use this consecutive shortest plan to determine because this way has considered the tolerance of the non-deterministic states.)

In the results, the plans do not include the leaf states, i.e., no path leading the untraveled leaf states in that episode to the goal. The following results are just the plans for the readers to understand how our approach works and what results that it can find. In practice, whenever the environment falls into a leaf state in the non-deterministic world, since we use reinforcement learning to train, we can send the leaf state as the input to the agent, the Artificial Neural Network can take care of it instantly. Hence we can say the neural network (with the trained weights) itself is a strong plan for the problem.

5.1.1 3-block problems

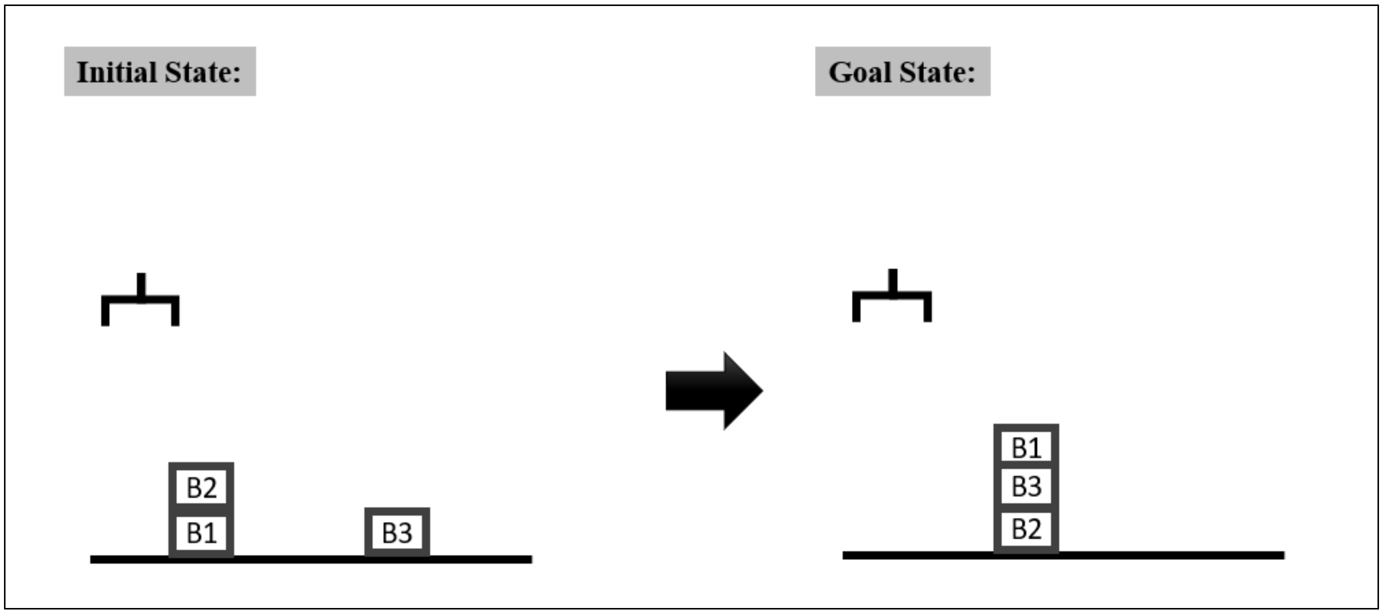


Figure 20a. 3-block Problem 1

Result:

```
Number of steps: 5

------------------------------------------------------------
| pickup b2 from b1 (dropped on table)                      |
| pickup b1 from table                                      |
| put b1 on b3                                              |
| pickup b3 from table                                      |
| put b3 on b2                                              |
------------------------------------------------------------
```

Figure 20a. Result of 3-block Problem 1

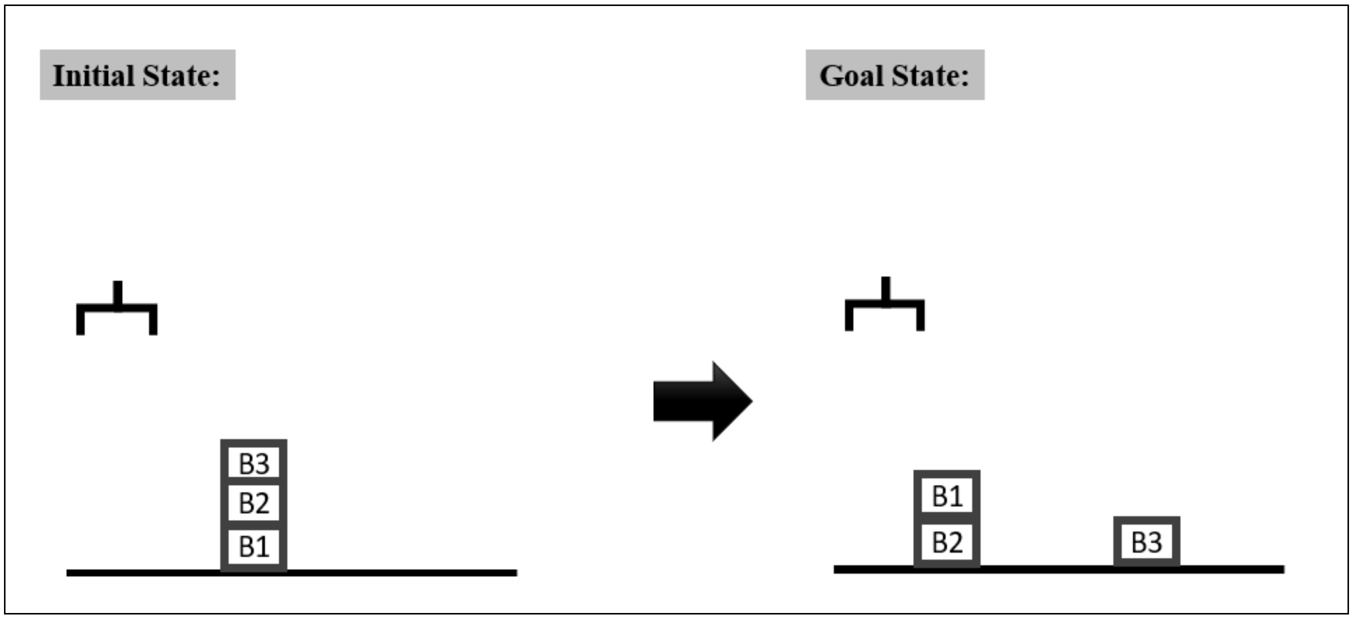
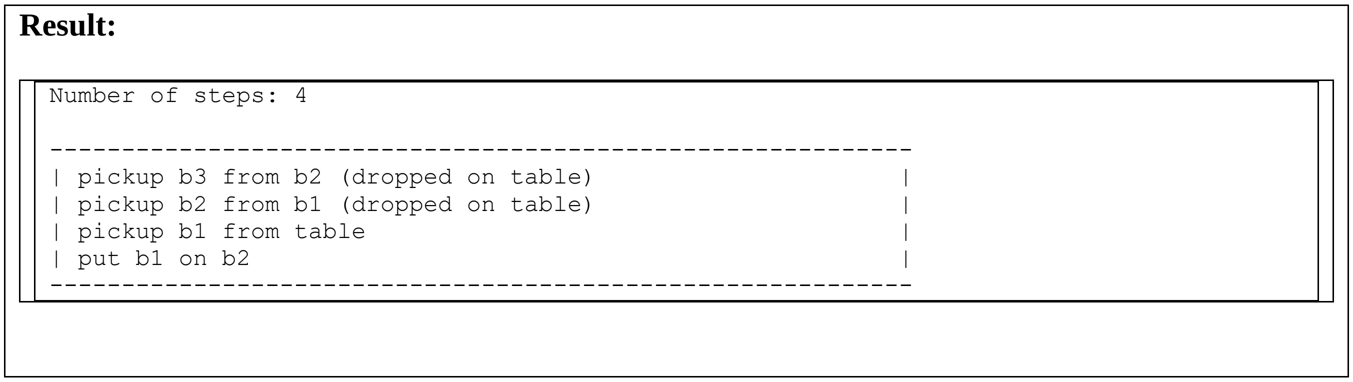


Figure 21a. 3-block Problem 2



Result:

```
Number of steps: 4

------------------------------------------------------------
| pickup b3 from b2 (dropped on table)                     |
| pickup b2 from b1 (dropped on table)                     |
| pickup b1 from table                                     |
| put b1 on b2                                             |
------------------------------------------------------------
```

Figure 21b. Result of 3-block Problem 2

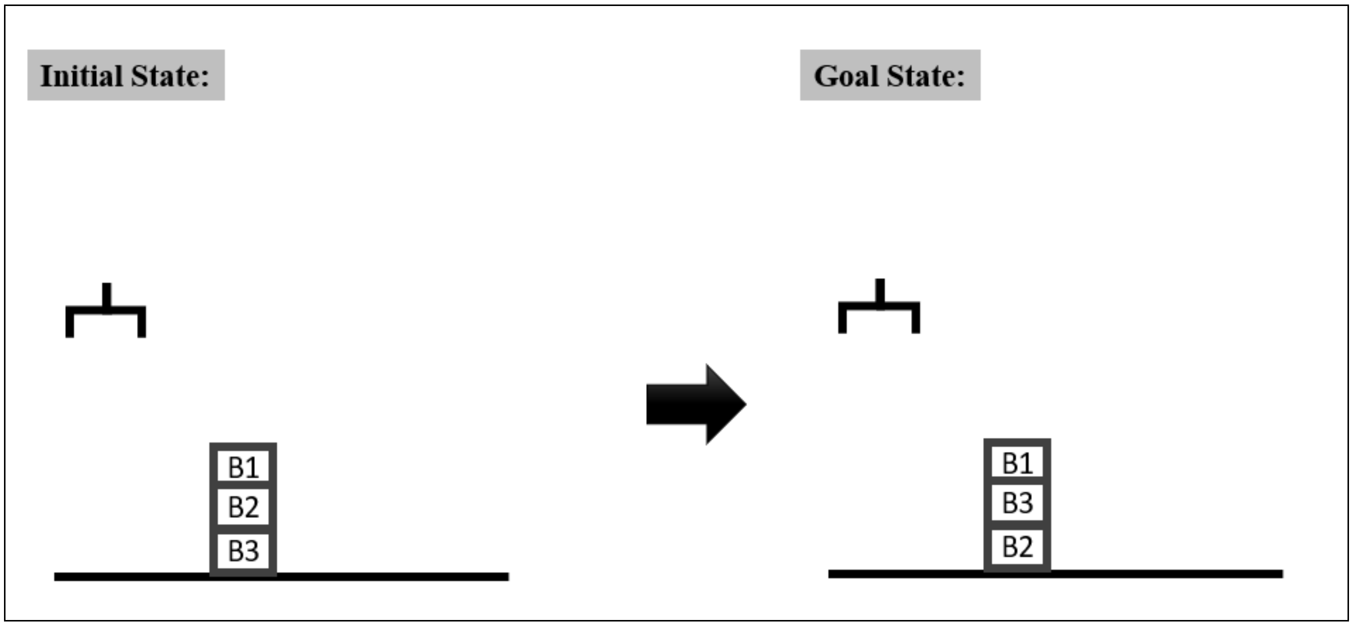
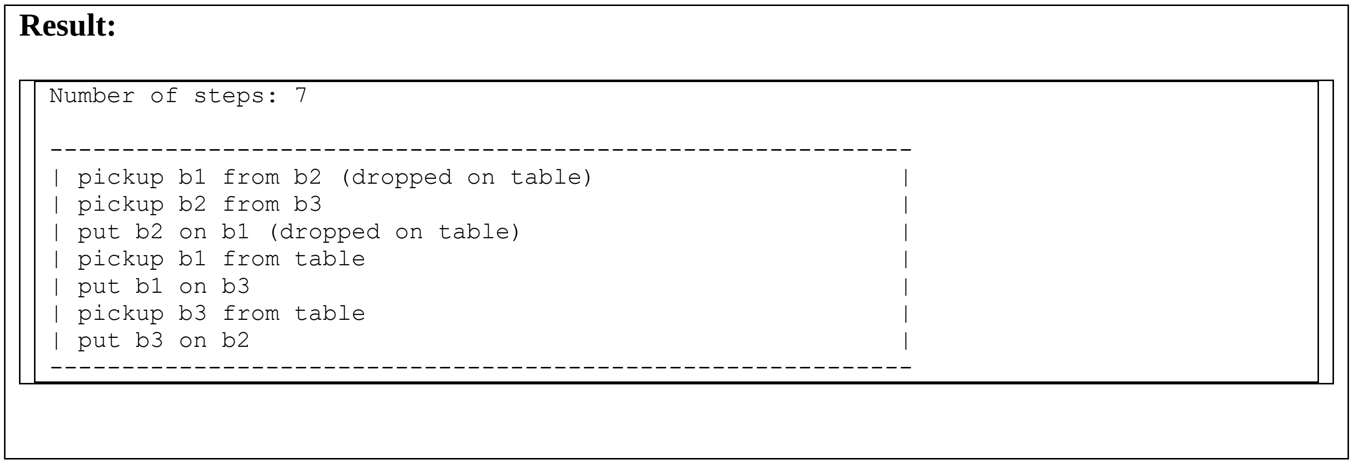


