A Self-Organizing Approach to Problem Solver for 15-Puzzle

Kyuseok Jung

The Graduate School

School of Electrical and Electronic Engineering

Yonsei University

A Self-Organizing Approach to Problem Solver for 15-Puzzle

A Master Thesis
Submitted to the Department of
Electrical and Electronic Engineering
and the Graduate School of Yonsei University
in partial fulfillment of the
requirements for the degree of
Master of engineering

Kyuseok Jung

February 2023

This certifies that the master's thesis
of Kyuseok Jung is approved.

Thesis Supervisor: DaeEun Kim

Euntai Kim

Kyeon Hur

The Graduate School

School of Electrical and Electronic Engineering

Yonsei University

February 2023

Acknowledgements

약 2년이 조금 넘는 시간동안 많은 도움을 받고, 많은 배움을 얻었습니다. 졸업을 앞둔 지금 돌이켜보면, 지금도 별반 달라지지 않았을지도 모르겠지만 연구실에 발을 들이던 때는 참으로 하고싶은 것이 많던 시기였습니다. 이상한 의욕에 차, 어떤 어떤 연구에 관심이 있다며 교수님과 처음 면담을 하였을때 그런 저를 보며 교수님이 어떤 생각을 하셨을지 어떤 생각을 하셨을지 궁금해집니다. 하고싶은 것은 많지만 경험도 부족했고, 실제적으로 깊은 연구가 어떤것인지 잘 알지 못하던 때였습니다.

하지만 아직 그런 티를 완전히 벗지는 못하였지만, 지금에 이르러 이렇게 졸업을 앞두고 이렇게 졸업 논문의 감사 인사를 적고 있는 저를 보면 감회가 새롭습니다. 이런 글을 빌어, 이렇게 어떻게든 졸업을 할수 있도록 도움을 주신 분들에게 감사의 말씀을 전하고 싶습니다. 먼저 무턱대고 찾아주신 저를 받아주셨고, 지금까지 이렇게 졸업 논문을 완성할수 있도록 포기하시지 않고 2년이 넘는 지도를 해 주신 김대은 교수님에게 가장 큰 감사를 드리고 싶습니다.

그리고 이어서 지금 졸업하는 저희 세 학생과 생활하며 언제나 큰 도움을 주신 강병문 선배에게도 큰 감사를 드립니다. 저희 셋이 강병문 선배와 같이 연구실 생활을 진행하면서 드린 도움보다, 동시에 졸업하는 학생 세명의 동시에 관리하시는데 신경을 쓰신것이 훨신 많았을 것이라고 생각합니다. 또한 이렇게 함께 졸업 준비를 하면서 서로 도움이 되는 자료들을 공유 주신 이해리, 최호식 동기분들께도 감사를 드리고 싶습니다.

그리고 이런 학교 내의 도움 이외에도 졸업을 진행하면서, 대학원 생활을 언제나 응원해 주시던 제 어머니, 박현숙 여사님에게 사랑하며 언제나 감사드린다고 말씀드리고 싶고. 저의 처참한 영어를 어떻게든 참으면서 거의 두달 밤동안 매일 화상통화를 하며 실시간으로 영어 번역을 해주던, 제 친구 ”프리랜서 번역가 김윤수”씨 에게 감사를 하고 싶습니다.

그리고 이 논문을 읽으시는분에게도 감사를 드립니다. 아직 제 성에 차지 않는 부분이 있지만, 제가 2년동안의 연구가 부끄럽지는 않은 결과를 냈다고 생각합니다. 만약 이 논문을 읽게 되신다면, 많은것을 얻어가셨으면 합니다.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Kyuseok Jung*)

Table of Contents

List of Figures

List of Tables

Abstract

Optimal path-finding is to find efficient paths of a specified starting point and target point. These starting points and target points can be positioned in a two-dimensional space such as on a map, the joint angle of the robot arm, and the state of the game or puzzle. The optimal path-finding problem used in so many cases is one of the important problems in the field of artificial intelligence. The 15-puzzle is a well-known classical problem in the field of optimal path-finding. It is a sliding puzzle that moves tiles in the grid to be positioned in order, consisting of fifteen square tiles that are numbered 1-15 and one blank space. The traditional approach of optimal path solver uses a graph search algorithm A* and it often utilizes a heuristic of calculating the number of misplaced tiles or the sum of distances between the position of each tile in a given board and the position in the goal frame. To overcome this attempts to improve these heuristics with databases or reinforcement learning approaches have been made. However, these improvements are presented with heuristics for fixed target points, not universal heuristics that operate targeting all nodes.

In this thesis, we present neural heuristics, which replace these heuristics to increase search efficiency, and can operate universally on all nodes, by learning to map the board's configuration to a distance to reach the target state via a neural network, and a network structure for them. The presented data collection method for heuristic training, the state analysis method, and the structure of the network showed sufficient performance improvement compared to the conventional heuristic. And we present Deep A* Iteration, a method that combines the heuristic training method and a reinforcement learning approach, which trains heuristics in a 15-puzzle environment without human knowledge and any training bases in data collection. Using a curriculum approach, Deep A* Iteration was able to collect and apply more difficult samples than sampling data using conventional heuristics. Furthermore, it is shown by experiments that neural networks trained with Deep A* Iteration have the property to converge on the Ideal heuristic. And finally, we present Branch sampling, which automatically constructs a training range while following the fundamental principles of the first presented data collection method to maximize sampling efficiency. Branch sampling is a collection of branches generated by A* searches, which can generate tens of thousands of times more data in a single search than traditional data collection methods. As opposed to pure Deep A* Iteration having to follow curriculum methods to increase its difficulty, applying Branch sampling automatically generates and trains more difficult samples of data.

In summary, this paper presents a new algorithm to train the heuristics of 15 puzzles, inspired by classical solutions to solve 15 puzzles and deep learning, and deep reinforcement learning.

Key words : A*, heuristic, 15-puzzle, N-puzzle, optimal path-finding, searching algorithm, deep learning, neural network, deep reinforcement learning, convergence, algorithm

Chapter 1

Introduction

Many attempts have been made to solve complex problems that humans have found difficult to interpret using machines. If such problems can be easily interpreted by understanding the relationship between the fundamental principles or states, researchers can compose logic and laws to solve them relatively easily. However, there are more instances where this cannot be achieved, and the fields of machine learning (hereafter referred to as ML) and artificial intelligence (hereafter referred to as AI) have been created to systematize attempts to design machines to solve such cases, with many studies still being conducted. These domains have attracted plenty of attention, whereby to solve such aforementioned problems with successive or multiple choices, AI or ML can arrive at which of the given options is the best one. The path-finding problem is one such problems, where the challenge is to determine the ideal direction to take (i.e., which direction is a best choice) to efficiently reach the desired goal. Humans have been solving path finding problems very routinely through tasks such as inferring which path to take from the current position to reach the desired position and estimating which path the arm must move to grab an object with the hand.

Just as humans routinely perform this task, path finding also finds applications in major engineering problems. For example, to discover a route for reaching the destination in autonomous driving, a navigation system is required. Similarly, to pick up an object at the desired position and angle with a robot arm and a manipulator, it is important to determine which angle and in what order each robot arm joint should be moved, Finally, path-finding can be used to solve problems such as what choices need to be made in what order to solve a puzzle with the fewest number of actions. Studies are also being

conducted to make this path-finding more efficient and faster with fewer number of actions. In this study, we have focused on solving the type of puzzle problems where path-finding can be used.

1.1 N-Puzzle

N-puzzle is a difficult puzzle problems that require path finding to solve it. Finding the shortest path from the initial position to the goal state has been proven to be an NP-hard or NP-complete problem (Ratner and Warmuth, 1990; Goldreich, 2011). Therefore, N-puzzle can be considered as one of the classical problems of artificial intelligence.

The N-puzzle, (8-puzzle on 3x3 configuration, 15-puzzle on 4x4 configuration, and 24-puzzle on 5x5 configuration) is a well known representative example of problem solving, which aims to find the path from the given initialized state to the goal state. The 15-puzzle is a sliding puzzle comprising 15 square tiles and a blank one, each with its own contents from 1 to 15. Generally, the 15-puzzle goal board has a board frame on which a set of numbers are arranged. In this puzzle, the order of 15 numbers at the initial configuration should be rearranged to ultimately reach the goal configuration. This problem can be solved by exploring all the states and paths that a puzzle can produce, however, as the size of the puzzle increases, the number of states also increases. For example, when the number N is 15, it almost impossible to search for all state spaces. There have also been many attempts to reduce computing time and memory resources to solve this N-puzzle problem.

The A*(Hart et al., 1968; Pohl, 1970) and IDA*(Korf, 1985) algorithms are path searching algorithms that find an efficient or optimal path to the goal with the graph structure approach, which can be applied to solve difficult problem that can not be easily attempted by applying a simple rule-based solution. This approach allows us to navigate a graph to ensure that the path to the goal is found while simultaneously narrowing the scope to be explored through indicators built by human knowledge. Most studies that have explored these algorithms have either designed a problem-based mathematical indicator or built a database through which the collected data enables more efficient navigation.

1.2 Deep Reinforcement Learning

Heuristic approaches designed with mathematics or rule bases described above (through databases), including other simple best first search and special rules, depend on which problems to solve. Further, there are limitations stemming from in-depth research requirements restrictions of using only local approaches. On the other hand, techniques combining neural networks and reinforcement learning are being used as problem-solving techniques, starting with the Deep Q-Network(DQN), which was first announced in 2013 and surprised people by solving Atari video games (Mnih et al., 2013).

Well-learned neural networks, from simple video games to robotics, show plenty of promise as universal solutions. Among the, Alpha Go(Silver et al., 2016) is at the forefront by having taken the lead solving Go using the Monte Calro Treee Search(MCTS), neural network techniques, and people's knowledge of Go. Attempts have also been made to solve the problem by combining MCTS and neural networks such as Alpha Zero that develops and learns without requiring human knowledge and Mu Zero that does not need any simulation environment for tree search (Silver et al., 2017; Schrittwieser et al., 2020). These MCTS-based papers, such as AlphaGo, have solved problems, which were once considered impossible to solve, by relying on to sufficient learning time, information, and computational resources.

However, the N-puzzle has dimensions that make it difficult to apply the general deep reinforcement learning algorithm. First, it has an extremely large state space ($N!$ states), and one would need to reach only one desired target state. Next, regardless of how many choices are made to solve the problem, the next step does not get completed until the desired target state is reached. This property is exhibited by several puzzles such as Rubik's cube and Sokoban. Thus, these problems cannot be easily applied to the existing deep reinforcement learning algorithm, instead, deep reinforcement learning algorithms, especially incorporated with properties to solve puzzle problems, are being newly proposed (Agostinelli et al., 2019; Agostinelli et al., 2021).

1.3 Motivation and Objectives

Motivated by deep reinforcement learning algorithms and conventional N-puzzle solvers, we propose a new algorithm for N-puzzle and other environments such as Rubik's

cube, that possesses the aforementioned properties. The main purpose of this thesis was to develop an efficient solver with deep learning and reinforcement learning to, solve special puzzles.

The goals of this study were as follows.

Optimized network design using the characteristics of the environment. The deep reinforcement learning algorithm for puzzles presented in previous studies uses a very large-scale deep learning network compared to other reinforcement learning methods to solve puzzles, thus slowing down the search process to solve the environment. This can be attributed to a large amount of computation is required to evaluate each state during the search process. Therefore, to solve the puzzle faster, a more optimized and suitable data processing method or network structure should be proposed. Our objective was to present a data processing technique to effectively solve the N-puzzle described in this study through an optimized network.

A new algorithm design with a different approach from existing studies. previous studies have explored the DeepCubeA algorithm to learn how close each state is to the target state through value iteration. However, overestimation has been observed in many research results for value iteration, wherein learning was performed by referring to the operating results of the network itself. In DeepCubeA, to suppress this overestimation as much as possible, the value should not be updated until the loss of the network is reduced to a certain extent. This is a limitation of the pure value iteration approach. It is essential that the network that evaluates the state needs to be more precise than a certain level. Therefore, it is necessary to propose an algorithm to evaluate how close the network is to the target state through an approach other than value iteration. Our objective was to present an algorithm to perform faster learning while collecting and learning data close to the actual optimal path, not value iteration.

Proof of convergence of the newly proposed algorithm. Algorithms that use neural networks cannot guarantee a perfectly optimal choice for any problem. This is because the operation of neural networks is basically all approximation functions, which is the same for the algorithm we insted to propose, however, there is no guarantee that the algorithm we present will send an optimal path to the goal. Therefore, we prove that our algorithm converges to find the optimal path even if it does not guarantee the optimal path this way if there is a network of sufficient size and training iterations, it will work as a solver to find the optimal path.

1.4 Organization of Dissertation

This thesis comprises six chapters. In Chapter 1, various problems and application cases of reinforcement learning algorithms using deep neural networks to solve them have been introduced. In the end, the motivation and objectives of this thesis have been highlighted.