Figure 22a. 3-block Problem 3


Result:

```
Number of steps: 7

-------------------------------------------------------------
| pickup b1 from b2 (dropped on table)                      |
| pickup b2 from b3                                         |
| put b2 on b1 (dropped on table)                           |
| pickup b1 from table                                      |
| put b1 on b3                                              |
| pickup b3 from table                                      |
| put b3 on b2                                              |
-------------------------------------------------------------
```

Figure 22b. Result of 3-block Problem 3

5.1.2 5-block problems

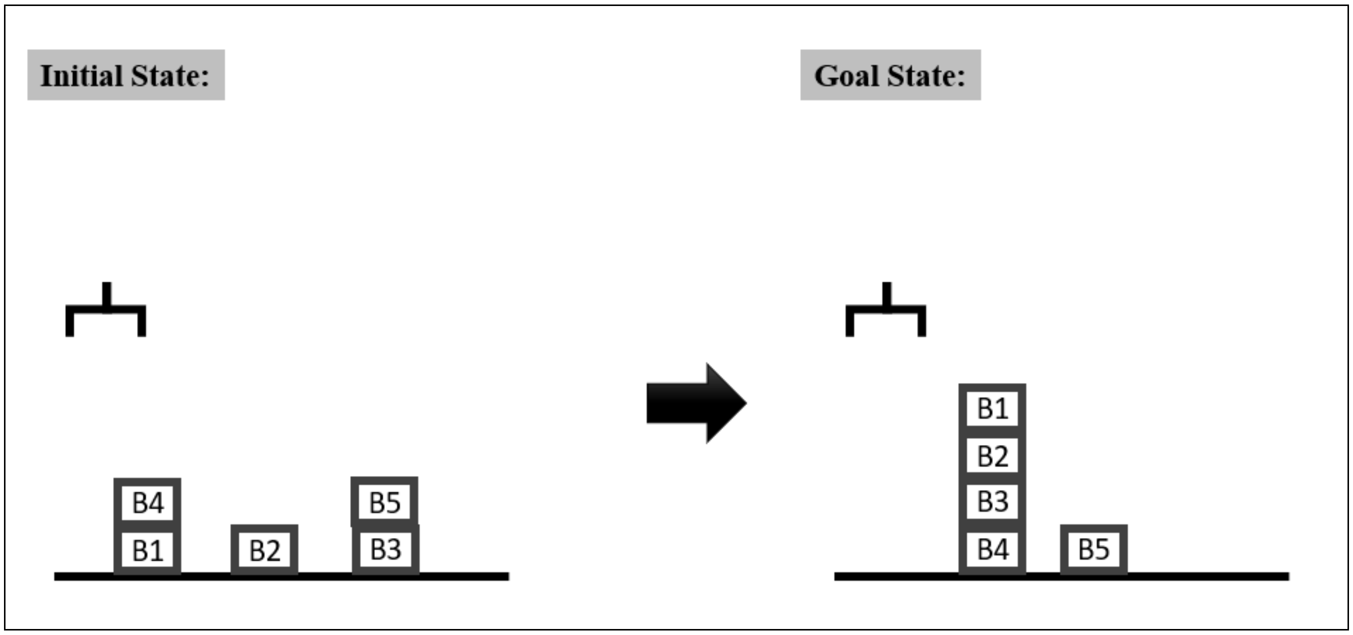


Figure 23a. 5-block Problem 1

Result:

```
Number of steps: 8

------------------------------------------------------------
| pickup b5 from b3 (dropped on table)                     |
| pickup b4 from b1 (dropped on table)                     |
| pickup b2 from table                                     |
| put b2 on b3                                             |
| pickup b3 from table                                     |
| put b3 on b4                                             |
| pickup b1 from table                                     |
| put b1 on b2                                             |
------------------------------------------------------------
```

Figure 23b. Result of 5-block Problem 1

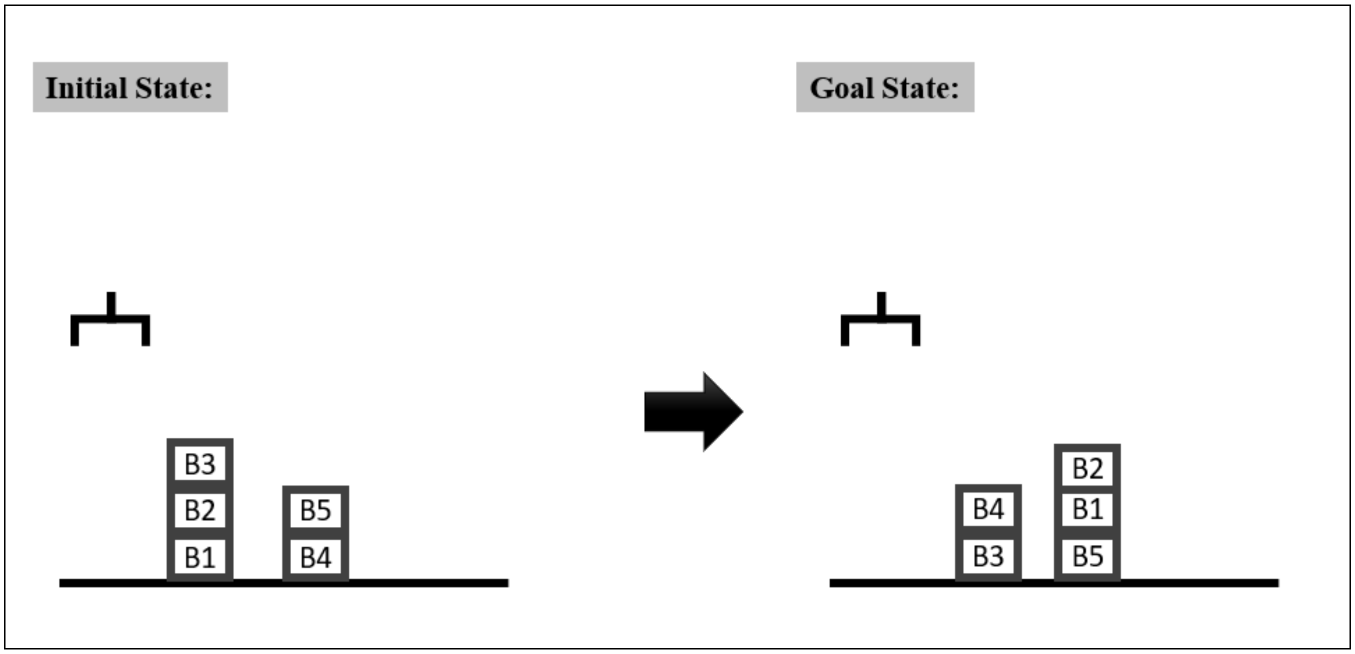


Figure 24a. 5-block Problem 2

Result:

```
Number of steps: 6

------------------------------------------------------------
| pickup b3 from b2 (dropped on table)                     |
| pickup b5 from b4 (dropped on table)                     |
| pickup b4 from table                                     |
| put b4 on b3                                             |
| pickup b1 from table                                     |
| put b1 on b5                                             |
------------------------------------------------------------
```

Figure 24b. Result of 5-block Problem 2

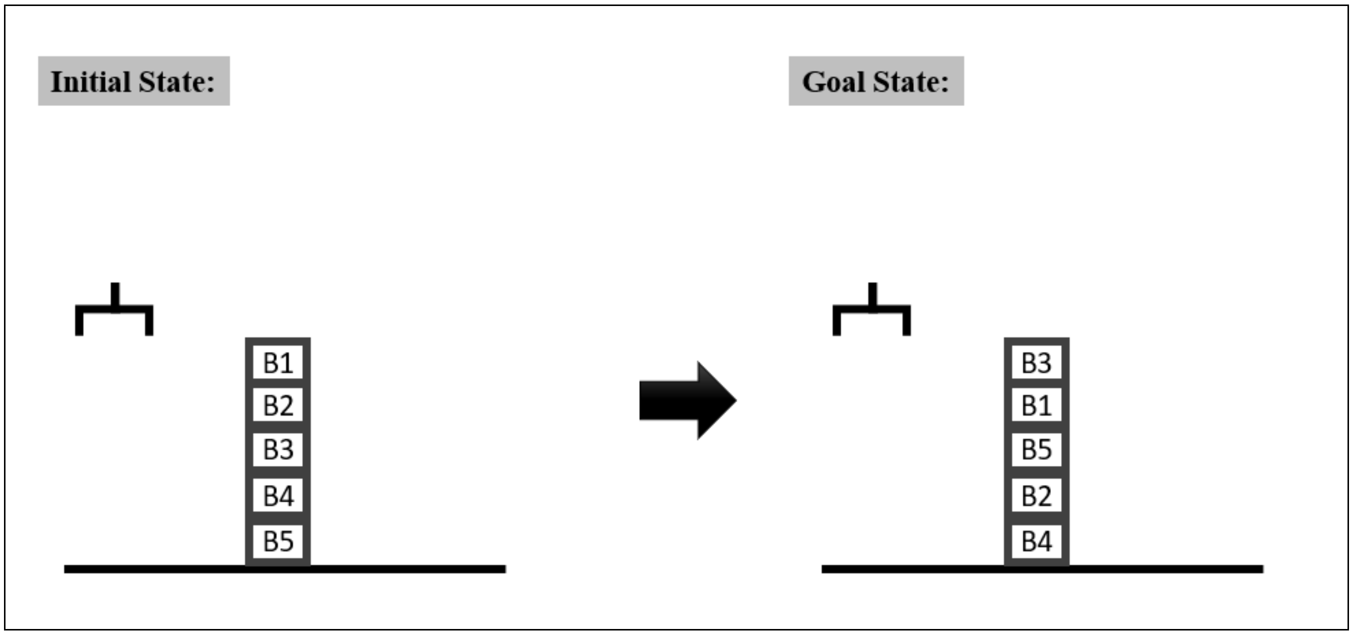


Figure 25a. 5-block Problem 3

Result:

```
Number of steps: 14

------------------------------------------------------------
| pickup b2 from b3                                        |
| put b2 on table                                          |
| pickup b3 from b4                                        |
| put b3 on b1                                             |
| pickup b1 from b2                                        |
| put b1 on table                                          |
| pickup b4 from b5                                        |
| put b4 on b3 (dropped on table)                          |
| pickup b2 from table                                     |
| put b2 on b4                                             |
| pickup b5 from table                                     |
| put b5 on b2                                             |
| pickup b1 from table                                     |
| put b1 on b5                                             |
------------------------------------------------------------
```

Figure 25b. Result of 5-block Problem 3

5.1.3 10-block problems

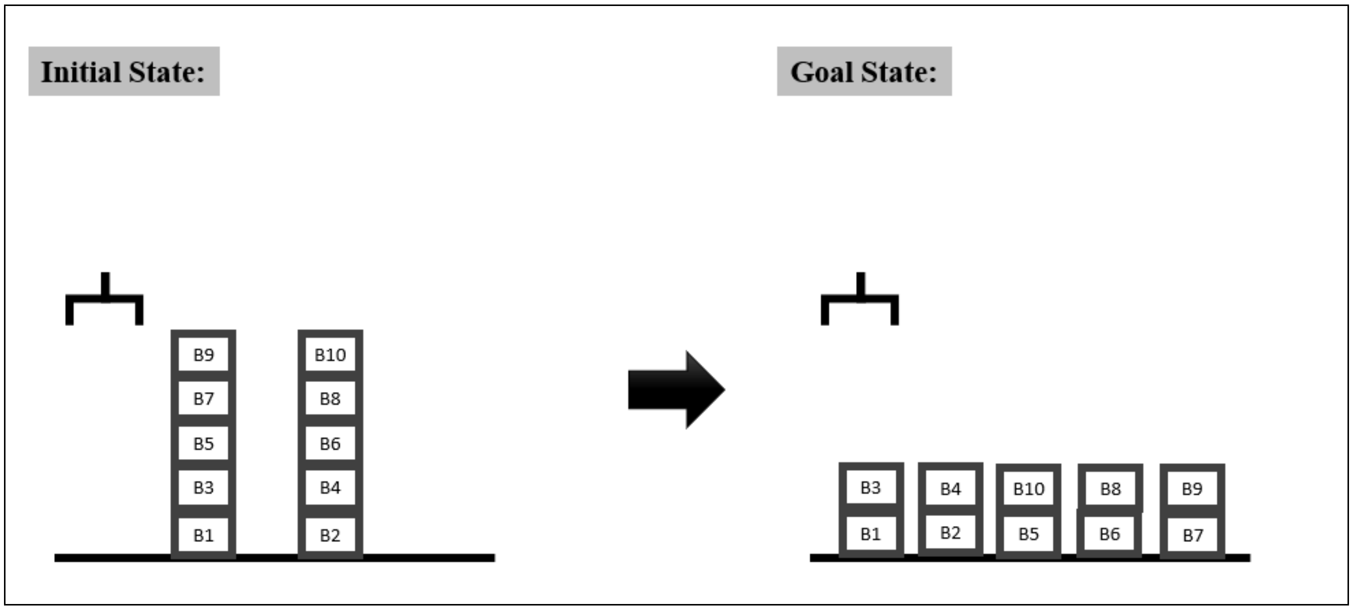


Figure 26a. 10-block Problem 1

Result:

```
Number of steps: 6

------------------------------------------------------------
| pickup b7 from b5                                        |
| put b7 on table                                          |
| pickup b10 from b8                                       |
| put b10 on b5                                            |
| pickup b6 from b4 (dropped on table)                     |
| pickup b5 from b3 (dropped on table)                     |
------------------------------------------------------------
```

Figure 26b. Result of 10-block Problem 1

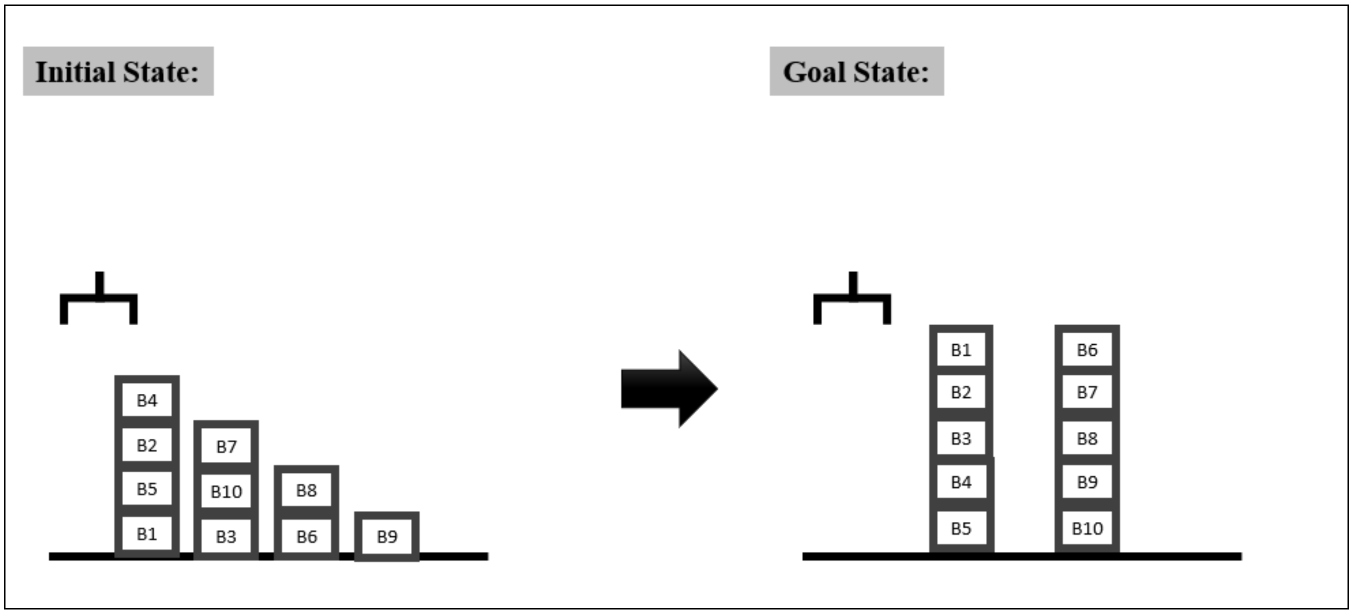


Figure 27a. 10-block Problem 2

Result:

```
Number of steps: 23

------------------------------------------------------------
| pickup b2 from b5 (dropped on table)                      |
| pickup b10 from b3                                        |
| put b10 on table                                          |
| pickup b4 from b2 (dropped on table)                      |
| pickup b7 from b10 (dropped on table)                     |
| pickup b5 from b1 (dropped on table)                      |
| pickup b8 from b6 (dropped on table)                      |
| pickup b6 from table                                      |
| put b6 on b7                                              |
| pickup b9 from table                                      |
| put b9 on b10                                             |
| pickup b1 from table                                      |
| put b1 on b2                                              |
| pickup b4 from table                                      |
| put b4 on b5                                              |
| pickup b3 from table                                      |
| put b3 on b4                                              |
| pickup b2 from table                                      |
| put b2 on b3                                              |
| pickup b8 from table                                      |
| put b8 on b9                                              |
| pickup b7 from table                                      |
| put b7 on b8                                              |
------------------------------------------------------------
```

Figure 27b. Result of 10-block Problem 2

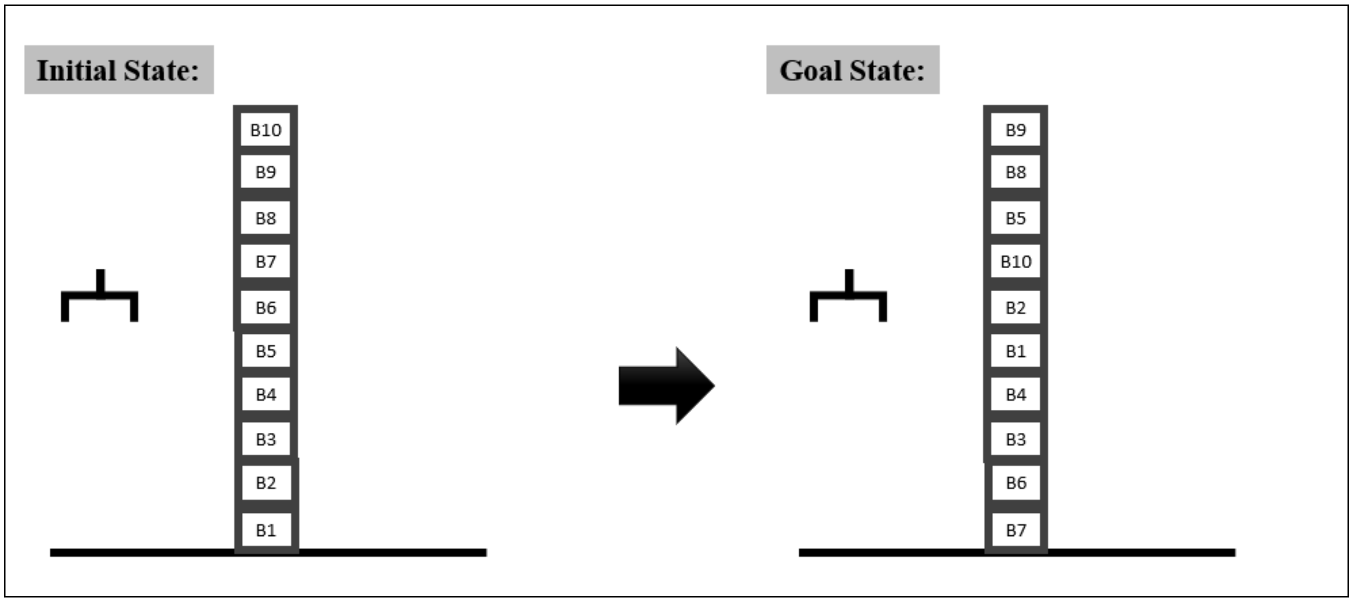


Figure 28a. 10-block Problem 3

```
Number of steps: 22

------------------------------------------------------------
| pickup b9 from b8 (dropped on table)                      |
| pickup b7 from b6                                         |
| put b7 on table                                           |
| pickup b5 from b4 (dropped on table)                      |
| pickup b3 from b2 (dropped on table)                      |
| pickup b8 from b7 (dropped on table)                      |
| pickup b6 from b5                                         |
| put b6 on b7                                              |
| pickup b3 from table                                      |
| put b3 on b6                                              |
| pickup b1 from table                                      |
| put b1 on b4                                              |
| pickup b10 from b9                                        |
| put b10 on b2 (dropped on table)                          |
| pickup b8 from table                                      |
| put b8 on b5                                              |
| pickup b10 from table                                     |
| put b10 on b2                                             |
| pickup b5 from table                                      |
| put b5 on b10                                             |
| pickup b9 from table                                      |
| put b9 on b8                                              |
------------------------------------------------------------
```