In Chapter 2, we will be introducing the background of the algorithm to solve the puzzle problem that we have discussed above. Among these, we will be focusing on the graph-based search algorithms for solving environments with reversible characteristics and application cases of deep neural networks. This chapter emphasizes the goals of this paper by introducing concepts that act as important factors in existing cases.

In Chapter 3, to the solve 15-Puzzle, a representative puzzle problem with reversible properties we have, we presented a data structural trick to reduces the time complexity of the priority queue of A* algorithms that can all be applied in a cost-one environment with the same graph edge. We also designed a neural network structure to effectively solve the 15-puzzle. The usage trick of the priority queue proposed was to reduce the input/output time complexity by a very small number of c on average by using the bucket priority queue, Meanwhile, the input/output time complexity of the priority queue whould use the existing heap structure takes $log(N)$. Furthermore, the neural network we designed learned heuristics (hereafter referred to as neural heuristics) by proposing a state parsing structure allowing the state space of the N-puzzle problem to be compressed while effectively helping interpret the state space. Compared to the network size used in previous studies, we were able to learn to predict heuristic by analyzing the state space while using a very small neural network. We also compared these results with existing applications to confirm significant performance improvements through our proposed method.

In Chapter 4, we have presented a method for learning heuristics such that the 15-puzzle can approximate the overall state space of the puzzle with reinforcement-learning curriculum methods without using human-made knowledge, i.e. conventional heuristics. We have also presented, theoretically, how the possible divergence of heuristics from the proposed method can be suppressed, thus converging to heuristics and predicting the intended optimal path we want. We also demonstrated, experimentally that a significant reduction in learning time occurs when zero-based learning, i.e., networks where no learning has progressed, was applied to the proposed reinforcement learning

curriculum method.

In Chapter 5, based on the method validated in Chapter 4, we collect the remaining branches derived from the A* search to improve the sampling efficiency required for learning, and present a way to self-organize and expand the learning range that was previously expanding manually. In addition, ‘the effectiveness of the proposed method to solve the data distribution and policy update problems was verified. we also compared the performance of our proposed final method with previously studied existing research cases.

Finally, the conclusions of the various methods presented, including potential future studies, have been drawn in Chapter 6.

Chapter 2

Background

There are many kinds of problems that machines find difficult to solve. For example, N-puzzles and other similar puzzles or games that can be repeated indefinitely until a single solution is found have enormous state space and reversible properties. In these problems, unplanned choices cannot have meaningful actions, which is why they are difficult to solve using AI. The classic approach to overcoming these challenges is to express the state of the environment through a graphical or tree representation and transform the problem to fit this form. We aim to apply a newly studied deep reinforcement learning technique to represent and solve the state space of the problem in graph form. However, before conducting such research, it is necessary to understand conventional research as well as the characteristics of the newly presented deep reinforcement learning research. Therefore, in this chapter, we investigate conventional N-puzzle solving methods as well as how newly emerging deep reinforcement learning algorithms work, what limitations exist, and finally, how the proposed deep reinforcement learning attempt works on the problem which we are trying to solve, as well as its limitations.

2.1 Conventional N-puzzle solving

A problem like N-puzzle and Rubik's cube can have its state space simplified by a reversible graph or a tree structure. When the problem is represented using a graph structure, it is possible to show the information of each state as nodes, and choices to solve a problem as the edge of a node. Puzzle problems like this can be simplified as a problem in which we find path between the selected node and a random one. This

suggests that a problem simplified in such a manner can be solved using a method that finds the shortest path on a graph like the Dijkstra algorithm (Dijkstra, 1959) or A* algorithm (Pearl, 1984), or a tree search method like Monte Carlo Tree Search (MCTS) (Chaslot et al., 2008).

In this thesis, the methods we propose to solve the puzzle problem are based on the graph search algorithm, because the graph-based search algorithm shows considerable strength in the state space with reversible characteristics. A tree-based search algorithm creates a new branch for every state that can be selected from a branch of the tree when creating a tree. In an environment with a reversible characteristic that can completely return to the previous state depending on the selection, a considerable amount of computation is wasted owing to the creation of meaningless branches that fall into infinite loops. However, in a graph-based search, the most optimised path is selected among numerous paths showing the same milestone state, and the loop due to the reversible characteristic is discarded as a non-optimised path. Therefore, for finding a path in an environment with reversible characteristics, this approach is generally used because it is possible to increase the computational efficiency by using a graph-based search method.

2.1.1 A* and heuristics

A* is algorithm that finds the shortest path from random starting point to the goal point in a graph structure. This algorithm is a modified version of the Dijkstra algorithm, which aims for the shortest route to a certain goal point. How fast A* algorithm proceeds with the search is proportionate to how much the search inclines towards target. This, in turn, is proportionate to the accuracy of the heuristic function that predicts the distance to a goal point from the current point.

Thus, many studies have attempted to develop a method for increasing the efficiency of the A* search by designing heuristic functions that fit the environment, and the following heuristic functions in N-Puzzle have been suggested: Hamming distance (Hamming, 1950), which counts unfit tiles between the current state to evaluate and the target state; Manhattan distance (Doran and Michie, 1966), which combines the distance the tile moved on the axis of row and column between the current state and target state; and Linear conflict (Hansson et al., 1992), which counts the conflict of tiles on the axis of row and column adding to the Manhattan distance. These heuristic functions have been designed to consider environmental rules and property. However,

it is impossible to design heuristics with higher accuracy because of the properties of complex non-linear environments.

Therefore, to overcome these challenges and predict heuristics with higher accuracy than described above, many studies were conducted to calculate heuristic values by constructing databases and processing the length of a pre-calculated optimal path. Culberson and Schaeffer's studies in Manhattan Distance reduce the number of search nodes by 0.13% by adding a human-designed 2^{18} transposition table, Endgame databases with all solutions below 25 movements, fringe reduction databases, and corner reduction databases (Culberson and Schaeffer, 1994). Subsequent studies applying half of the corner and fringe databases to Manhattan Distance reduce the number of search nodes by 0.10% (Culberson and Schaeffer, 1998). There is also the study by Korf and Felner, which reduces the number of search nodes to around 0.01% by applying the Disjoint database and reflected database (Korf and Felner, 2002). Cases have been studied in which human knowledge is used to build a database of sub-problems necessary to reach a fixed target point, and through this, solutions to problems have improved dramatically (Felner et al., 2004).

However, there is a clear limit in approaches that generates heuristic value by building databases of already calculated distances, or heuristic functions designed by analysing patterns and rules of environment. First, a heuristic function designed by analysing environmental rules or patterns need deep analysis and research on the environment itself. In light of environmental complexity and non-linearity, enhancing the performance beyond a certain point is exponentially harder, making it impossible to achieve higher search efficiency. Furthermore, methods that increase efficiency by constructing a database as described above need a deep understanding of the environment and enormous storage capacity to store some of the large state space.

2.1.2 Variation approach

Certain studies aimed to arrive at efficient (but not optimal) solutions or aimed to suggest completely new possibilities, in the case that heuristic was not designed using pure A* or through databases. The research by Parberry presents a greedy breadth-first search algorithm that finds non-optimal solutions, but can resolve the problem in 0.07 ms in a typical desktop environment (Parberry, 2014). In the study by Bhasin and Singla, genetic algorithms may solve NP-hard problems, so based on this, an algorithm was presented to solve 15 puzzles basing on Genetic Algorithms (Bhasin and Singla,

2012). In addition, Bischoff's study suggests that Local Value Iteration can be treated using an approach from Reinforcement Learning, if non-optimal solutions are acceptable (Bischoff et al., 2013).

2.2 Deep Reinforcement Learning Methods

Deep Reinforcement Learning on approximation functions using neural networks can be seen as a replacement methodology for recording and importation on assessment of a certain state by agents in traditional reinforcement leaning techniques. Neural networks have the capability to approximate a relatively bigger state space then other machine leaning techniques or memory storage forms depending on the original state (Schmidhuber, 2015; Van Hasselt et al., 2016). This has attracted considerable attention since the announcement of DQN, which efficiently trains the environment by taking a screen image of an Atari video game as an input in 2013 (Mnih et al., 2013). Now thanks to the interpretation ability of neural networks on large state spaces, many environments have been navigated through Deep Reinforcement Learning. As the potential for real-world applications has increased, not only simulated, game-like environments but also real-world environments to control robots have been considered. Thus, neural network approximation abilities have been verified on many state spaces, and through this we can also witness possibilities that heuristics in complex environments can approximate in some degree.

2.2.1 Value iteration Methods

Value iteration methods generally train a random state or predict what reward a choice get, to solve the environment through dynamic programming by using predicted values. The most standard method is Q-learning (Watkins and Dayan, 1992), which is designed to predict the discounted sum of the highest reward that could be yielded in the future, by choosing such an action on a random state. The most standard method is Q-learning (Watkins and Dayan, 1992), which is designed to predict the discounted sum of the highest reward that could be yielded in the future, by choosing such an action on a random state. Through this, Q-learning predicts what actions could yield highest rewards in the future and chooses an action to do so. There was an attempt to replace Q-learning, which predicts a value of each possible action with status inputted, with a neural network (Lin, 1992). Subsequent studies on the deep-learning-based approach

have been actively conducted, expanding to not only neural networks but also to deep neural network. This is now called DQN (Mnih et al., 2013).

Many studies has been conducted to further improve DQN. First, Q-learning suffers from overestimation bias trait, which can impede training efficiency. Double Q-learning was suggested to suppress this as possible (Hasselt, 2010). However DQN also suffers from overestimation bias trait because it derived from Q-learning. To suppress this, Double DQN was proposed, which is a combination of DQN and Double Q-learning (Van Hasselt et al., 2016). DQN utilises Experience replay (Lin, 1992) for continuous utilisation of training data. Conventional Experience replay in networks samples data at a finer dispersion while training gathered data. However, there are cases where data samples are too few or fail at training. Thus, Prioritised experience replay was suggested, in which Q-learning samples far more training samples having higher TD-error to better train these cases (Schaul et al., 2015). There also are many improvement plans on Q-learning and DQN algorithms (Wang et al., 2016; Sutton, 1988; Sutton et al., 1998; Bellemare et al., 2017; Fortunato et al., 2017).

The reason that only two such improvements, namely, Double DQN and Priority experience replay, are mentioned here is because those plans improve two core problems of enhanced learning using deep neural networks. First, Double Q-learning technique mitigates the overestimation bias that inevitably occurs on Value methods.

Overestimation bias increase the tendency of Q-learning to make a wrong choice because it overestimates values of an action higher than realistically possible to acquire in a real environment. Such overestimation biases could occur in any algorithm that is based in Value iteration. As Value iteration utilises trained approximation function as training data, the smallest amount of bias occurring on such a function could accumulate continuously. This is a phenomenon that could occur in all algorithms utilising Value iteration.

Second, Priority experience replay tries to mitigate the problem that network training does not get evenly distributed in the entire state space. In the entire state space, network training must distribute evenly for a good training, but data distribution says otherwise. Consequently, Priority experience replay measures poorly trained parts and concentrates on training to improve. These problems are widespread in all methods utilising deep neural networks and can occur in the algorithm on puzzle problem solving we are trying to accomplish, so they need to be overcome.

Value iteration is suitable for environments that need long plans as it can calculate indexes of acceptable accuracy, like Bischoff's study (Bischoff et al., 2013). In another attempt to solve the Rubik's cube, Corli's work (Corli et al., 2021) presents techniques such as maintaining the symmetry of state through Hamiltonian operators to solve the Rubik's cube using DDQN (Van Hasselt et al., 2016). Such studies have had some success.

2.2.2 Policy Methods

Policy methods are a general term that generally work in random states to train and choose a good choice and then solve a problem by utilising those deemed to be a good one. In an environment where one choice must be made from all given ones, these policy methods generate probabilities on what shall be selected in situations that need discrete choices. These policy methods first emerged from REINFORCE algorithm (Williams, 1992; Willianms, 1988) and have now been combined with Value method, consisting of an Actor who chooses a target and a Critic who evaluates the target. In the regard, an algorithm based on Actor–Critic (Konda and Tsitsiklis, 1999) is generally studied.

Among Actor–Critic algorithms, like a case of Q-Learning and DQN, Advanced Actor–Critic method (A2C. A3C is an asynchronous version) (Mnih et al., 2016) in which the Actor and Critic were trained using deep neural network were first suggested in deep reinforcement learning. Some studies for Actor–Critic algorithms perform continuous control to be used for motor joint control of robots that need real-number values within a certain range (Lillicrap et al., 2015; Fujimoto et al., 2018; Haarnoja et al., 2018), not just chosen selection, while others limit the Actor update to be neither too big nor too small (Schulman et al., 2015; Schulman et al., 2017) so as to converge into best policy fast. In these Actor-Critic algorithm, like a case of Q-Learning and DQN, Advanced Actor-Critic method(A2C. A3C is a asynchronous version) (Mnih et al., 2016) which Actor and Critic were trained with deep neural network, were first suggested in deep reinforcement learning, leading to studies for Actor-Critic type algorithm which perform continuous control which can be used as motor joint control of robots that needs real number values inside certain range (Lillicrap et al., 2015; Fujimoto et al., 2018; Haarnoja et al., 2018), not just chosen selection, and studies that limit Actor update not to be too big nor small (Schulman et al., 2015; Schulman et al., 2017), to converge into best policy fast.