Figure 28b. Result of 10-block Problem 3

5.1.4 15-block problems

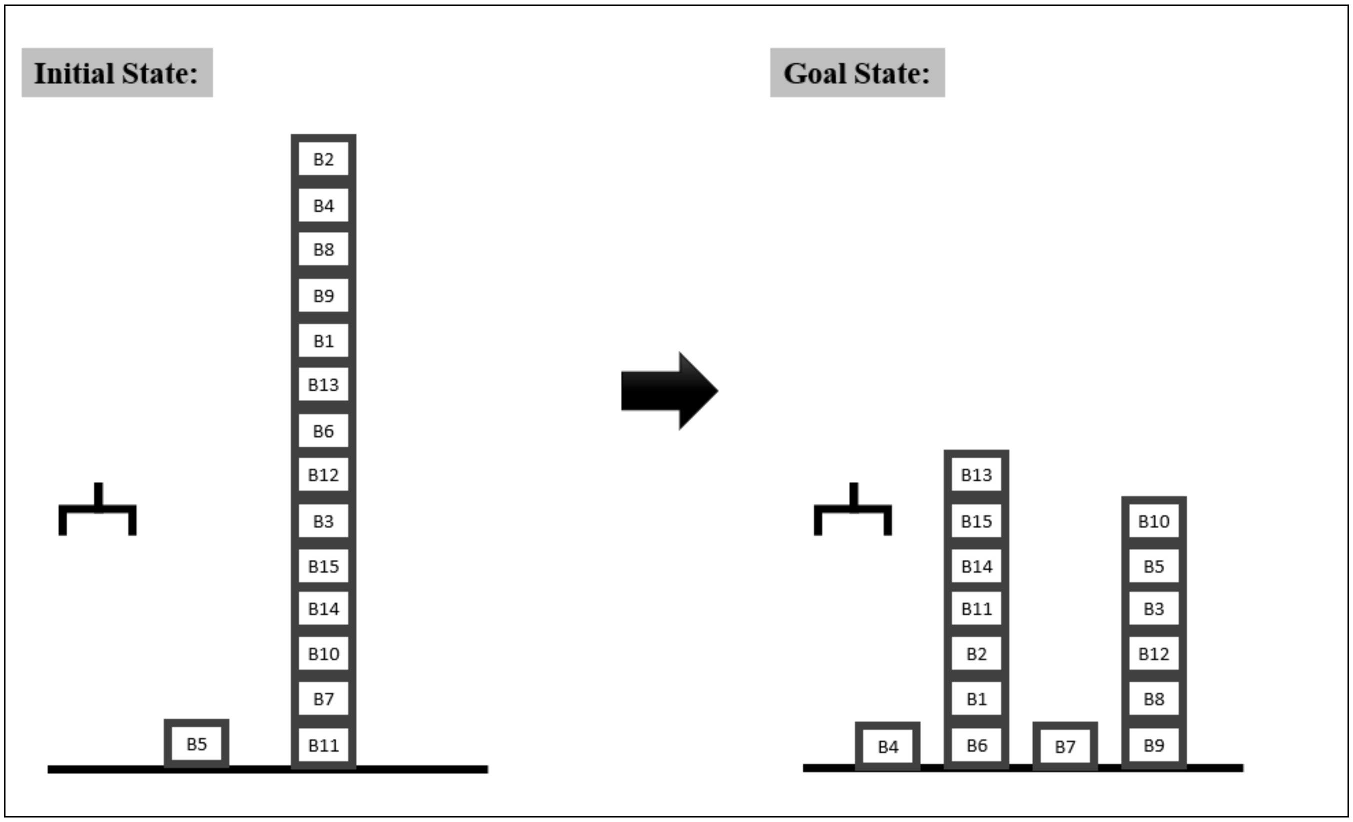
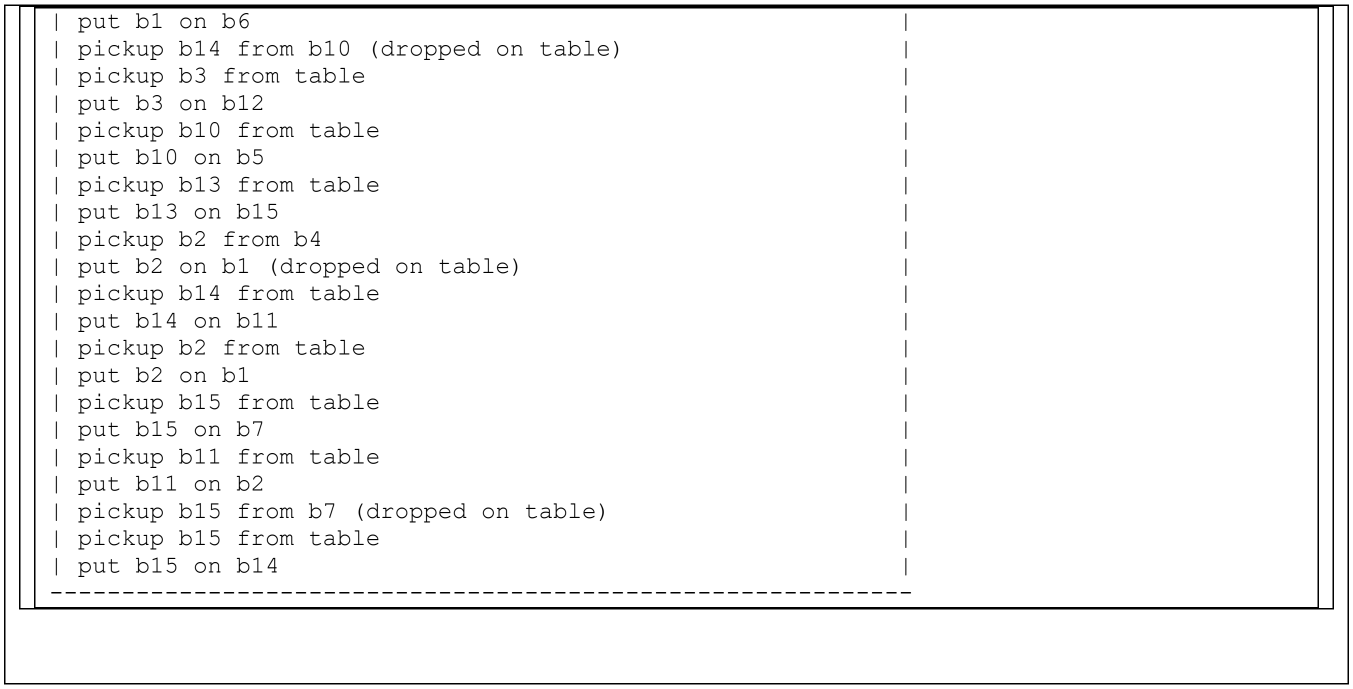


Figure 29a. 15-block Problem 1

Result:

```
Number of steps: 39

------------------------------------------------------------
| pickup b4 from b8                                        |
| put b4 on table                                          |
| pickup b9 from b1 (dropped on table)                     |
| pickup b13 from b6 (dropped on table)                    |
| pickup b12 from b3 (dropped on table)                    |
| pickup b15 from b14 (dropped on table)                   |
| pickup b10 from b7 (dropped on table)                    |
| pickup b3 from b15 (dropped on table)                    |
| pickup b7 from b11 (dropped on table)                    |
| pickup b6 from b12                                       |
| put b6 on b7                                             |
| pickup b12 from table                                    |
| put b12 on b8                                            |
| pickup b5 from table                                     |
| put b5 on b3                                             |
| pickup b1 from b13 (dropped on table)                    |
| pickup b6 from b7 (dropped on table)                     |
| pickup b1 from table                                     |
```



```
| put b1 on b6                                              |
| pickup b14 from b10 (dropped on table)                    |
| pickup b3 from table                                      |
| put b3 on b12                                             |
| pickup b10 from table                                     |
| put b10 on b5                                             |
| pickup b13 from table                                     |
| put b13 on b15                                            |
| pickup b2 from b4                                         |
| put b2 on b1 (dropped on table)                           |
| pickup b14 from table                                     |
| put b14 on b11                                            |
| pickup b2 from table                                      |
| put b2 on b1                                              |
| pickup b15 from table                                     |
| put b15 on b7                                             |
| pickup b11 from table                                     |
| put b11 on b2                                             |
| pickup b15 from b7 (dropped on table)                     |
| pickup b15 from table                                     |
| put b15 on b14                                            |
-------------------------------------------------------------
```

Figure 29b. Result of 15-block Problem 1

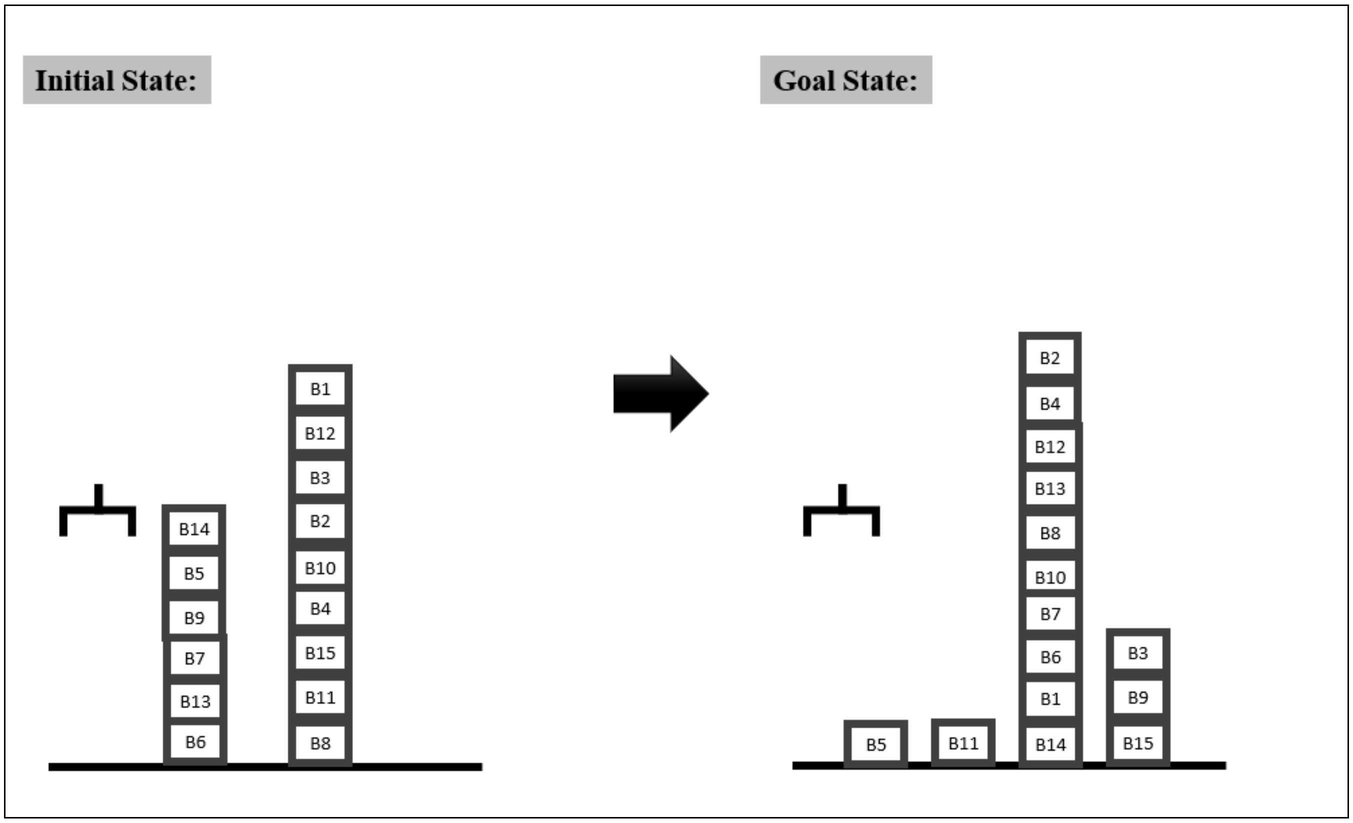


Figure 30a. 15-block Problem 2

Result:

```
Number of steps: 46

------------------------------------------------------------
| pickup b12 from b3                                       |
| put b12 on table                                         |
| pickup b5 from b9                                        |
| put b5 on table                                          |
| pickup b2 from b10 (dropped on table)                    |
| pickup b4 from b15 (dropped on table)                    |
| pickup b7 from b13 (dropped on table)                    |
| pickup b11 from b8 (dropped on table)                    |
| pickup b10 from b4                                       |
| put b10 on b15 (dropped on table)                        |
| pickup b13 from b6                                       |
| put b13 on b8 (dropped on table)                         |
| pickup b1 from b12 (dropped on table)                    |
| pickup b9 from b7 (dropped on table)                     |
| pickup b15 from b11 (dropped on table)                   |
| pickup b3 from b2                                        |
| put b3 on b9                                             |
| pickup b1 from table                                     |
| put b1 on b14                                            |
| pickup b10 from table                                    |
| put b10 on b7                                            |
| pickup b9 from table                                     |
| put b9 on b15                                            |
| pickup b2 from table                                     |
| put b2 on b4                                             |
| pickup b14 from b5                                       |
| put b14 on table                                         |
| pickup b8 from table                                     |
| put b8 on b11                                            |
| pickup b12 from table                                    |
| put b12 on b13                                           |
| pickup b8 from b11                                       |
| put b8 on b11 (dropped on table)                         |
| pickup b4 from table                                     |
| put b4 on b5                                             |
| pickup b6 from table                                     |
| put b6 on b1                                             |
| pickup b7 from table                                     |
| put b7 on b6                                             |
| pickup b4 from b5 (dropped on table)                     |
| pickup b8 from table                                     |
| put b8 on b10                                            |
| pickup b13 from table                                    |
| put b13 on b8                                            |
| pickup b4 from table                                     |
| put b4 on b12                                            |
------------------------------------------------------------
```

Figure 30b. Result of 15-block Problem 2

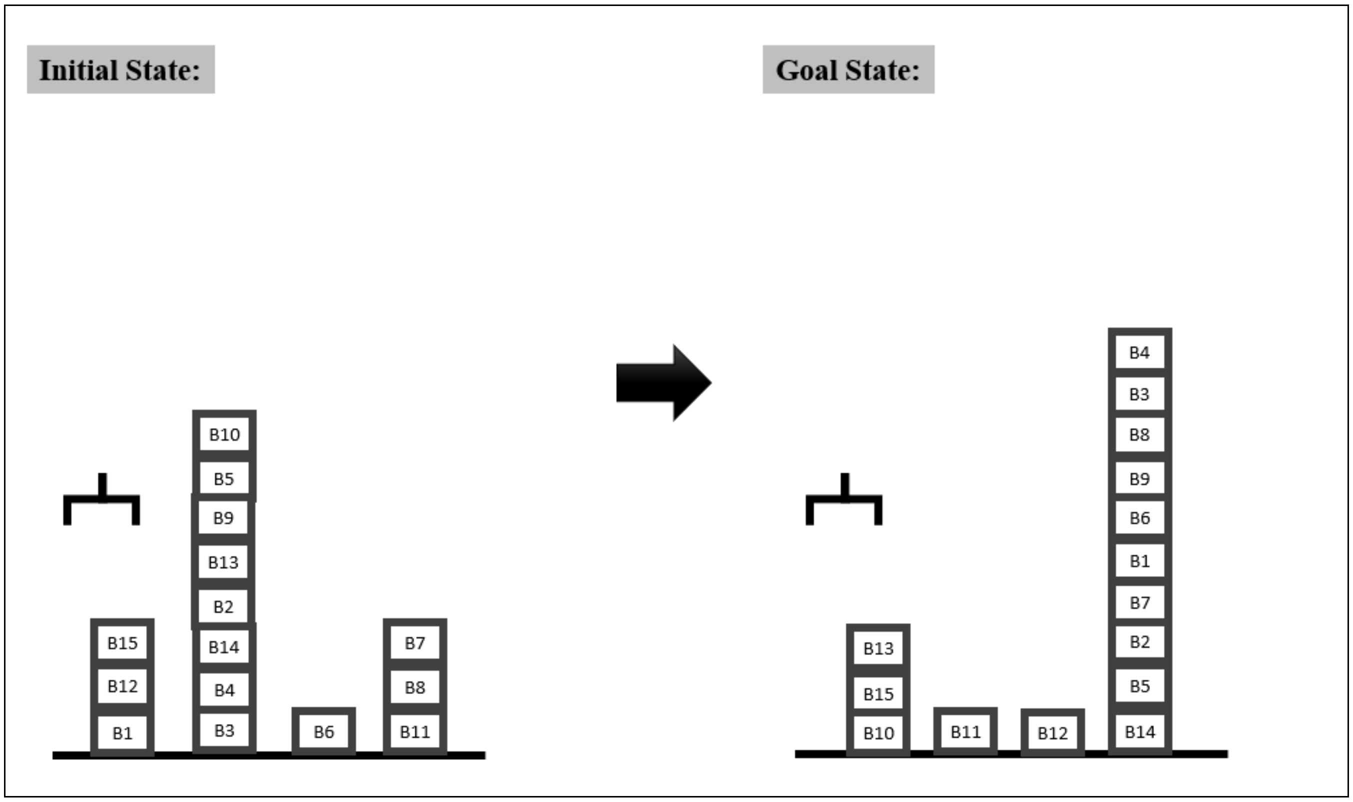
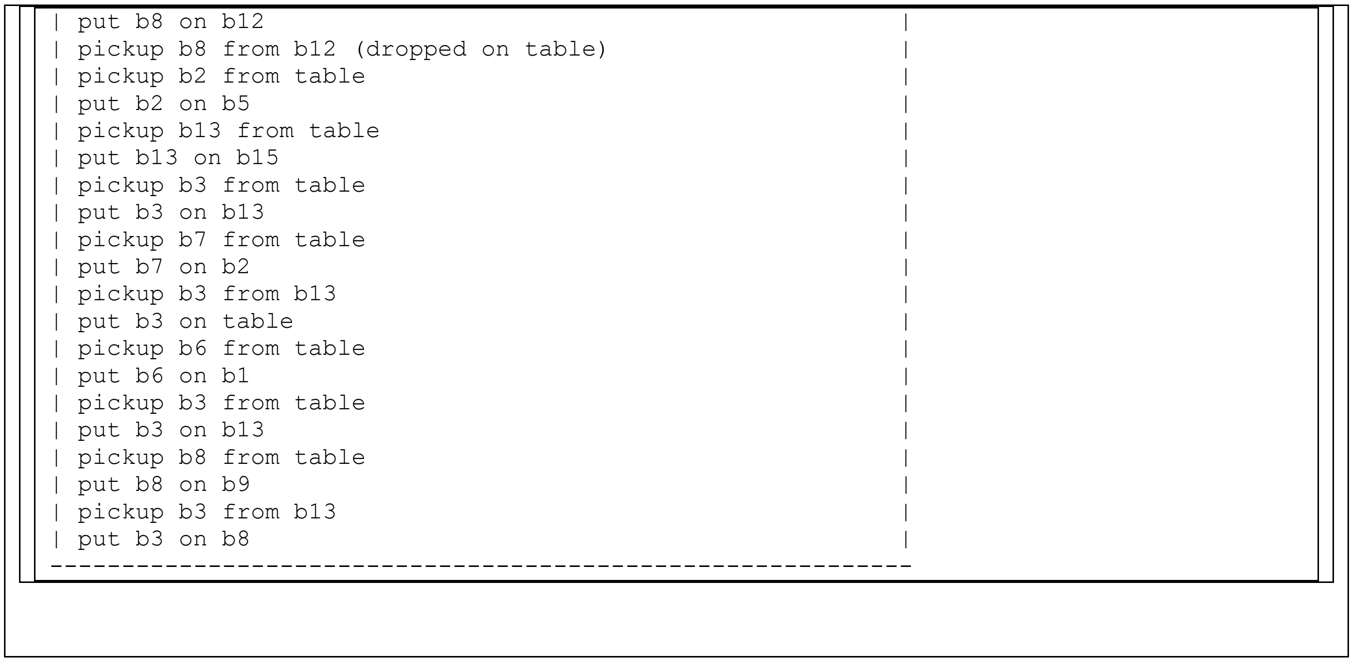


Figure 31a. 15-block Problem 3

Result:

```
Number of steps: 42

------------------------------------------------------------
| pickup b5 from b9                                        |
| put b5 on table                                          |
| pickup b13 from b2                                       |
| put b13 on table                                         |
| pickup b12 from b1 (dropped on table)                    |
| pickup b2 from b14 (dropped on table)                    |
| pickup b10 from b5                                       |
| put b10 on b5 (dropped on table)                         |
| pickup b7 from b8 (dropped on table)                     |
| pickup b14 from b4 (dropped on table)                    |
| pickup b8 from b11 (dropped on table)                    |
| pickup b1 from table                                     |
| put b1 on b7                                             |
| pickup b8 from table                                     |
| put b8 on b11                                            |
| pickup b9 from b13                                       |
| put b9 on b6                                             |
| pickup b15 from b12                                      |
| put b15 on b10                                           |
| pickup b5 from table                                     |
| put b5 on b14                                            |
| pickup b8 from b11                                       |
```



```
| put b8 on b12                                           |
| pickup b8 from b12 (dropped on table)                   |
| pickup b2 from table                                    |
| put b2 on b5                                            |
| pickup b13 from table                                   |
| put b13 on b15                                          |
| pickup b3 from table                                    |
| put b3 on b13                                           |
| pickup b7 from table                                    |
| put b7 on b2                                            |
| pickup b3 from b13                                      |
| put b3 on table                                         |
| pickup b6 from table                                    |
| put b6 on b1                                            |
| pickup b3 from table                                    |
| put b3 on b13                                           |
| pickup b8 from table                                    |
| put b8 on b9                                            |
| pickup b3 from b13                                      |
| put b3 on b8                                            |
-----------------------------------------------------------
```