However, these pure policy methods are not suitable for environments that need a substantially long plan. As explained above, when policy methods choose discrete actions, the agent provides a probability on what to choose, so even if there is a very low chance that each action would lead to wrong choices, a wrong selection can be made in high probability when there is an environment that needs such a long plan. Given these traits, pure policy methods are clearly unfit for our puzzle problem.

2.2.3 MCTS Methods

If Monte-Carlo Tree Search (MCTS), Policy method, and Value method are combined, an environment that need seriously long plan could be solved efficiently. Enlarging the tree search efficiently through a policy method by combining evaluation values made from an expanded tree via value method can enable us to pick actions of highest value from long plans. For solving problems needing such a lengthy plan, a training was conducted to solve Go problems basing on human records, leading to the Alpha-Go program, released in 2016 (Silver et al., 2016).

Algorithm Alpha Zero, which utilises self-playing and therefore does not need human records, can also solve Go, chess, or Shogi (Silver et al., 2017). Finally, Algorithm MuZero is released, which can solve not only Go, Chess, and Shogi but also Atari games, using Reinforced Learning on a model base that trains the environment itself so an internal simulator was not needed for the Tree search (Schrittwieser et al., 2020).

Thus, attempts to solve problems by planning far ahead through MCTS proceed efficiently. However, the environment that have been tested to introduce this algorithm, although it needs a significantly longer plan, clearly has differentiated traits from the puzzle problem we aim to solve. The environments solved above need long plans but are usually irreversible, so there are many cases wherein multiple answers are available inside an enormous state space needed for solving problems or winning a game. Consequently, getting a reward on solving the problem is relatively easy, and the environment can be trained more easily than puzzle problems.

It is clear that such an MCTS-based approach is not completely incapable of solving a puzzle problem. Scheiermann's work (Scheiermann and Konen, 2022), an attempt to solve Rubik's cube through MCTS-based approach, and DeepCube algorithm citemcaleer2018solving make it easier to reach acceptable efficiency than former attempts. However, it is clear that such approaches have limits.

2.3 Deep Reinforcement Learning Methods for Puzzle Solving

As mentioned above, the puzzle problems we are considering are not easy to solve using conventional Deep reinforcement Learning, which aims to solve a more general environment. Thus, an algorithm is needed that considers a specific puzzle problem or environment.

2.3.1 DeepCube

DeepCube (McAleer et al., 2018) is an algorithm specifically aimed at solving Rubik's cube only, inspired by Alpha Zero algorithm and utilising a similar approach. However, given the specific traits of the puzzle environment described above, specifically, the low probability of reaching the target point, DeepCube algorithm can provide alternative training methods compared to the conventional method of solving the environment and training using the results.

Autodidactic Iteration is a method to train the value and policy network suggested in DeepCube. This method to train the network generates samples that have been shuffled a certain number of times from the target state of puzzle environment (Rubik's Cube) when sampling the training state. After calculating all adjacent states in the sampled state, it evaluate values of adjacent states and proceeds with value iteration through the highest combined value of values and rewards that were calculated. Furthermore, by calculating actions to set the state when combining value and rewards, the highest value will be calculated and chosen, and the Policy network will be trained to do so. In this moment, action reward of puzzle problem environment get 1 only when reaching the target space, all others get a value of 0. In this moment, the action reward of puzzle problem environment gets a 1 only when reaching the target space, while all others get a value of 0. By doing this, value networks get a higher value as it is closer to the target state, Policy networks calculate which choices have a better chance to get closer to it. Value and Policy networks trained this way work accurately on smaller plans, but cannot choose values and actions that consider a substantially larger plan.

Thus, MCTS solver uses Value and Police networks calculated as described above in DeepCube. Through the Policy network, the Tree is expanded with choices having a higher chance of being good choices. After combining all values calculated in tree, a

choice is made by which one choice would be the highest in current state. Through this, the algorithm can make a choice that has a statistically higher value considering a long plan. Thus, DeepCube can solve the Rubik's cube more often even in difficult cases. Although not all of the cases has been possible to solve, it is better at finding the optimal path as compared to the conventional Rubik's Cube solver, which did not use deep learning when encountering higher difficulty.

2.3.2 DeepCubeA

DeepCubeA Algorithm (Agostinelli et al., 2019) is inspired by the DeepCube algorithm and utilises the A* algorithm and heuristics for solving puzzles without using MCTS. The A* algorithm can calculate a route to a target point in any situation, because heuristic accuracy only dictates search efficiency and whether the route is optimal or not.

Deep Approximation Value Iteration (DAVI) only predicts values, unlike Autodidactic Iteration, so the value network is designed to train heuristics that show how many moves are really left, not predicting rewards. When DAVI trains, samples that had been shuffled certain times are generated from the target state. All states adjacent when sampled were calculated, and the time left till target point (the heuristic) is predicted, training a value that added +1 on the lowest heuristic values of adjacent state to be outputted through the Value network. Deep Approximation Value Iteration(DAVI) only predict values unlike Autodidactic Iteration, so the value network is designed to train heuristics that show how many movement is really left, not predicting rewards. By repeating DAVI, the network trains a length of shortest path to reach each state to the target in the state space in the puzzle environment.

Performing the A* algorithm by utilising heuristics calculated in this way, the shortest route from start state to the target can be calculated with high search efficiency. However, although it has a higher search efficiency, A* search needs to generate thousands or even millions of nodes to find the path and evaluate it through heuristics, and calculating each heuristic one by one through network leads to lower search speed. To overcome this, DeepCubeA researchers suggested the following method instead of conventional A* to effectively perform network heuristic evaluation on devices, and although rough, to enable faster search.

Batch Weighted A* Search (BWAS) has the following difference. First, conventional

A* performs heuristic evaluation by the nodes, one by one, while expanding the search. However, while processing massive calculation of neural networks in devices like GPUs, taking one sample at a time is inefficient because of a delay that occurs when transferring data from CPU memory to GPU memory. Therefore, when there is a need for calculating multiple samples on neural network, it is traditional to calculate multiple samples at once. Thus, in BWAS, as the search expands, multiple nodes expand simultaneously to evaluate as many samples as possible, optimising the time taken to evaluate heuristics through the neural network by evaluating multiple nodes at once. Second, in BAWS, a concept of weighted A* was applied, reducing the rate of optimal path following A*. However, a greedier search was implemented to design a route from the starting point to the target point with higher efficiency.

Through this, DeepCubeA could solve problems like Rubik's Cube, 15-, 24-, and 35-Puzzle, Lights out, and Sokoban with higher efficiency. However, DeepCubeA utilised an absurdly large neural network of 14 hidden layers for higher heuristic accuracy and a longer training time of two days. The reason such a large neural network is needed and a longer time taken is that when DeepCubeA trains heuristics through Value iteration, overestimation biases inevitably occur, so networks are only updated when those biases substantially disappear. To solve this, a preprocessing suitable for the environment, a design for the neural network structure, a more efficient training method, and also a design of a training algorithm based on real data in which overestimation bias does not occur are needed. Furthermore, DeepCubeA can only be a local solution of multiple cases, which is not a universal heuristic of environment calculated through DeepCubeA, because it only trains heuristics for a single fixed target.

2.4 Summary of Chapter2

In this chapter, we look into the process of conventional approaches to solve 15-Puzzle, why deep reinforcement learning algorithm has limitations on the puzzle problem we are trying to solve, and what kind of attempts have been made to overcome these limitations.

The conventional classic 15-puzzle solving attempt has been mainly studied to improve the heuristic based on A* algorithm, but the limit was reached on function-based heuristic design from the perspective of higher performance. Consequently, many studies were conducted by processing actual, readily available data to construct databases.

However, this method of designing heuristics to fit the problem can only be a local solver, so there is an inevitable need for very deep examination and research to improve performance continuously.

Conventional deep reinforcement learning researches show generally good performance in many highly complex problems because of the superior performance of the approximation function of deep neural networks, but shows a clear limit, and the puzzle problem we aim to solve is one of the typical problems that face this limit.

Thus, to solve these problems, a special enhanced learning algorithm is proposed, unlike the contemporary methods that need a substantially large network and long training period to proceed. This shows that a novel approach to solve these problem from another perspective is needed.

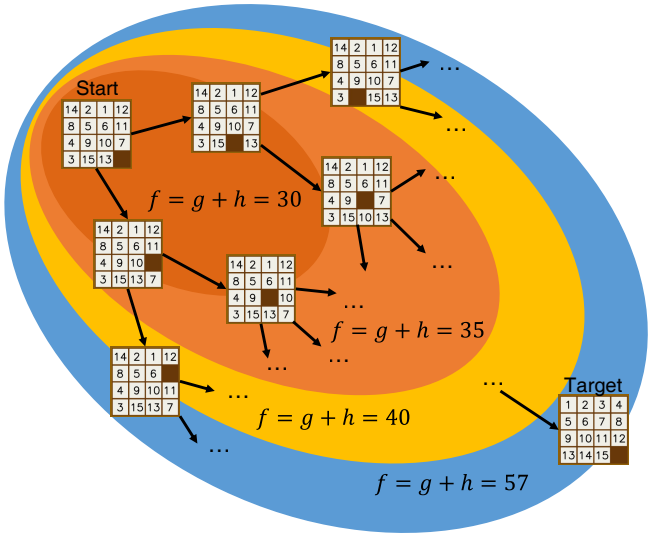
Chapter 3

Neural Heuristic and Bucket priority queue

The 15-puzzle search is a non-trivial problem. It is not feasible to find the best route from the initial board to the goal board through a brute-force search. While solving the N-puzzle, the puzzle's states can be represented as a graph where all edge costs are measured as integer numbers, estimating how far the search has progressed and how many steps are needed to reach the goal. Even with a graph representation of the problem, the state space of the puzzle is tremendously large, and it is difficult to reach the near-optimal path in a reasonable time using limited computing resources.

3.1 Computational Burden of A*

The A* algorithm is a graph search algorithm that finds the optimal path based on two functions: $g(s)$, the sum of the graph edge cost, which is an optimal path from the initial state to the current state, and an ideal heuristic function $\bar{h}(s)$, the sum of the graph edge cost, which is the optimal path from the current state to the target state or from the current state to the goal. However, the cost function $\bar{h}(s)$ is not easily estimated in most problems. Instead, a heuristic function $h(s)$, as an estimated function built via human knowledge, may substitute the ideal heuristic function. An ordered best-searching algorithm, A* searches as an order the minimum $g(s)+h(s)$ that promises the optimal path. As can be seen, $h(s)$ is an important factor affecting search efficiency. When $h(s)$ is smaller than $\bar{h}(s)$, as the gap increases between the two, the number of nodes to search increases. In contrast, when $h(s)$ is larger than the



(a)

Figure 3.1: A* algorithm diagram for 15-puzzle

ideal $\bar{h}(s)$, the search path with A* cannot guarantee an optimal path, because as $h(s)$ becomes smaller than $\bar{h}(s)$, more nodes are searched through the breath-first search, and in the reverse case a greedy search proceeds.

Let $f(s) = g(s) + h(s)$ be an evaluation function for a state or a board configuration s. The A* algorithm finds the best node based on the estimate $f(s)$ and adds the adjacent states of that node to the pool of interest as new nodes to search for. This process is repeated continuously to find the desired goal as shown in Figure 3.1. It is identical to searching in the order of having lower $f(s)$ value to find the target state. The process searches all the state spaces that have a smaller $f(s)$ than the length of optimal path between the start and target state, where the optimal path length is not known in advance. If a heuristic function $h(s)$ is identical to the ideal heuristic function $\bar{h}(s)$, it is possible to efficiently find the optimal path without wasting any calculations. The function $f(s)$ for all the states in the optimal path from the start to the target state is the same as the length of optimal route, and $f(s)$ for the nodes not on the route is higher than the optimal route length. In contrast, if $h(s)$ is smaller than the ideal heuristic function $\bar{h}(s)$, $f(s)$ of nodes that are not in the optimal path can be smaller than the optimal path length, and so a large number of nodes need to be visited.

An obligatory process of the A* algorithm is to visit all the nodes present in the search scope determined by the heuristic function, thus often involving accessing a huge number of nodes. Using the cost function $g(s) + h(s)$, each node sample is evaluated. The

best-first search with the A* tries to find the node with the minimum cost, which is the closest to the goal. A search process generally takes linear time $O(n)$ to find the minimum cost. An alternative method is to use heap as a priority queue, which takes $O(logn)$ to maintain the heap when a new sample node is added and deleted. This is definitely faster than finding the smallest sample in all samples. It is true that this process increases the queue input and queue out of the sample with the number of samples contained in the heap. A significant computational burden is thus incurred in the search process.