Figure 31b. Result of 15-block Problem 3

5.1.5 20-block problems

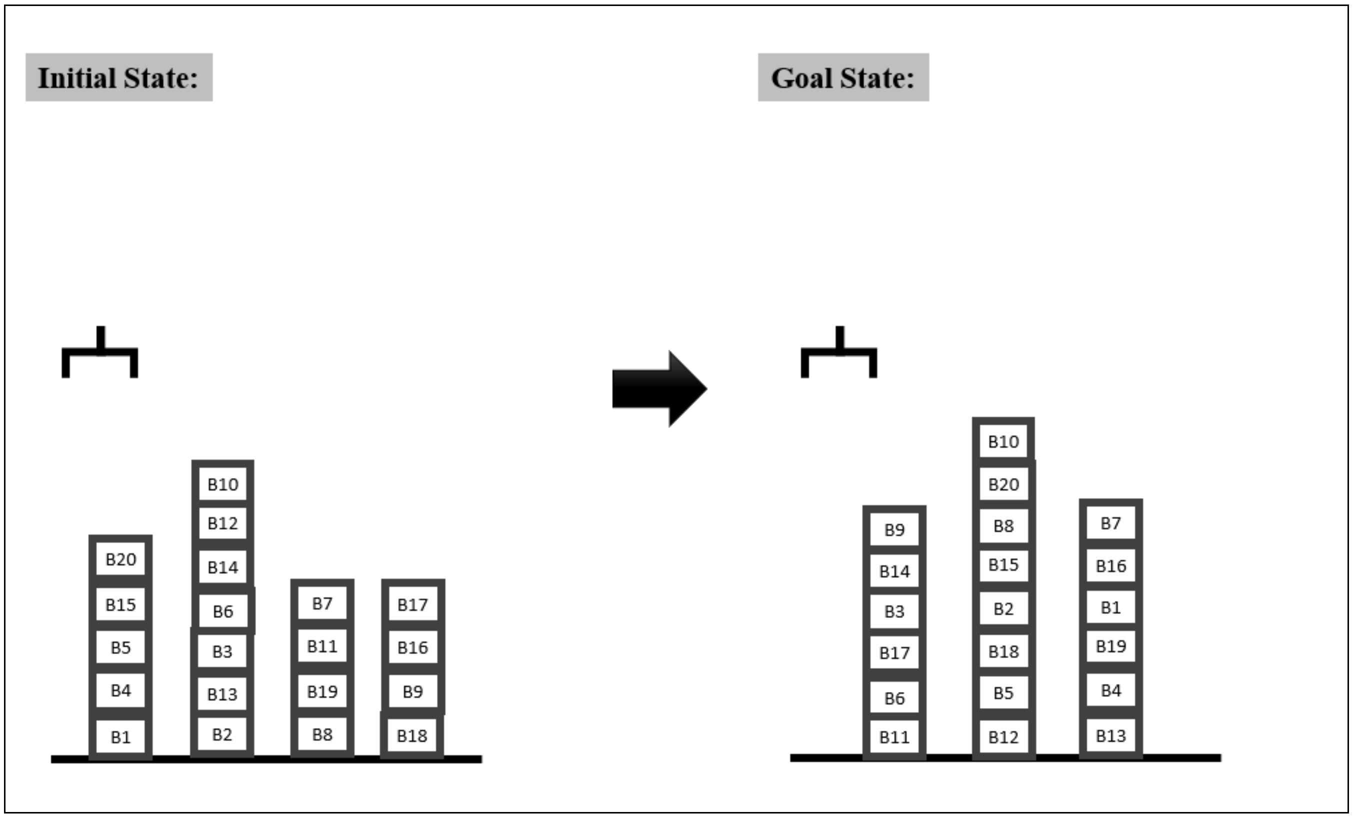


Figure 32a. 20-block Problem 1

Result:

```
Number of steps: 60

------------------------------------------------------------
| pickup b12 from b14 (dropped on table)                  |
| pickup b6 from b3 (dropped on table)                    |
| pickup b15 from b5                                      |
| put b15 on table                                        |
| pickup b16 from b9                                      |
| put b16 on table                                        |
| pickup b7 from b11 (dropped on table)                   |
| pickup b4 from b1 (dropped on table)                    |
| pickup b19 from b8 (dropped on table)                   |
| pickup b13 from b2 (dropped on table)                   |
| pickup b5 from b4 (dropped on table)                    |
| pickup b17 from b16                                     |
| put b17 on b7                                           |
| pickup b14 from b6 (dropped on table)                   |
| pickup b11 from b19                                     |
| put b11 on b1 (dropped on table)                        |
| pickup b9 from b18 (dropped on table)                   |
| pickup b10 from b12                                     |
| put b10 on b9                                           |
| pickup b3 from b13 (dropped on table)                   |
| pickup b20 from b15                                     |
| put b20 on b8                                           |
| pickup b10 from b9                                      |
| put b10 on b9 (dropped on table)                        |
| pickup b18 from table                                   |
| put b18 on b5                                           |
| pickup b17 from b7 (dropped on table)                   |
| pickup b19 from table                                   |
| put b19 on b4                                           |
| pickup b17 from table                                   |
| put b17 on b6                                           |
| pickup b14 from table                                   |
| put b14 on b3                                           |
| pickup b16 from table                                   |
| put b16 on b1                                           |
| pickup b5 from table                                    |
| put b5 on b12                                           |
| pickup b4 from table                                    |
| put b4 on b13                                           |
| pickup b1 from table                                    |
| put b1 on b19                                           |
| pickup b7 from table                                    |
| put b7 on b16                                           |
| pickup b6 from table                                    |
| put b6 on b11                                           |
| pickup b3 from table                                    |
| put b3 on b17                                           |
| pickup b9 from table                                    |
| put b9 on b14                                           |
| pickup b15 from table                                   |
| put b15 on b2                                           |
| pickup b2 from table                                    |
| put b2 on b18                                           |
| pickup b10 from table                                   |
| put b10 on b9                                           |
| pickup b8 from table                                    |
| put b8 on b15                                           |
| pickup b10 from b9 (dropped on table)                     |
| pickup b10 from table                                     |
| put b10 on b20                                            |
-------------------------------------------------------------
```

Figure 32b. Result of 20-block Problem 1

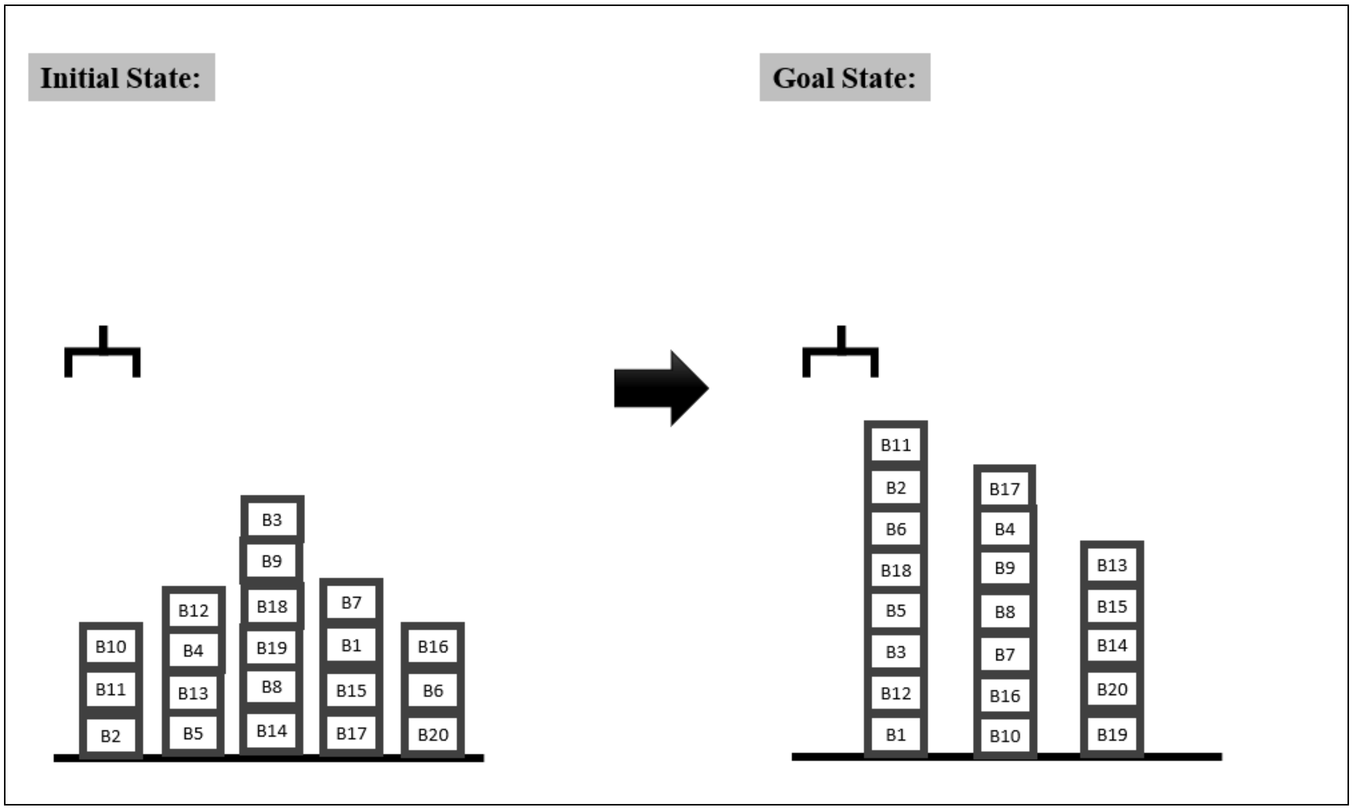


Figure 33a. 20-block Problem 2

Result:

```
Number of steps: 52

-------------------------------------------------------------
| pickup b9 from b18                                        |
| put b9 on table                                           |
| pickup b19 from b8 (dropped on table)                     |
| pickup b4 from b13 (dropped on table)                     |
| pickup b1 from b15                                        |
| put b1 on table                                           |
| pickup b6 from b20                                        |
| put b6 on table                                           |
| pickup b12 from b4 (dropped on table)                       |
| pickup b18 from b19                                         |
| put b18 on b19 (dropped on table)                           |
| pickup b3 from b9                                           |
| put b3 on b12 (dropped on table)                            |
| pickup b7 from b1                                           |
| put b7 on b19 (dropped on table)                            |
| pickup b8 from b14 (dropped on table)                       |
| pickup b13 from b5 (dropped on table)                       |
| pickup b16 from b6 (dropped on table)                       |
| pickup b10 from b11 (dropped on table)                      |
| pickup b7 from table                                        |
| put b7 on b16                                               |
| pickup b16 from table                                       |
| put b16 on b10                                              |
| pickup b18 from table                                       |
| put b18 on b5                                               |
| pickup b15 from b17 (dropped on table)                      |
| pickup b13 from table                                       |
| put b13 on b15                                              |
| pickup b9 from table                                        |
| put b9 on b8                                                |
| pickup b12 from table                                       |
| put b12 on b1                                               |
| pickup b3 from table                                        |
| put b3 on b12                                               |
| pickup b17 from table                                       |
| put b17 on b4                                               |
| pickup b5 from table                                        |
| put b5 on b3                                                |
| pickup b6 from table                                        |
| put b6 on b18                                               |
| pickup b2 from table                                        |
| put b2 on b6                                                |
| pickup b8 from table                                        |
| put b8 on b7                                                |
| pickup b20 from table                                       |
| put b20 on b19                                              |
| pickup b4 from table                                        |
| put b4 on b9                                                |
| pickup b14 from table                                       |
| put b14 on b20                                              |
| pickup b15 from table                                       |
| put b15 on b14                                              |
-------------------------------------------------------------
```

Figure 33b. Result of 20-block Problem 2

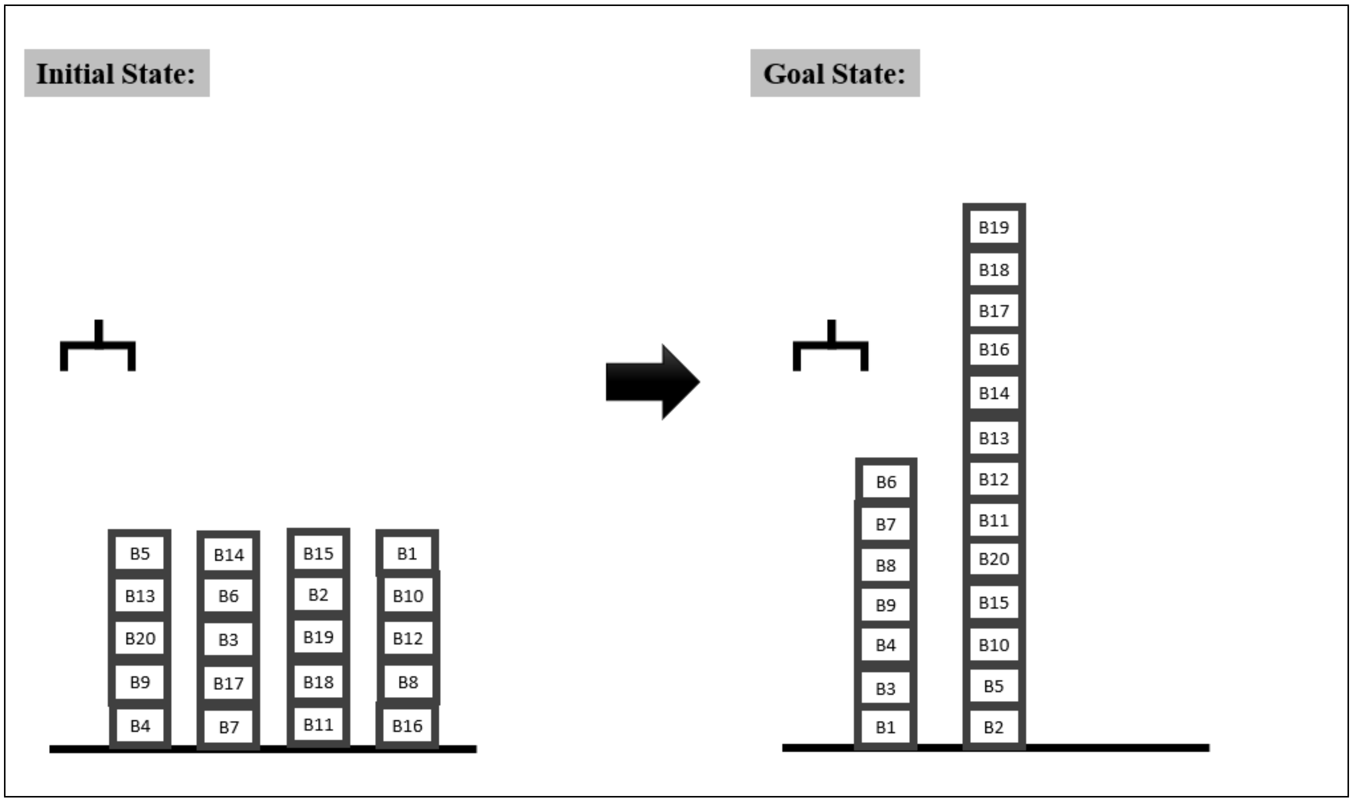


Figure 34a. 20-block Problem 3

Result:

```
Number of steps: 52

------------------------------------------------------------
| pickup b6 from b3 (dropped on table)                      |
| pickup b10 from b12                                       |
| put b10 on table                                          |
| pickup b13 from b20 (dropped on table)                    |
| pickup b8 from b16                                        |
| put b8 on table                                           |
| pickup b2 from b19                                        |
| put b2 on table                                           |
| pickup b18 from b11 (dropped on table)                    |
| pickup b17 from b7 (dropped on table)                     |
| pickup b20 from b9 (dropped on table)                     |
| pickup b5 from b13 (dropped on table)                     |
| pickup b3 from b17 (dropped on table)                     |
| pickup b15 from b2                                        |
| put b15 on b2 (dropped on table)                          |
| pickup b12 from b8                                        |
| put b12 on b11 (dropped on table)                         |
| pickup b1 from b10 (dropped on table)                     |
| pickup b14 from b6                                        |
| put b14 on b13 (dropped on table)                         |
| pickup b15 from table                                     |
| put b15 on b10                                            |
| pickup b3 from table                                     |
| put b3 on b1                                             |
| pickup b13 from table                                    |
| put b13 on b12                                           |
| pickup b7 from table                                     |
| put b7 on b8                                             |
| pickup b5 from table                                     |
| put b5 on b2                                             |
| pickup b17 from table                                    |
| put b17 on b16                                           |
| pickup b4 from table                                     |
| put b4 on b3                                             |
| pickup b10 from table                                    |
| put b10 on b5                                            |
| pickup b8 from table                                     |
| put b8 on b9                                             |
| pickup b20 from table                                    |
| put b20 on b15                                           |
| pickup b6 from table                                     |
| put b6 on b7                                             |
| pickup b11 from table                                    |
| put b11 on b20                                           |
| pickup b12 from table                                    |
| put b12 on b11                                           |
| pickup b14 from table                                    |
| put b14 on b13                                           |
| pickup b16 from table                                    |
| put b16 on b14                                           |
| pickup b18 from table                                    |
| put b18 on b17                                           |
------------------------------------------------------------
```