The use of a heap structure that increases the burden as this number increases overcomes the time complexity of $O(n)$, which originally occurred structurally. Finding ways to reduce this time further is one of the most important concerns in computer algorithm research. We need optimization techniques from this perspective that we can apply even in very special cases (in this case, N-Puzzle). First, to define the problem of N-Puzzle in detail, all graph edges of the N -Puzzle problem are 1 because the cost of the problem we are trying to solve is the number of slides. That is, in the N-puzzle problem, $g(s)$, the shortest distance between the start state and the current state in all cases, can be treated as a positive integer. Similarly, $h(s)$, which is the shortest distance between the expected target state and the current state, can also be treated as a positive integer. Here, $f(s)$, which must be found in the A* algorithm, is a positive integer because it is given by $g(s)+h(s)$. At this time, A* finds a node that is the smallest key $f(s)$ among nodes with a key $f(s)$ consisting of positive integers. As key $f(s)$ is a positive integer, there are nodes with keys having exactly the same value. We offer these conditions, under which we present a technique for reducing time complexity within the A* search process.

3.2 Conventional N-Puzzle heuristics Limitation

$$\text{Hamming distance}(t,c) = \sum_{i \in N} \begin{cases} 1, & \text{if } t^i = c^i \\ 0, & \text{otherwise} \end{cases} \tag{3.1}$$

$$\text{Manhattan distance}(t,c) = \sum_{i \in N} |t_x^i - c_x^i| + |t_y^i - c_y^i| \tag{3.2}$$

Table 3.1: Experiment cases and optimal path lengths (E: empty tile)

	initial	target	path
Case1	14 2 1 12 8 5 6 11 4 9 10 7 3 15 13 E	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 E	56
Case2	14 2 1 12 8 5 6 11 4 9 10 7 3 15 13 E	10 1 2 6 9 4 5 13 15 11 3 7 14 E 12 8	52
Case3	10 5 2 6 4 14 13 3 1 15 11 9 7 E 12 8	E 5 1 10 13 11 9 2 12 7 4 14 3 15 6 8	50
Case4	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 E	1 14 2 6 9 12 13 8 10 5 15 4 11 E 7 3	46
Case5	12 6 11 13 8 4 15 10 5 2 E 1 9 3 7 14	1 12 14 9 15 6 11 E 4 8 10 7 5 3 13 2	46

$$\text{linear conflict}(t,c) = 2(\text{conflict}_{\text{column}}(t,c) + \text{conflict}_{\text{row}}(t,c)) + \text{MD}(t,c) \qquad (3.3)$$

Possible heuristics that can be used in the 15-puzzle are the Hamming distance and the Manhattan distance. The Hamming distance calculates the count of mismatched tiles (Equation 3.1), while the Manhattan distance calculates the additions of distances between each number at the current state and its position in the target state (Equation 3.2). Specifically, this counts the number of moves along the grid. Another heuristic, called linear conflict, follows the Manhattan distance rule and additionally considers the number of detour moves for passing over each other to reach the goal configuration (Equation 3.3), e.g., if there are two tiles in the same row or the same column but another axis is not arranged.

We selected five cases in which initial state and target state were randomly selected as benchmarks for measuring performance improvement when applying the A* algorithm. The selected cases are presented in Table 3.1.

Figure 3.2 shows a curve of the ideal heuristic, the Hamming distance heuristic, the Manhattan distance heuristic, and the linear conflict heuristic for an example of one sample from the optimal path. Such heuristics get lower than the ideal heuristic when the true distance reaches a certain degree. In Figure 3.2 (a), the Hamming distance is 6, the Manhattan distance is 15, and the linear conflict is 29. In Figure 3.2 (b), the Hamming distance is 4, the Manhattan distance and the linear conflict are each 6, and the conventional heuristics are separated from the ideal heuristic. The search performance of A* will be degraded by a large gap from the ideal heuristic. For Figure 3.2 (a), a detailed interpretation of the diverging points of each heuristic along the real path is given in the Appendix.

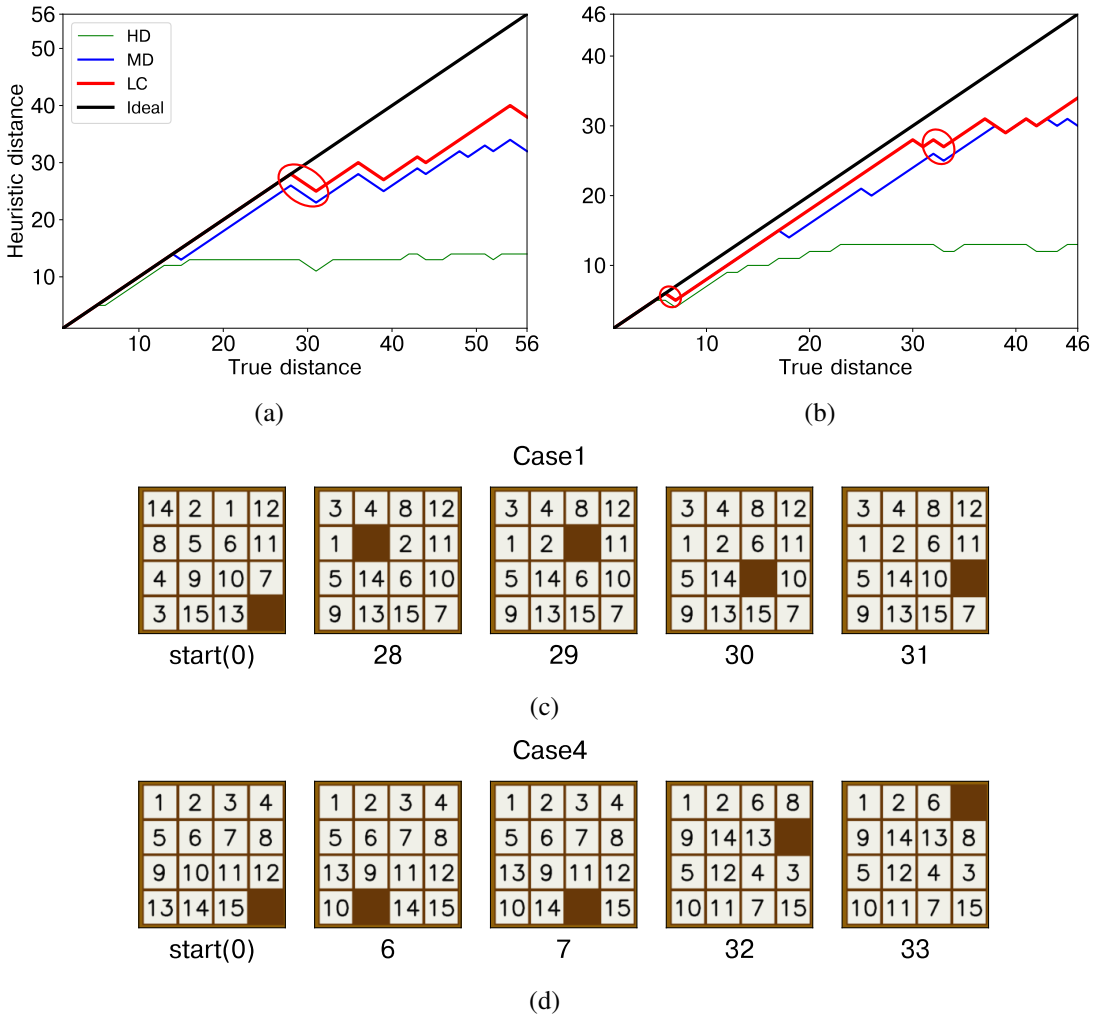


Figure 3.2: Ideal heuristic with the conventional heuristics (a) optimal path for Case 1 (length 56), (b) optimal path for case 4 (length 46), and (c) board examples for Case 1 and Case 4 (from Table 3.1)

In order to solve the problem through A*, a specialized heuristic function, appropriate for the problem, is required. A suitable rule-based heuristic function tailored to these problems is difficult to design. Furthermore, it is almost impossible to follow the actual idealistic heuristics as much as we want using purely human-designed heuristics. Therefore, in his study, we do not use human-constructed rules like the conventional heuristics mentioned above or database solutions. We present a method of learning and applying neural networks to build better heuristics from reliable random data.

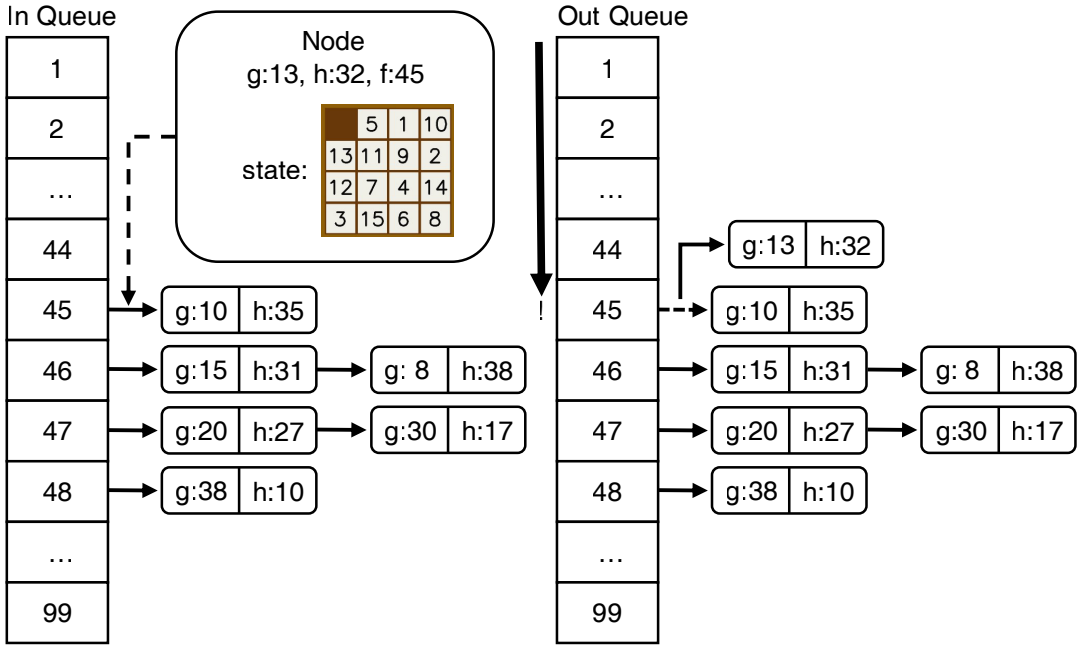


Figure 3.3: Bucket priority queue with the cost

3.3 Method

3.3.1 Bucket Priority Queue

The cost $g(s)+h(s)$, which is an integer number representing distance, will be estimated by the N- puzzle. This distance varies from small to large. A bucket priority queue is proposed for storing costs that are evaluated from the board configuration to reduce the search time for finding the minimum cost. A newly generated board sample with the same cost can be easily inserted into the bucket structure. The cost value f, which is to be sorted, is already defined as an integer value, so it can be accessed directly by the index of the bucket. The linked list or variable length array (deque or list structure in Python) that makes up the bucket takes $O(1)$ time to add the last element. Furthermore, the minimum cost board can be easily fetched. The processes take less time than $O(c)$ where c is the number of buckets as a small constant value because they need to find a minimum element from the bucket by moving to the next slot of the bucket to check the availability of elements whenever a bucket is empty (see Figure 3.3). Again, the linked list or variable length array(deque or list structure in Python) of the bucket takes $O(1)$ time to export the last element. Thus, all graph edge costs and heuristics are integers or can be divided by fixed unit values, and if combined value of $g(s)+h(s)$ is estimated to be limited, we can use a bucket priority queue to process

all sequences in near-constant time complexity. In the case of the 15-puzzle, the expected cost for $g(s)+h(s)$ ranges between 1 to 99. Compared to the heap structure, the computing time of searching for the minimum cost with the method is significantly smaller.

3.3.2 Ideal Heuristic Samples

$$\text{Dataset Sample} = \bigcup_{i,j \in N} \{(s_i, s_j, |i-j|)\} \tag{3.4}$$

An accurate sample of $\bar{h}(s)$ or a set of boards and the cost to the goal are required to imitate the ideal heuristic function $\bar{h}(s)$. Our method is based on learning the shortest distance for each pair of board configurations.