Figure 34b. Result of 20-block Problem 3

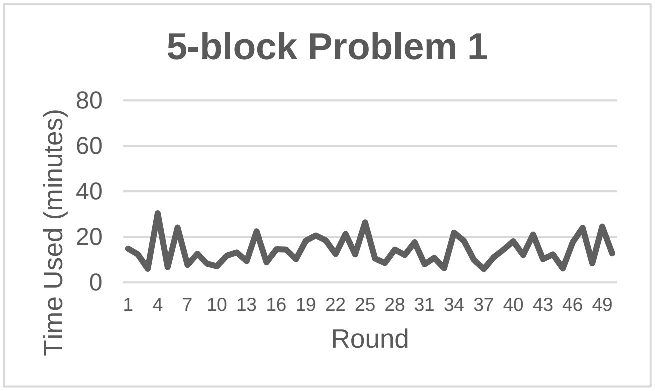
5.2 Train for general problems (ten 5-block problems)

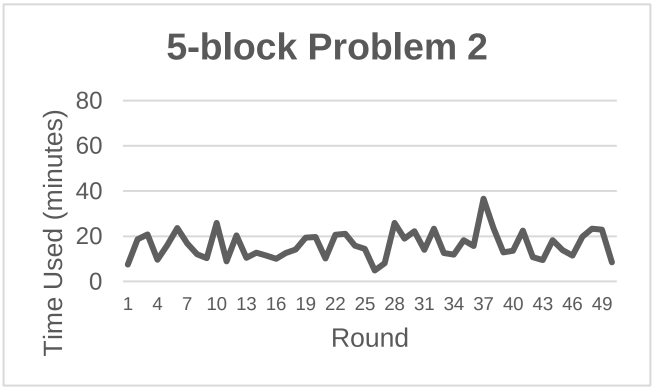
After testing the problems above in Section 5.1, we know that the proposed approach can solve the specific problems. We want to further investigate whether our current method can solve general problems, i.e. train the model with multiple problems to see whether it can understand the pattern well and then solve each individual problem or even unseen problems. We want to see what the result is and whether there is anything we can improve from the current method for preparing to solve general problems later.

In this experiment, we used ten 5-block problems to train. For each problem, we trained for 50 episodes, then go to the next problem. We trained the ten problems one by one in each round (so total of 500 episodes in one round). After finishing one round, then start the next round from problem 1 to problem 10 again. We totally trained for 50 rounds. Therefore, for each problem we actually trained for 2500 episodes, but mixing with other problems, so there was an interval between the next round training, i.e. after training 50 episodes the problem, and then 450 episodes for other 9 problems, then again train 50 episodes for the problem.

The result showed that reinforcement learning could not learn to solve each of the problems by following this approach. In addition, the training of other problems confused the model for solving the one it could originally solve.

Although the result is not very successful at this point, we can analyze the experiment so that we can improve the method in the future. First, we look at the time used in a problem-wise view (See Figure 35).





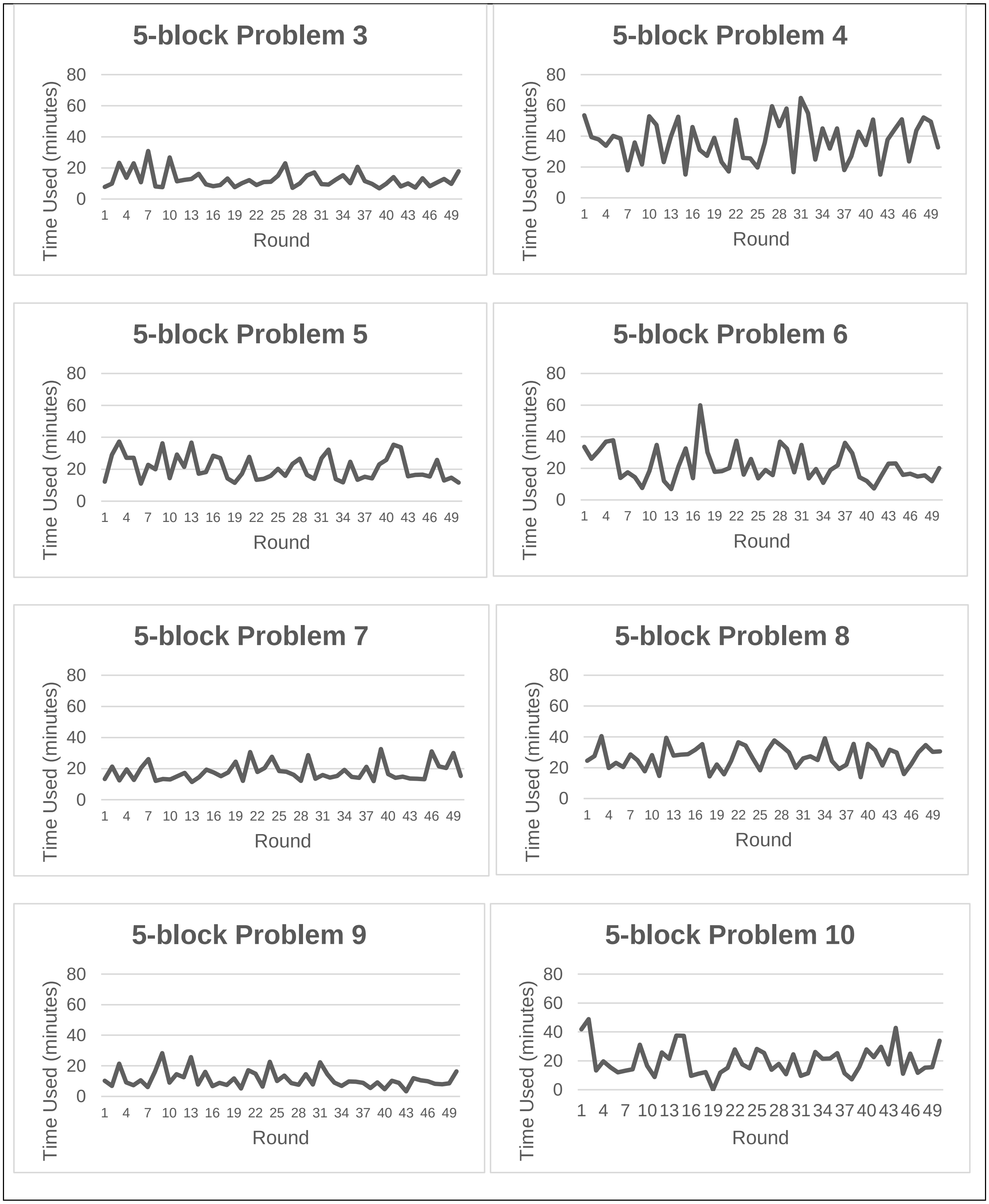


Figure 35. Experiment training time used in problem-wise view

Theoretically, if a problem is trained for more episodes, the weights should keep improving more and more, thus the time used to solve the problem would become shorter and shorter. Nevertheless, the result we see shows that the time used for the problems do not follow a decreasing trend, but sometimes going upward and sometimes going downward for every 1~5 rounds. This means that when the weights were learned to solve that problem better, later it performed worse again, then better again, and worse again, so on and so forth.

We can see that when the weights learned to solve some problems better, it performed worse on the others. Reversely, when it performed better on those that it did badly before, it performed worse on those it did well initially.

We can also find that in some particular problems, it needs to take a longer time to train. It is not due to the problem of the reinforcement learning model, but due to the complexity of the problem(s) is higher than the others. In the ten 5-block problem samples we tested, Problem 4 is the most difficult one, it needs to take more steps to get to the goal state. Hence, the average time used on Problem 4 is the longest (See Figure 36).

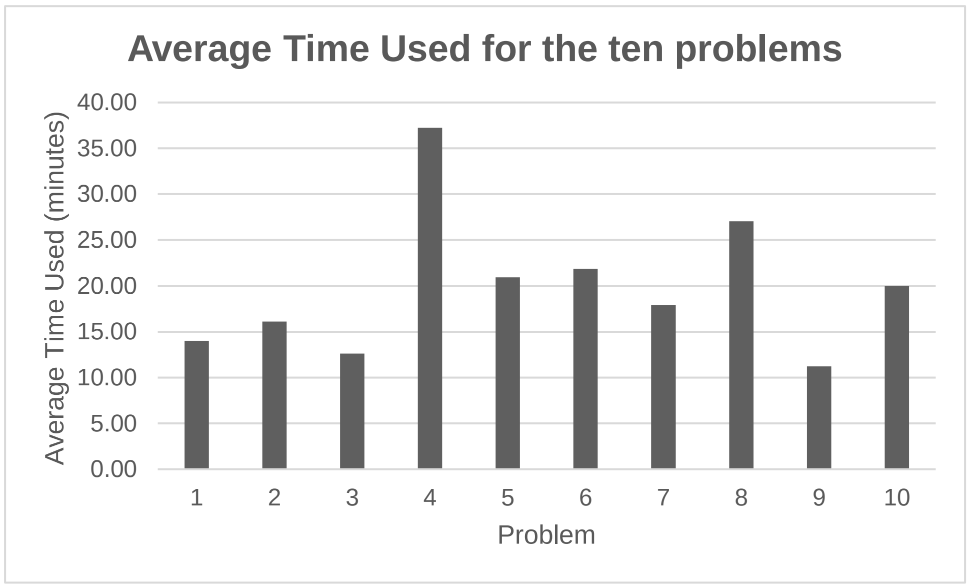


Figure 36. Average time used for each problem running 50 rounds

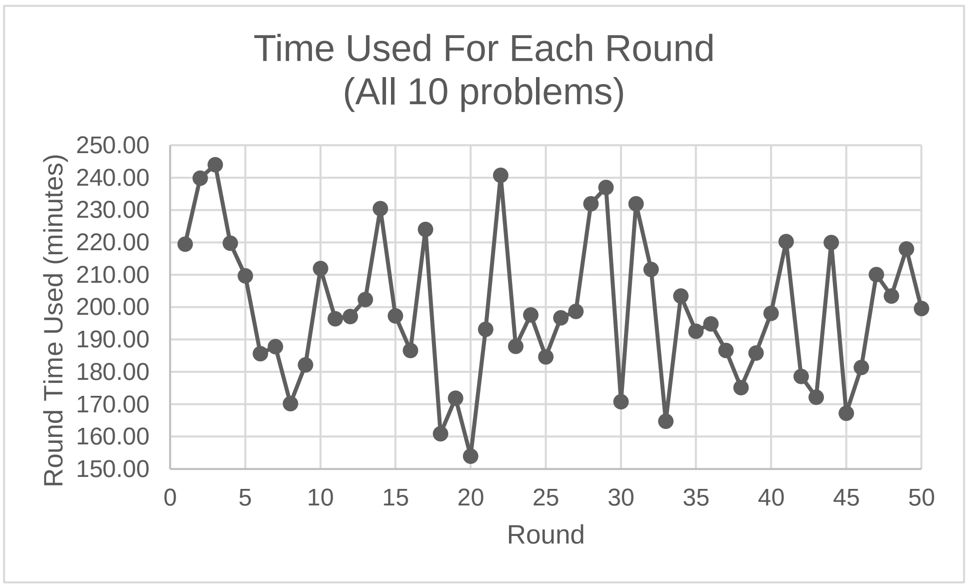


Figure 37. Average time used for each problem running 50 rounds

If we compare Figure 35, 36 and 37, we will find that at each round, the weights are favor on some particular problems, but it keeps switching to lean on different problems. The reason for this is, in the reward method, we set that the agent would receive a penalty from the environment if it chooses a move that cannot shorten the distance to the goal state. Therefore, the situation is, when the weights learned to solve Problem 1 very well, it received a large number of penalties when it trained other problems. Hence, the time used for other questions became longer. After it received a lot of penalties, the weights kept updating to new values, this time perhaps it worked well on Problem 8. However, when the next time the model tried to use the weights which favor Problem 8 to solve Problem 1, the time used for Problem 1 became longer again, and also it received a great number of penalties. This is the reason why the weights could not be tuned across the problems.