We define the starting state (initial board) as s_i, the target state (goal board) as s_t, and then A* using conventional heuristics, like Manhattan distance or linear conflict. For example, the A* search with a conventional heuristic can find the optimal path from the state s_i to the state s_t, and the optimal path can be parsed as sub-paths. Here, the optimal path from the initial board to the goal board is the shortest route, which differs from the notion of visiting in a particular search sequence of nodes. For an ideal heuristic function, the path and the path segments (a subset of paths) can be collected. The length of the path segment in the collection becomes $h(s)$. Using the distance of nodes in the path, the optimal distance between a pair of nodes in all segments of the paths can be easily calculated. This process harvests a total of N^2 pairs of samples and their shortest distance when an optimal path is given over N length. (Specifically, $N^2 - N$ pairs could be found by eliminating the path between the identical nodes)

3.3.3 Input State Parsing

Using the procedure above, a pair of boards and their optimized shortest distance can be obtained.

Then, neural networks will learn the cost function $h(s)$ for a pair of board nodes. The pair of board configurations are mapped to the number of remaining optimal distances to reach the goal state, by the neural networks. Each pair of boards has a meaning depending on the positional relationship of the numbers. The number representing the board does not adequately represent it. Therefore, the raw pair of boards could be entered just as an array of numbers, not a combination of positions. In that case, as

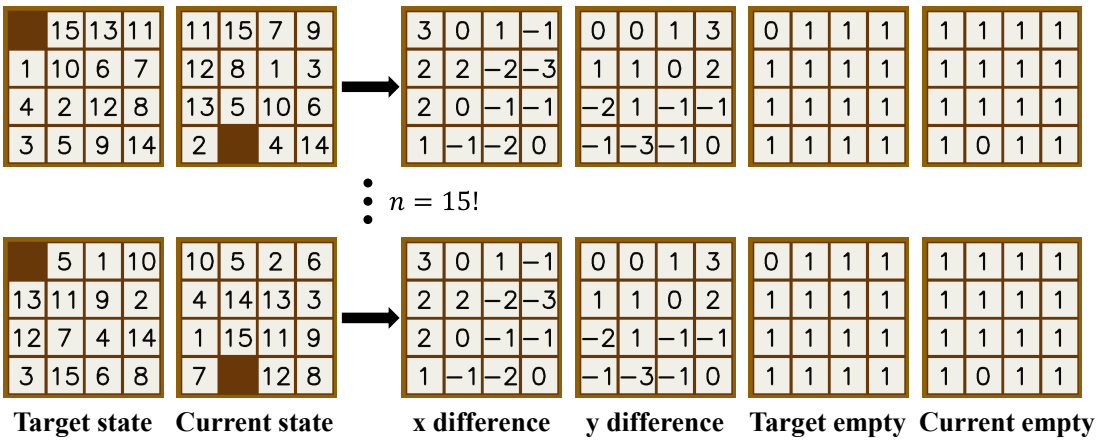


Figure 3.4: Final state parsing for neural network input

the number of states on one board is $16!/2$, the number of states on the pair of nodes to be learned is squared. This exceeds the amount of data that can be collected at a reasonable time because $16!/2$ alone, the number of states on a single board, is huge.

To guide learning about the relationship between a pair of nodes, two types of data are generated: board distance and two empty tile positions. The board distance is the difference of x,y location between the current state and the target state at each number. And two empty positions, i.e., the empty positions of the current board and target boards, the specification of a location on the empty board, which boards are available to slide, or which board is at the corner are considered. A combination of these is shown in Figure 3.4. Through this, it is possible to train heuristics easily by inputting a position relation of each object, not a dispersion of array of numbers of network. In the same time, actually identical states with only permutations, identically presented in the combination we propose, and number of states can be decreased as much as the number of states divided by $(\mathrm{N}-1)!$. By receiving the positional relationship of each object, only the positional relationship can be considered, and the number of cases to be learned can be considerably reduced to $16! \times 4$. Finally, a network input image size of $N \times N \times 4$ is built for learning a pair of boards, i.e. the current board and the target board, and their path length score.

- **Raw 1D** : concatenated flatten by 1d raw Initial and Target state.
- **Raw 2D** : concatenated 2d by raw Initial and Target state
- **Board distance** : 2D information showing the local x, y coordinates that must be moved between each number.

- **Board distance + target empty** : added information indicating the empty location of the target state to the board distance.
- **Board distance + current empty** : added information indicating the empty location of the current state to the board distance.
- **Full** : added information indicating the location of empty in target state and current state to the board distance.

We investigate the effect of the input that the network accepts. In the above cases, networks with various configurations of input information were constructed.

3.3.4 Dataset Configuration

To construct a series of laws through the obtained ideal heuristic, data of scale is required to generalize. Our method is based on the following considerations for effective construction. The neural network must approximate the ideal heuristic with the least deviation through very large amounts of data. In an N-puzzle, data is more likely to be compressed by approaches such as rotation and symmetry. Basically, the pair of current and target states is the square of the state space of the N-puzzle. For this reason, when we construct a heuristic into a dataset, even if a lot of data is collected, it could be a very local subset.

Therefore, we present the following two methods: first, training certain number of fixed data that we shall now call fixed datasets, and second, gathering random shuffled data by each training epoch that we now shall call generated datasets. Using a fixed dataset of sufficient size is a popular method for network learning. As mentioned earlier, fixed datasets could be a local subset, but this is a method that is equally occurring but used in various tasks such as image learning. Therefore, if we collect a sufficient number of data, we can also consider the heuristic learning task. Another way is to use generated datasets instead of real datasets. This is done by randomly generating new data for every single epoch. By doing so, new data is generated on each epoch and losses are minimized on unrepeated data, enabling training of the entire heuristic without specializing on certain fixed cases. However, solving A* at two random points is time-consuming, so the disadvantage is that it may take a little longer time for each epoch than the number of data input.

Furthermore, it takes time to solve A* at two random points, as mentioned earlier. In

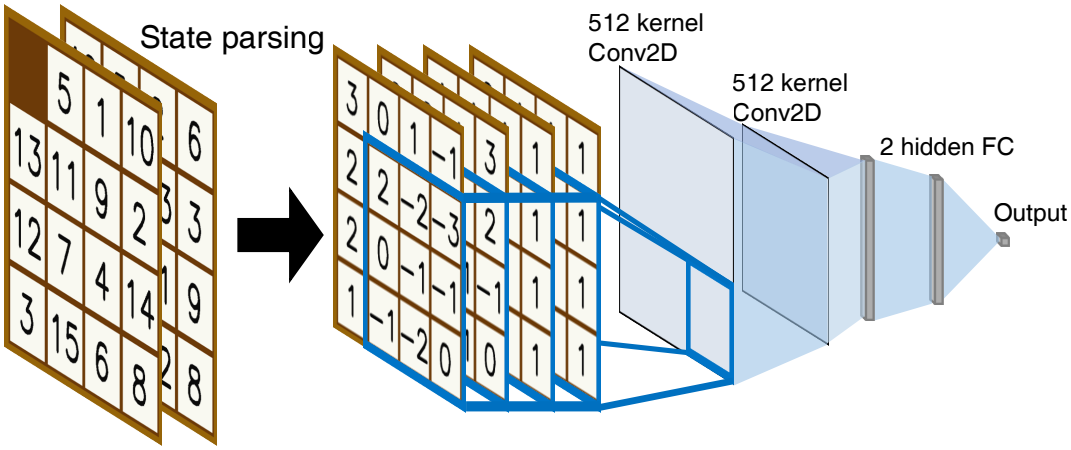


Figure 3.5: Heuristic evaluation overview with largest neural network Net_L

our implementation, if the optimal distance is about 60, it may take about 10 minutes to find the optimal path by linear conflict, and an out-of-memory problem may arise while solving A*. This could be burdensome when collecting datasets or creating datasets every epoch. Thus, when selecting start and target state when generating datasets, a start state is selected completely randomly, while a target state involves sliding 30 times from the start state, shuffling the tiles, and generating data by performing A* search on the chosen start and target state. Additionally, when we built a generated dataset, it was possible to minimize the time taken to generate datasets in training by separating the A* solver and the learning process into multi-processes.

- **Fixed 1M Dataset** : A dataset with a fixed $1e6$(1M) sample collected in advance. Learn all samples in a 128-size batch per epoch.
- **Fixed 100K Dataset** : A dataset with a fixed $1e5$(100K) sample collected in advance. Learn all samples in a 128-size batch per epoch.
- **Fixed 10K Dataset** : A dataset with a fixed $1e4$(10K) sample collected in advance. Learn all samples in a 128-size batch per epoch.
- **Random Generated Dataset** : Random Generation a new sample every epoch. Learn 300 batches of 128 size per epoch.

We also use the above inferences to investigate how many datasets are needed for learning and how accuracy varies in some cases. Learning datasets were divided into the above cases.

Table 3.2: Description of four types of neural networks. (Conv2D: Convolution 2D layer, FC: Fully connected layer)

Layer	Net_L	Net_M	Net_S
1	Conv2D-512 kernel 3x3 filter 1,1 stride same padding	Conv2D-256 kernel 3x3 filter 1,1 stride same padding	Conv2D-256 kernel 3x3 filter 1,1 stride same padding
2	Conv2D-512 kernel 3x3 filter 1,1 stride	Conv2D-256 kernel 3x3 filter 1,1 stride	FC-64 node
3	FC-512 node	FC-64 node	FC-1 node (output)
4	FC-256 node	FC-1 node (output)	
5	FC-1 node (output)		

3.3.5 Network Structure

To handle the board configurations (N-puzzle), an input form of $N \times N \times 4$ is designed and we build a convolution neural network and fully connected network, which has often been observed in deep learning. We have designed three types of 2D input convolution neural networks, consisting of small, medium, and large networks and one type of 1D input fully connected network as presented in Table 3.2 and Figure 3.5. All hidden activation functions in the network use rectified linear activation functions (ReLU) (Glorot et al., 2011). The AdamW optimizer was used (Loshchilov and Hutter, 2017). The output layer is a wide range of regression tasks, so it does not apply activation and outputs it through linear function. We estimated that when the generated $N \times N \times 4$ image is input, local conflict can be detected via a pattern seen in board distance over a 3×3 convolutional network and combined to estimate distance (Krizhevsky et al., 2012).

We have to apply all the padded 3×3 filters per pixel to embed the information of its local pixel location without decreasing the image as it passes through the first layer on the largest network, Net_L and the medium-sized network Net_M. Subsequently, information was collected and synthesized through a 3×3 filter to which padding was not applied, and then regression was performed through the fully connected layer. Net_S is the smallest network that performs convolutional operations, and it was possible to substantially reduce the computation with a single convolutional and fully connected

layer of the same structure. Finally, we construct a network Net_{1d} with three hidden layers consisting of 1000 nodes to verify the performance in a network of similar size when processing in pure array format rather than the state parsing that we designed.

3.3.6 Training loss

The selection of the objectives to learn in neural networks is important. In many cases of neural network learning, performance changes exist. We need to learn the neural network that we have configured for the purpose of estimating distance or heuristics. Most of these regression tasks use the Mean Squared Error (MSE, Equation 3.5), which generates the gradient in proportion to the error size as an objective function, i.e., loss. In order to serve as an effective heuristic, the network is not perfect in a small range, but produces universally precise results. In such a scenario, even if some outliers exist, the overall precision should not be sacrificed to approximate the outliers. Huber loss (Equation 3.6) is a loss that shows robust behavior to these outliers in the regression task that the neural network performs (Huber, 1992).

$$MSE(x) = \frac{1}{n}\sum_{i=1}^{n}(y-x)^2 \tag{3.5}$$

$$Huber(x) = \begin{cases} \frac{1}{2}(y-x)^2, & \text{if } |y-x| \leq \delta \\ \delta|y-x| - \frac{1}{2}\delta^2, & \text{otherwise} \end{cases} \tag{3.6}$$

As given in Equation 3.5 and Equation 3.6, Huber loss, in which the magnitude of gradient is fixed and does not increase after the variable δ, can ignore an outline with a large error, unlike Mean Square error loss, in which the gradient is proportional to the size of the error. In addition, when the error is less than the δ value, the specific gravity is reduced compared to other samples, so the range of errors can be adjusted as much as possible. Therefore, in this method, we train with a δ of 0.1 to make network error smaller than magnitude 0.5, which is the threshold of the rounding network output. Through this, the raw output from the network reduces the proportion of learning to some extent when the magnitude of the error with heuristics is less than or equal to 0.1. The remaining samples can be optimized at the same rate overall, indirectly defining the allowable range of learning errors, thereby ignoring samples that have already been sufficiently learned and learning samples that have not been learned.

Finally, by rounding the output value from the network, the heuristics export integer values to use in the bucket prior queue method that mentioned earlier to be introduced and to be identical to other classical heuristics. The range of errors to be met, such as 0.5 mentioned above, was selected through this process because the range for rounding to an integer is ±0.5.

3.4 Experimental results

To use neural networks in a high-speed implementation for A* search algorithms, programming codes with Python with Numba and JAX are designed, and various control parameters are tested (Lam et al., 2015; Bradbury et al., 2018). The system consists of an AMD Ryzen 9 5950X 16-Core Processor, and the GPU, NVIDIA RTX3080.

3.4.1 Bucket Priority Queue

Table 3.3: Computing time by bucket priority queue and heap structure

	Heap	Bucket	Ratio
Case1	42.3s	37.7s	89.12%
Case2	5.78s	5.32s	92%
Case3	14.1s	12.9s	91.4%
Case4	1.71s	1.6s	93.5%
Case5	1.78s	1.65s	92.6%

First, we compare the bucket priority queue and heap when solving the case introduced in Table 3.1 when using linear conflict heuristics. In both cases, the number of search nodes was the same, and the types of searched nodes were exactly the same. In each case, the time to solve a given case is listed in Table 3.3, and the result was an average value after seven computational time measurements for precise measurements.

Experiments in Table 3.3 indicate that in all cases, the bucket priority queue uses less computational time. Within the investigated case, the computational time ratio of the heap structure and bucket priority queue decreased from 94% to 90%, so the computational time difference increased proportionally as the case was difficult and took a long time to resolve. In this experiment, heap did not show a large computational time in small search numbers because the time required for In Queue and Out Queue

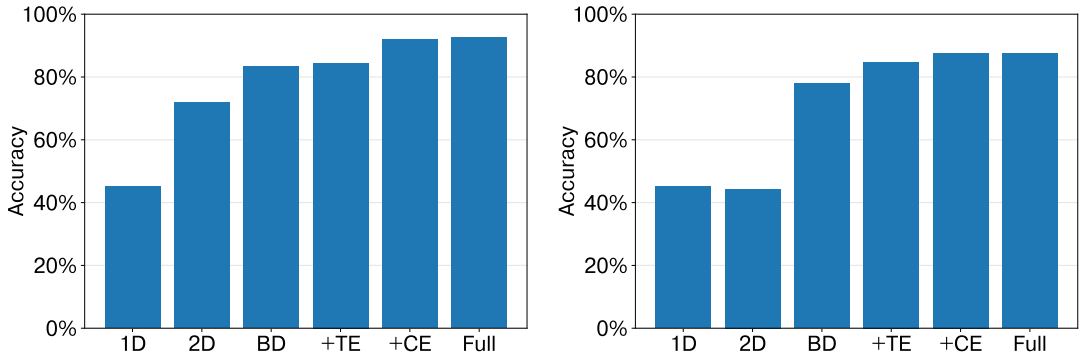


Figure 3.6: Test accuracy comparison; between the network input types at last 100 epoch (a) Net_L Test Accuracy,(b) Net_S Test Accuracy (Full: board distance with current empty and target empty, BD: board distance, +CE: board distance with current empty, +TE: board distance with target empty, 2D: 2D raw input, 1D: 1D raw input)

is $O(log(n))$, but it can be inferred that as its size increases, the time taken in the heap structure will increase, revealing the speed difference between heap and bucket queue.

3.4.2 Test Accuracy by Input State Structure

We conducted the following experiment to determine how much the accuracy of the network varies depending on the type of input data. First, we made sure that the network cannot interpret the positional relationship of each number and that raw input is just a list of numbers. An experiment was conducted on a 1draw connected by flattening the numerical information of the current and target board without processing the numerical information of the current and target board, and a heuristic accumulation when the numerical information of the current and target board was continued in a 2D state. When learning 1d data, the 2D Convolution network was not available, so it was learned through a fully connected Net_{1d} network. The learned results showed no significant change even when the size of nodes or the size of hidden layers was increased. In addition, four cases of combined board distance (stacked x, y different in Figure 3.4, board distance + current empty, board distance + target empty and Full(board distance + current empty + target empty) were tested to analyze the role of board distance and empty position, which are the decomposed features we claimed. At this time, raw 2d, board distance, board distance + current empty, board distance + target empty, and Full inputs can accept 2D convolutional inputs, all learned over the NetL network and NetS network to measure the accretion of 38400 test data for each epoch. The result of the

last learning is shown as Figure 3.6.

As shown in Figure 3.6 (a), the accuracy of 1D input was lowest at the end of learning, and 2D raw using a network with convolution operations shows better results. Board distance, in which raw information is parsed into displacement information of each numeric object, shows better results. Board distance + target empty, which adds the empty position of the target state to the board distance, shows a better tendency accuracy. Board distance + current empty, which adds the empty position of the current state to the board distance, clearly shows better accuracy results. Finally, Full, which is board distance + current empty + target empty, shows a higher tendency accuracy than board distance + current empty.

In Figure 3.6 (a), Net_L 2D raw as an input gives higher accuracy than Net_{1d} 1D raw as an input. In contrast, as seen in Figure 3.6 (a), Net_L 2D raw as an input gives lesser accuracy than Net1d 1D raw as an input. This shows that, unlike Net_L, the size of Net_S is not enough to interpret or map location relationships. It is speculated that the result of using board distance over 1Draw and 2Draw inputs at Net_L and Net_S shows good accuracy by reducing the number of cases to be learned and focusing only on the positional relationship of each object. However, the empty position shows a relatively low accuracy compared to the results used because board distance alone does not know which space is empty, so board distance alone cannot convey all the information of the actual board pair. Board distance + target empty shows lower precision than board distance + current empty. It can be easily inferred that it is much easier to infer the target empty using the current empty than to infer the current empty using the board distance's structure to express the position based on the current state. Finally, although board distance + current empty alone may be sufficient information, it is interpreted that Full tends to have higher expectations because it is more intuitive to input target empty than to infer target empty through current empty in the network for corner state detection. These experiments suggest that using board distance and empty position together is a good input to approximate heuristics.

3.4.3 Dataset Generation

We generate a fixed dataset with samples and sizes of $1M$, $100K$, and $10K$, and a random generation dataset that generates new data for each epoch, as described in the method. Subsequently, in Net_L and Net_S, the input is parsed into the Full input presented by the method. The network learned with fixed datasets confirmed that the ac-

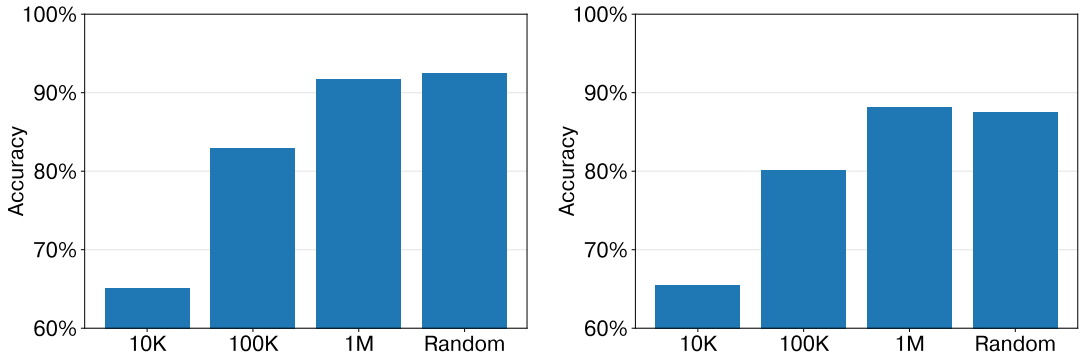


Figure 3.7: Test Accuracy by dataset configuration; Comparison Test Accuracy between Random generation dataset and Fixed dataset at the Last 100 Epoch (a) Net_L Test Accuracy,(b) Net_S Test Accuracy

cretion of the self-learned dataset showed precision above 99.5% before all three cases reached 100 epochs. From this, we can see that Net_L shows a large enough network size for mapping fixed samples we experiment with. Furthermore, the measured accretion achieved 92.5% when learning was conducted by parsing the random generation dataset from Net_L to the final input presented by the method, the same as above during 3000 epochs. This is during the process of learning non-overlapping datasets. When learning the heuristics law, it can be considered that the heuristic law that must be learned was larger than the size that could be expressed by the network, so the accumulation is no longer increasing.

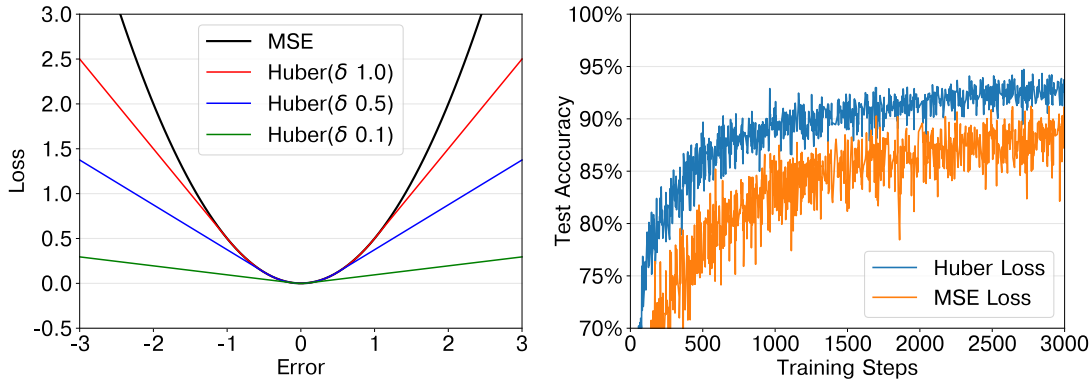
To understand how well Fixed Dataset and Generated Dataset approximate real heuristic laws, we measured Test Accuracy by generating 38400 new unlearned Test samples every epoch in all cases. If the Test Accuracy was measured at the last 100 epochs where each learning was conducted, the results will be as shown in Figure 3.7. Comparing a Fixed dataset of 1 million samples with Random Generated dataset, the Test Accuracies of both cases are shown to be almost identical in the case of Net_L and Net_S. However, in the case of Fixed dataset, the difference between Training and Test Accuracy is massive, so there is a high chance for overfitting. Thus, we decided to use Random Generated dataset, which is basically a fixed dataset of infinite size.

In the following experiments, we compare the learning accuracy when learning using Huber loss and MSE loss with delta values of 0.1 and the three size-specific models Net_L, Net_M, and Net_S accuracy. As for the network input, we used Full input as shown

in the dataset experiment because we checked which data above showed the highest accuracy. In this experiment, we measured the accuracy of 38400 test data per epoch, and the Test Accuracy results measured during the last 100 epochs of learning are shown in Figure 3.9. (a), (b) of Figure 3.9 to bridge the Accuracy between the networks learned by the same model and by the same model. The Test Accuracy in Net_L was 92.5% for Huber loss and 87.5% for MSE loss, in Net_M showed 91.5% for Huber loss and 85% for MSE, and in Net_S showed 86.5% for Huber loss and 82% for MSE loss. Each network model reduced Test Accuracy in the order of L, M, and S. It can be inferred that this is because of fewer parameters to map the learned data by model size or approximate the law of heuristics. Furthermore, the models having various sizes learned using Huber loss showed 5% higher Test Accuracy on average compared to the models learned using MSE loss. As shown in the graph in Figure 3.8 (a), the Huber loss does not increase the gradient no matter how much the regress error is delta or higher, making it more robust for outliers with large regress errors. For this reason, it can be seen that even in the graph in Figure 3.8 (b), which measured epoch-specific Test Accuracy when learning Net_L with Huber and MSE, respectively, the Huber exhibits lower deviations and higher mean Test Accuracy compared to MSE. It can be seen that Huber loss is more suitable than MSE loss only for such heuristic data.