PART6: RESULTS AND DISCUSSION

We have conducted extensive experiments to test our approach. The planning problems were the benchmark problems obtained from an international planning competition (Bryce and Buffet 2008). Experimental results showed that our approach was able to find optimal solutions for specific planning problems. When guided by using the AI search, reinforcement learning was indeed more effective in learning.

For 3-block problems, 5-block problems, 10-block problems, 15-block problems, 20-block problems, we created 3 problems in each group to test. This experiment showed clearly and thoroughly that our approach could solve all the specific N-block problems up to 20 blocks, not by chance. We believe that our approach should be able to solve specific problems of more blocks if we adjust the 2D matrix to a much larger one. The reason is that we are using an Artificial Neural Network to predict the Q-values without exploring the huge search space.

Although our attempt to use the current method to solve general 5-block problems was not successful, the experiment is still meaningful. We found that when we let the model train on different problems one after another, the model could not understand the relationship and the difference between different 5-block problem tasks. It ended up thinking of the ten 5-block problems as one task, mixing up all the weights, and failing to solve each individual problem. We reviewed our experiment and we figured that our current way of learning multiple problems might not be a suitable way, we would try to improve the training approach later. Another important finding which we observed from the experiment was that, even though we bound the current state and the goal state together as the input, owing to the limitation of Artificial Neural Network, we had to flatten the two 20×22 matrices into an 880 array. Hence, the artificial neural network may not understand the first half (0 ~ 439) and the second half (440 ~ 879) have a special relationship that they needed to be compared row by row. The model might assume all the values in the input are all just connected to all other values. Therefore, we plan to adopt Convolutional Neural Network (CNN) in the future to replace the current Artificial Neural Network. For example, Convolutional Neural Network is widely used in the image processing domain because it can separate an image into 3 layers of input – Red (R), Green (G), Blue (B). CNN can detect the relationship between different layers so that it should be able to understand the relationship between the two states in the reinforcement learning model. Additionally, CNN can capture the spatial feature from the input, which ANN cannot do. It should be able to learn better not only between the current state and the goal state, but also inside a state matrix, perhaps it can recognize the relationship among different rows. Last but not least, we consider adding more hidden layers in the network, to make the network deeper, this should be also very helpful to improve the learning quality.

PART7: CONCLUSION

In this study, we have explored a new direction to tackle AI planning problems by using reinforcement learning. Our research focused on a well-known benchmark planning problem, blocks-world. In order to make the planning problems fit in a machine learning model, we proposed a novel approach to convert a state (i.e., the configuration of blocks and the status of a robotic arm) into a two-dimensional matrix. After transforming the blocks-world problems into tabular data, a neural network can be used to process the data.

Inspired by the way of how a planning problem is modeled, we also modeled the output from the neural network as a 2D matrix of the same shape as a state. The output represents the expected future rewards (i.e., Q-values) for each of the possible actions if they apply to the current state (i.e., the input to the neural network). As not every action is applicable to the current state, we designed an approach to label the valid and invalid actions using a mask. As a result, the agent can only choose a valid action in decision-making, thus significantly improving the learning efficiency.

By using the semi-gradient Temporal Difference learning technique, our approach can update the weights of the neural network in every time step once it observes a reward and the state transition caused by the action it chooses. To help our approach make the correct decision when choosing actions, we also designed some strategies to make the learning process guided (e.g., by heuristic search or supervised learning). Hence, the time-consuming learning process can be greatly shortened compared to the brutal force that lets the model learn from scratch without any guidance. In fact, our experimental study showed that a model might not be able to learn any meaningful results without necessary guidance.

By tuning the weights of the neural network, the neural network can predict accurate Q-values (i.e., expected future rewards), based on which our approach can choose the correct action that can maximize the Q-values, thus bringing the system closer to the goal state. As a result, once a neural network is trained, the correct action can be predicted immediately without the need for an enormous table to store all the states and the huge efforts to search for an appropriate action. Therefore, with our approach, the unmanageable exponential-grown state space can be handled by using the neural network.

In general, our approach offers an alternative way to solve the Fully-Observable Non-Deterministic (FOND) planning problem. Instead of searching for a strong cyclic plan for each of the possible non-deterministic outcomes, our neural network can instantly make a decision based on the current state to choose the right action that will lead to a new state closer to the goal. By recursively applying our approach to each unhandled outcome (i.e., states), a plan will be effectively found. Hence, the optimal solution we find for a specific problem is indeed the neural network itself, which can handle any state in the problem and make the corresponding decision. (The results shown in Section 5.1 are examples of how the shortest plans look like, they are not necessarily the optimal plans because of the non-deterministic effect.)

The experimental results of this study show that reinforcement learning has the capability to solve specific AI Planning problems without human intervention. This experience learned from this study will be valuable to the research fields of AI planning and reinforcement learning. Furthermore, the knowledge and experience will pave the way to a more advanced reinforcement learning approach that can generalize to a class of planning problems, i.e. solve new planning problems without additional training.

PART8: FUTURE RESEARCH

This research is a preliminary study to find out whether machine learning can be used in AI planning problems. The results we obtained have demonstrated that for a specific task, reinforcement learning can learn without prior knowledge and is capable of finding optimal solutions.

In the future, we plan to develop a more advanced approach that can solve the blocks-world problems in general, not case by case. As we mentioned in Section 6, we will try to use a Convolutional Neural Network (CNN) to investigate whether it can understand the relationships between the current state and the goal state and among different rows in a state better than the traditional Neural Network. We will employ cutting-edge deep learning techniques to make the CNN go deeper and more powerful. We will also re-design the reward system settings to explore more efficient ways to guide the model to achieve more generalized capability.

We can foresee that it will be a great breakthrough if a reinforcement learning approach is trained once and then is able to solve a class of planning problems, thus making it possible to use limited resources and computing power to deal with the explosively large state space, state transitions, and the action combinations.

REFERENCES

Bădică, A., et al. (2020). Quantifying Blocks World State Space. 2020 International Conference on INnovations in Intelligent SysTems and Applications (INISTA), IEEE.

Baral, C., et al. (2000). "Computational complexity of planning and approximate planning in the presence of incompleteness." **122**(1-2): 241-267.

Barto, A. G., et al. (1995). "Learning to act using real-time dynamic programming." **72**(1-2): 81-138.

Bertoli, P., et al. (2001). MBP: a model based planner. Proc. of the IJCAI’01 Workshop on Planning under Uncertainty and Incomplete Information.

Bonet, B. and H. Geffner (2003). Faster heuristic search algorithms for planning with uncertainty and full feedback. IJCAI, Citeseer.

Bresina, J., et al. (2012). "Planning under continuous time and resource uncertainty: A challenge for AI."

Bryce, D. and O. Buffet (2008). International planning competition uncertainty part: Benchmarks and results. In Proceedings of IPC, Citeseer.

Chen, Y., et al. (2004). "SGPlan: Subgoal partitioning and resolution in planning."

Cimatti, A., et al. (2003). "Weak, strong, and strong cyclic planning via symbolic model checking." **147**(1-2): 35-84.

Dey, A. J. I. J. o. C. S. and I. Technologies (2016). "Machine learning algorithms: a review." **7**(3): 1174-1179.

Fikes, R. E. and N. J. J. A. i. Nilsson (1971). "STRIPS: A new approach to the application of theorem proving to problem solving." **2**(3-4): 189-208.

Fu, J., et al. (2016). "Fast strong planning for fully observable nondeterministic planning problems." **78**(2): 131-155.

Fu, J., et al. (2011). Simple and fast strong cyclic planning for fully-observable nondeterministic planning problems. IJCAI Proceedings-International Joint Conference on Artificial Intelligence.

Gupta, N. and D. S. J. A. I. Nau (1992). "On the complexity of blocks-world planning." **56**(2-3): 223-254.

Hansen, E. A. and S. J. A. I. Zilberstein (2001). "LAO∗: A heuristic search algorithm that finds solutions with loops." **129**(1-2): 35-62.

Hoffmann, J. and B. J. J. o. A. I. R. Nebel (2001). "The FF planning system: Fast plan generation through heuristic search." **14**: 253-302.

Hogg, C., et al. (2009). Learning hierarchical task networks for nondeterministic planning domains. Proceedings of the 21st international jont conference on Artifical intelligence.

Homer, S. and A. L. Selman (2011). Computability and complexity theory, Springer Science & Business Media.

Hopfield, J. J. J. P. o. t. n. a. o. s. (1982). "Neural networks and physical systems with emergent collective computational abilities." **79**(8): 2554-2558.

Jain, A. K., et al. (1996). "Artificial neural networks: A tutorial." **29**(3): 31-44.

Jaramillo, A. C., et al. (2014). "Fast strong planning for FOND problems with multi-root directed acyclic graphs." **23**(06): 1460028.

Kingma, D. P. and J. J. a. p. a. Ba (2014). "Adam: A method for stochastic optimization."

Koehler, J. and J. J. J. o. A. I. R. Hoffmann (2000). "On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm." **12**: 338-386.

Kyan, M., et al. (2014). Unsupervised Learning: A Dynamic Approach, John Wiley & Sons.

Long, D. and M. J. U. T. E. J. f. t. I. P. Fox (2002). "Progress in AI planning research and applications." **3**(5): 10-25.

McCulloch, W. S. and W. J. B. o. m. b. Pitts (1943). "A logical calculus of the ideas immanent in nervous activity." **5(4)**: 115-133.

McDermott, D., et al. (1998). PDDL-the planning domain definition language, Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational ….

Muise, C. J., et al. (2012). Improved non-deterministic planning by exploiting state relevance. Twenty-Second International Conference on Automated Planning and Scheduling.

Pistore, M. and P. Traverso (2001). Planning as model checking for extended goals in non-deterministic domains. IJCAI.

Priddy, K. L. and P. E. Keller (2005). Artificial neural networks: an introduction, SPIE press.

Rosenblatt, F. (1962). "Principles of neurodynamics: Perceptions and the theory of brain mechanisms."

Sutton, R. S. and A. G. Barto (2018). Reinforcement learning: An introduction, MIT press.

Werbos, P. J. P. D. d., Harvard University (1974). "Beyond regression:" new tools for prediction and analysis in the behavioral sciences."

Wiering, M. and M. Van Otterlo (2012). Reinforcement learning, Springer.

Winterer, D., et al. (2016). "Structural symmetries for fully observable nondeterministic planning."

Yegnanarayana, B. (2009). Artificial neural networks, PHI Learning Pvt. Ltd.