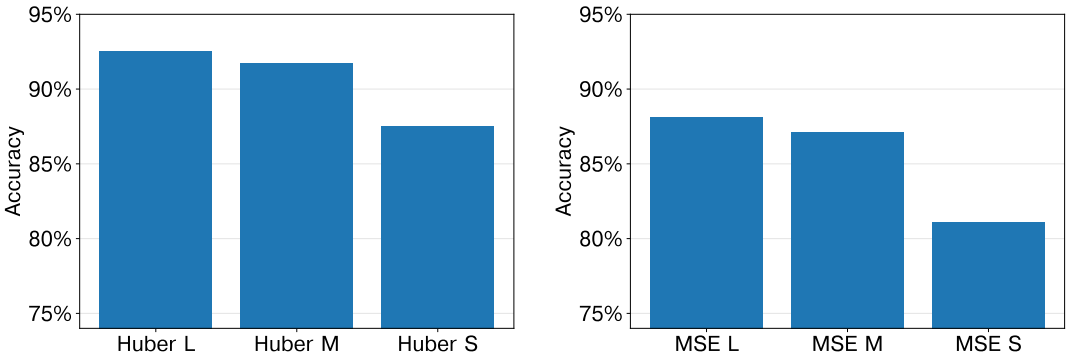
3.4.4 Network Size and Training Loss

In the following experiments, we compare the learning accuracy when learning using Huber loss and MSE loss with delta values of 0.1 and the three size-specific models Net_L, Net_M, and Net_S accuracy. As for the network input, we used Full input as shown in the dataset experiment because we checked which data above showed the highest accuracy. In this experiment, we measured the accuracy of 38400 test data per epoch, and the Test Accuracy results measured during the last 100 epochs of learning are shown in Figure 3.9. (a), (b) of Figure 3.9 to bridge the Accuracy between the networks learned by the same model and by the same model. The Test Accuracy in Net_L was 92.5% for Huber loss and 87.5% for MSE loss, in Net_M showed 91.5% for Huber loss and 85% for MSE, and in Net_S showed 86.5% for Huber loss and 82% for MSE loss. Each network model reduced Test Accuracy in the order of L, M, and S. It can be inferred that this is because of fewer parameters to map the learned data by model size or approximate the law of heuristics. Furthermore, the models having various sizes learned using Huber loss showed 5% higher Test Accuracy on average compared to the



(a) (b)

Figure 3.8: (a) Loss graph by regress error, (b) Net_L Test Accuracy



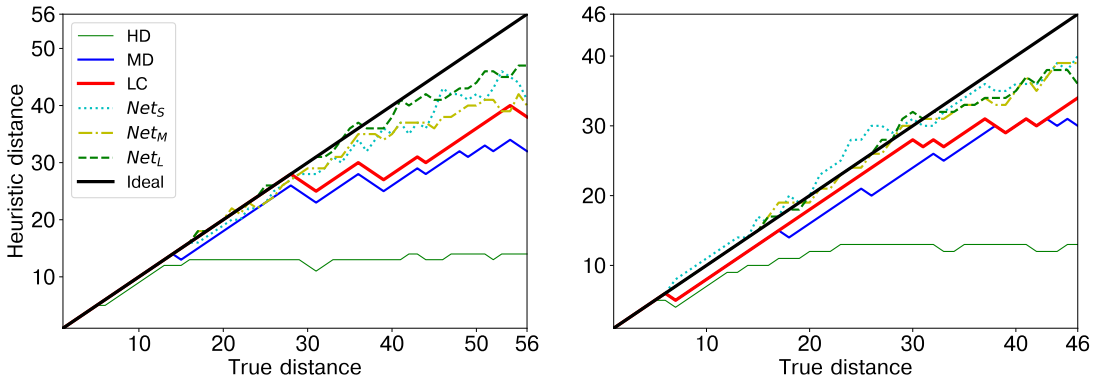
(a) (b)

Figure 3.9: Test accuracy by loss configuration; Comparison Distribution of Test Accuracy between the Huber Loss and MSE Loss at each Network Size for the Last 100 Epoch (a) Trained with Huber Loss(δ:0.1), (b) Trained with MSE Loss

models learned using MSE loss. As shown in the graph in Figure 3.8 (a), the Huber loss does not increase the gradient no matter how much the regress error is delta or higher, making it more robust for outliers with large regress errors. For this reason, it can be seen that even in the graph in Figure 3.8 (b), which measured epoch-specific Test Accuracy when learning Net_L with Huber and MSE, respectively, the Huber exhibits lower deviations and higher mean Test Accuracy compared to MSE. It can be seen that Huber loss is more suitable than MSE loss only for such heuristic data.

3.4.5 Efficiency of Neural Heuristics

First, we check whether the learned network in Figure 3.9 (a) can effectively approximate the ideal heuristics. The proposed neural heuristics were found in Case 1 and



(a) (b)

Figure 3.10: Comparison between the deal heuristic and neural heuristics for random samples (a) optimal path of the case 1(length 56), (b) optimal path of the case 4 (length 46)

Table 3.4: Average computing time for heuristics (MD: Manhattan distance, LC: linear conflict), Net^{cpu}: CPU calculation, Net^{gpu}: GPU accelerated calculation

	MC	LC	Net_L^{cpu}	Net_M^{cpu}	Net_S^{cpu}	Net_L^{gpu}	Net_M^{gpu}	Net_S^{gpu}
time	1.78μs	2.64μs	247μs	113μs	**32.3**μs	148μs	92μs	86μs

Case 4. Table 3.1 presents the results, which are quite close to the ideal heuristics of h(s), as shown in Figure 3.10. It can be seen that the learned neural heuristics follow ideal heuristics that are not only better than Hamming distance but also Manhattan distance or linear conflict. In Figure 3.10, we can see the entanglement of states not considered by linear conflict (shown in Figure 3.2, move 5,6,31,32 from (a), move 28,29,30,31 from (b)). However, neural heuristics detect the entanglement of such states and show a similar pattern to the ideal heuristics. In addition, our dataset eventually collects ideal heuristics from 1 to 30 and proceeds with learning. However, the curves seen in Figure 3.10 showed that each neural heuristic followed the ideal heuristic value to some extent, even at 30 or more points where it would not have been learned. This can be interpreted as meaning that the network has learned the law of actual heuristics through patterns as well as data mapping. However, as explained above, the heuristic cannot guarantee an optimistic solution if it is higher than the ideal value in all cases. Looking at the graph with a small jaw that goes beyond the ideal value, it can be thought that the neural heuristic we present will be a near-optimal solution. In the future, to ensure the optimality of neural heuristics, research needs to be conducted on how to suppress or limit the occurrence of bumps beyond these ideal heuristics.

We found that the Loss experiment and Figure 3.9 (a) show different Test Accuracies for each size of the network. According to the experiment, the larger the size of the model, the more precise heuristic approximation can be obtained. However, the larger the size of the network model, the more computational resources and time are required. Given the characteristics of the A* algorithm, which requires millions of searches to solve a problem, it is difficult to determine for sure which one is more important in terms of the precision and computational time of the heuristic. Therefore, to check the efficiency of the neural heuristic we present, we investigate how long it takes to compute a single board when using the CPU and accelerating to the GPU, and the result is presented in Table 3.4. For this result, the average value was used after measuring 10,000 calculation times for precise measurement.

In this result, the neural heuristics we present depend on the size of the model and the hardware, but they were usually more than 50 times slower than the conventional methods, Manhattan distance, and linear conflict. However, Net_S showed only 12 times the computational time of linear conflict when calculated using only the CPU. In contrast, when using a GPU, it showed a slower calculation time. This can be thought of as a result of the fact that the computation of the Net_S network was small enough to process data in the CPU rather than the time from bottlenecks such as moving CPU memory to GPU memory. As other Net_L and Net_M networks had a relatively large network size compared to Net_S, the computation time accelerated through the GPU was faster than that calculated through the CPU, confirming that it took 148μs and 92μs, respectively, i.e., 35 times longer than linear conflict.

The neural heuristics have a larger computational capacity than conventional heuristics. As can be expected, it has a relatively large computational time. However, the A* algorithm determines the total number of search nodes and search time depending on how well the heuristic is created. In other words, if the number of nodes searched while searching for Path is less than the increased calculation time, it can be seen that the efficiency of the heuristic increases. Accordingly, we investigate the search time, the number of search nodes, and the length of the path solved by the A* algorithm when the case presented in Table 3.1 is solved using each heuristic, and the result is presented in Table 3.5.

In Table 3.5, the computational time varies depending on the case. As linear conflict improved over Manhattan distance, we could see that the A* search was completed in less time It can be seen that Net_S^{cpu} and Net_L^{gpu} solve at a higher rate than conven-

Table 3.5: Search time of search, node, path length for heuristics; (a) Search time, (b) Node that created in search, (c) Length of the path (MD: manhattan distance, LC: linear conflict, -: out of memory)

(a)

	MD	LC	Net_L^{cpu}	Net_M^{cpu}	Net_S^{cpu}	Net_L^{gpu}	Net_M^{gpu}	Net_S^{gpu}
Case1	-	37.7s	34.9s	21.2s	**5.61s**	22s	18s	14.3s
Case2	49s	5.39s	10.1s	4.39s	**2.63s**	6.6s	3.95s	6.83s
Case3	63s	12.9s	3.56s	15.3s	6.22s	**2.22s**	13.4s	15s
Case4	17.6s	1.6s	2.34s	2.77s	**797ms**	1.49s	2.43s	2.08s
Case5	4.68s	1.65s	418ms	1.02s	1.35s	**269ms**	888ms	3.56s

(b)

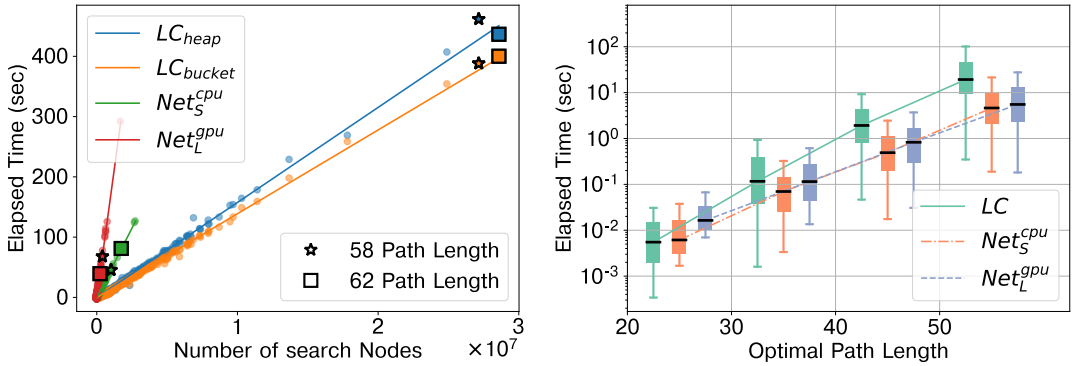
	MD	LC	Net_L	Net_M	Net_S
Case1	-	4.6M	**135K**	168K	144K
Case2	6.8M	676K	39K	**35K**	90K
Case3	3.8M	687K	**14K**	122K	155K
Case4	2.4M	202K	**9K**	22K	20K
Case5	679K	212K	**1.6K**	8.3K	34K

(c)

	MD	LC	Net_L	Net_M	Net_S
Case1	-	56	56	56	56
Case2	52	52	54	52	52
Case3	50	50	50	50	50
Case4	46	46	46	46	46
Case5	46	46	46	46	48

tional linear conflict. Net_L has generated a much lower number of nodes than the node generation of other heuristics in all cases except Case 2. Net_S has a large number of nodes generated compared to Net_L but still maintains a considerably lower number of nodes compared to linear conflict. However, Net_S^{cpu} is much faster in computation than other neural heuristics and outperforms other heuristics in Case 1, Case 2, and Case 4. However, as seen in Table 3.5 (c) for Case 2 or Case 5, it is not guaranteed that A* results by neural heuristics are optimal solutions. As neural heuristics are approximate that they cannot be admissible, as the example shown bump in Figure 3.10. In this case, the number of searches increases, if not compared to the existing heuristics, and the search time increases as well. However, we cannot be perfectly satisfied with that condition mathematically as the neural heuristic is an approximate year. In the future, techniques to suppress that phenomenon are needed.

We also investigated in each case using linear conflict and heap, linear conflict and bucket priority queue, Net_S^{cpu}, and Net_L^{gpu} The A* algorithm takes time to solve and the number of search nodes in the shuffled 10000 random start, target pairs. The result is shown in Figure 3.11.



(a) (b)

Figure 3.11: Comparison at random 10000 samples; Comparison with each heuristics and priority queue as heap and bucket priority queue (a) Scatter plot of elapsed time and number of search nodes, (b) Elapsed time according to the optimal path length expressed in log scale

Figure 3.11 (a) shows the result of comparing the time taken to solve the sample with A* in each case and the number of search nodes. As the time taken to unpack A* is directly proportional to the number of nodes created, the angle of regression line per case increases in proportion to the time it takes to process one node. The angles of regression lines per case increase in order of LC_{bucket}, LC_{heap}, Net_S^{cpu}, and Net_L^{gpu}. This shows that the use of bucket priority queue is more advantageous for searching the smallest sample in a large number of samples than the use of a heap structure using linear conflict heuristics. In addition, as it takes a long time to calculate neural heuristics as experimented with each heuristics calculation time, it can be seen that the angle of these regression lines increases. Nevertheless, we can see that Net_S^{cpu} and Net_L^{gpu} tend to have a low elapsed time overall owing to the small number of search nodes. In addition, a direct comparison of two samples showing the longest optimal path length of 62,58 among the 10000 samples shows obvious differences. In both cases, it can be seen that the number of nodes searched and the time taken when using natural heuristics were significantly lower than when A* search was performed through linear conflict.

Figure 3.11 (b) shows a plot investigating the distribution of time taken to perform A* search for each length of the Optimal Path in each case. In samples with an optimal path length of 20 to 30 or lower, linear conflict finds the solution faster because it takes an average of less than 10 ms to unpack the case in all cases. This is a phenomenon that occurs when the increase in the number of searches due to inefficient heuristics

is less than the computational time taken by the neural heuristics because the number of searches increased owing to inefficient heuristics does not increase so significantly. However, as the optimal path length increases (i.e., as the problem becomes more difficult), the solution distribution of the neural heuristics begins to decrease compared to the linear conflict. In the last 50–62 optimal path length, it can be seen that both cases clearly show a small resolution time distribution. This shows that neural heuristics can be a very effective solution in more difficult problems than the existing rule-based heuristics linear conflict.

In conclusion, Net_L may be recommended if having a lower number of search numbers and memory efficiency are important and computational resources (such as GPU or CPU of many cores) are abundant. Net_L has shown the most efficient navigation with the highest precision, although it is relatively slower than other networks. This is because when using a GPU through this efficiency, it was fast and showed excellent performance compared to other networks. Net_S may be recommended if the computational resources are limited to a small amount (single-core cpu operations), and we want quick results even if substantial memory is consumed. Although Net_S showed less precise navigation compared to other neural heuristics, the computational speed was fast enough to overcome it with the smallest amount of computation and the number of parameters. Through this, three of the five cases presented were solved at an high speed, because it has very high efficiency compared to existing heuristics.

3.5 Summary of Chapter 3

The 15-puzzle is a slide puzzle that moves tiles in the grid world. The best-first search algorithm with A* has been used to solve this problem. In this chapter, we propose a novel approach using a neural heuristic to learn the optimal route from a given board to the goal configuration. To obtain the optimal route, the approach leverages the conventional heuristics, the Manhattan distance, or the linear conflict heuristic. Data samples for the path length estimation are collected from the optimal routes, and then the samples are given to the neural heuristic for learning. The best-first search process tries to find the best route based on this neural heuristic. The neural heuristic is built in the form of deep neural networks.

The A* algorithm can be applied to solve the 15-puzzle, and it ultimately finds the optimal route to the goal configuration. However, its search space using conventional

heuristics is very large, which needs much computing time, as the heuristics roughly sketch the path length to the goal. Our suggested algorithm develops new evaluation functions to determine the cost of more refined search routes with the neural networks as time passes. That is, the neural networks map a pair of board configurations, the initial state and the goal state, to the number of moving steps to the goal state.

Our approach parses a pair of board configurations into the position difference of tiles between the initial and goal configurations and the location of blank space for each board. This helps learn neural networks with better encoding of configuration information.

The bucket priority queue uses a linear array sorted by the cost to the goal, while heap memory uses a tree structure to find the minimal cost of search node. We store the cost to the goal in the priority queue for a given board and manage the minimum cost at every moment of node search. This process can efficiently identify the next search node to be explored. In this chapter, we observed that the A* algorithm with the bucket priority queue considerably reduces the computing time to find the goal node as well as the shortest path to the goal.

Here, instead of using well-known heuristic functions directly for A*, the A* algorithm with the conventional heuristics was run to find the optimal path length for a pair of random boards, the initial configuration and the goal configuration. Then the optimal path obtained from A* was transformed into a dataset for the ideal heuristic. The dataset was used as a training set for deep neural networks, that is, a neural heuristic. We tested three different sizes of deep neural networks consisting of convolution layers and fully connected layers. A large size of networks showed better accuracy performance, while a small size of networks had more efficient computing time.

Finally, the developed neural heuristic becomes close to the ideal heuristic even for goal states requiring many moving steps for the 15-puzzle, compared to the conventional heuristics. The A* puzzle solver with the suggested heuristic often produces optimal routes more efficiently.

In this chapter, we use conventional heuristics, the linear conflict, and the Manhattan distance based on human knowledge to collect data samples for training neural networks. The efficiency of data collection process depends on the heuristics. Possible new path routes found with the neural heuristic could be recorded as another dataset for learning. In the future, incremental learning may be progressed for the neural heuris-

tic, starting from the route samples formed by the Manhattan distance or linear conflict heuristic. Alternatively, a more general form of learning could be tested with a self-organising neural heuristic and its optimal routes, starting from short route paths.

Chapter 4

Deep A* iteration

Table 4.1: Description of four types of neural networks

Net_S	Net_M	Net_L	Resnet
Conv2D-256 kernel 3x3 filter	Conv2D-256 kernel 3x3 filter	Conv2D-512 kernel 3x3 filter	FC-5000 node
FC-64 node	Conv2D-256 kernel 3x3 filter	Conv2D-512 kernel 3x3 filter	FC-1000 node
FC-1 node (output)	FC-64 node	FC-512 node	Res 2x1000
	FC-1 node (output)	FC-256 node	Res 2x1000
		FC-1 node (output)	Res 2x1000
			Res 2x1000
			FC-1 node (output)

In the previous chapter, we extracted the location-related information of each important tile from 15 puzzles to allow it to be processed in small networks, reducing the size of the state space that the network needs to process and giving hints on the actual distance between them. The method is designed to approximate heuristics using patterns of positional relations simply represented by a small number of fully connected hidden layers, after passing the four-channel data extracted from the positional relationship information of the starting and target slide, into one or two convolution networks. This structure can be seen in Figure 4.1. These methods allow us to design networks of three

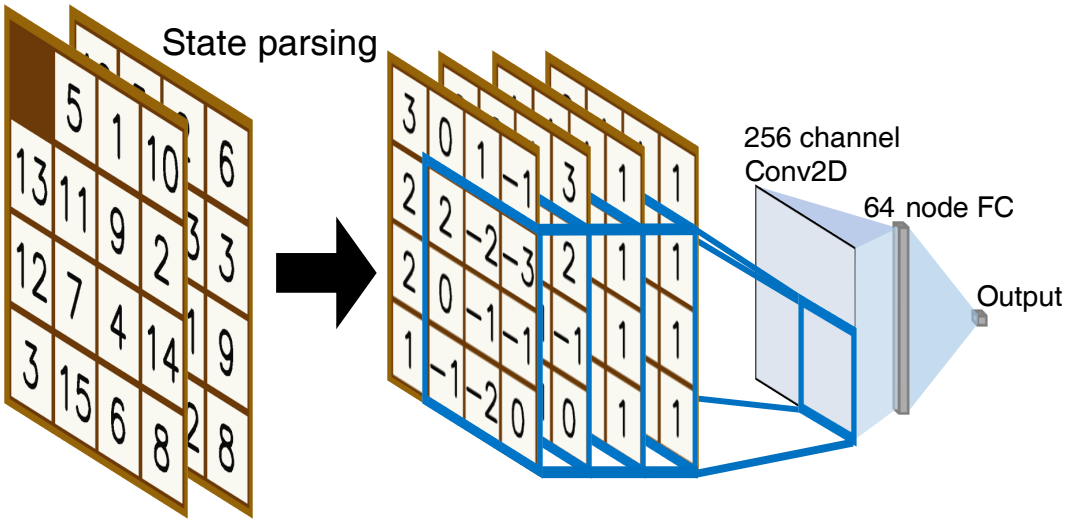


Figure 4.1: Description of the process of estimating heuristics; the process of estimating 15 puzzle's initial and target state's heuristic through Net_S

sizes, Net_L,Net_M, and Net_S. This structure could be compared with Residual networks, presented by DeepCubeA (Agostinelli et al., 2019) in Table 4.1.

As we can see in Table 4.1, we can obtain a relatively precise network despite the very small network size, compared to the Residual network which used four Residual layers, where firstly 5000 nodes were fully connected, secondly 1000 nodes were fully connected. In summary, the Residual network trained in the DeepCubeA paper uses 11 hidden layers with a large number of nodes, but in Table 4.1 above, the smallest network, Net_S, uses only one convolution layer and one fully connected layer, however, there is no major problem to use it as a heuristic. Furthermore, DeepCubeA, which simply receives input from the slide plate, has not been properly established in heuristics for random target points. It is thought that when two plates are input as raw values, the size of the state space increases to a square, so it cannot be properly processed even with a network of existing sizes. Thus in this chapter, we use the corresponding input parsing techniques and networks to speed up heuristic evaluations and achieving a good generalization performance, when solving training times and real samples.

$$P = \{n_1, n_2, n_3, n_4, ...n_k\} = A^*(s,t) \tag{4.1}$$

$$X_P = \bigcup_{n_i \in P} \bigcup_{n_j \in P} (n_i, n_j) \tag{4.2}$$

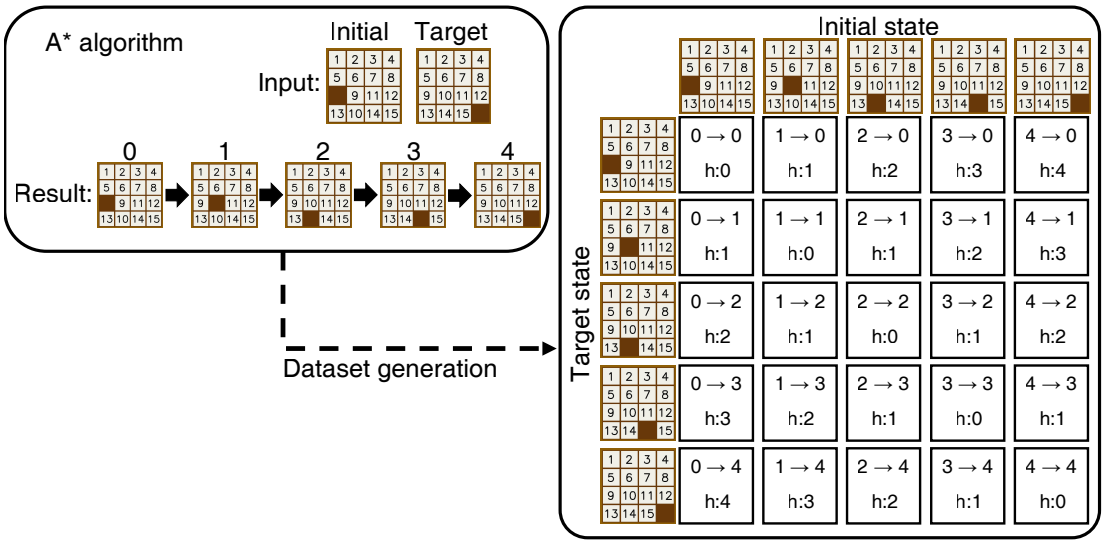


Figure 4.2: Process of gathering an ideal heuristic sample

$$Y_P = \bigcup_{n_i \in P} \bigcup_{n_j \in P} distance(n_i, n_j) \tag{4.3}$$

$$\theta \leftarrow \mathbf{argmin}_\theta Loss(Y_P, H(\theta, X_P)) \tag{4.4}$$

To create a dataset which approximates heuristics, we generated the optimal path of a random sample with a fixed number of shuffles through classical, human-designed heuristics. Then we utilised it as a training data by generating the optimal path distance between all nodes on the path.

First, we define that the heuristic approximation function we are trying to train is $H(\theta)$, which is determined by the network parameter, θ. To start the process described above, as shown in Equation 4.1, we specify random start nodes, s, and target nodes, t, and generate the optimal path, P, obtained through human-designed classical heuristics and A*. Thus generating two-dimensional array representing the distance of each node pairs by extracting all nodes by n_i and n_j, which forms the optimal path, P like Equation 4.2 and 4.3 as well as Figure 4.2. X_P in Equation 4.2 is the set of each pair of nodes in the optimal path, P, and YP in Equation 3 is the distance between each pair of nodes in P. Finally, the parameter, θ, will be trained through Equation 4.4, in order to output a distance between the collected pair of nodes, when a pair of nodes generated in Equation 4.2 and 4.3 will be entered into the heuristic function.

However, this method has a clear limitation due to using conventional heuristics. The

difficulty of constructing human-designed heuristics beyond a certain efficiency to solve A* could be a problem in extending the method to other tasks. For this reason, among the numerous problems, those with theoretically established heuristics aren't that many, so a problem could arise when extending and applying the method to other problems. Although the heuristic has been well-designed due to multiple studies, there still is some inefficiency which leads to more time consumption on conducting A* search for training. Because of this, attempts to collect actual samples are a major obstacle for training due to the decreasing time efficiency. This is mentioned in the paper in which DeepCubeA was proposed. We present the following methods to solve problems that may arise from using these classical heuristics.

4.1 Method

4.1.1 Deep A* iteration

$$P_\theta = \{n_1, n_2, n_3, n_4, ...n_k\} = A^*(s,t,H(\theta)) \tag{4.5}$$

$$X_{P_\theta} = \bigcup_{n_i \in P_\theta} \bigcup_{n_j \in P_\theta} (n_i, n_j) \tag{4.6}$$

$$Y_{P_\theta} = \bigcup_{n_i \in P_\theta} \bigcup_{n_j \in P_\theta} distance(n_i, n_j) \tag{4.7}$$

$$\theta \leftarrow \mathbf{argmin}_\theta Loss(Y_{P_\theta}, H(\theta, X_{P_\theta})) \tag{4.8}$$

In our previous paper, we confirmed that heuristics, which trained on neural networks on complex problems, find faster solutions when compared to classical heuristics. This implies that heuristics trained with neural networks can proceed with data collection faster when we collect information about complex problems.

Thus, to train via network, we specify random start nodes, s, and target nodes, t, similar to the ones in Equation 4.5, to generate optimal paths, P_θ, which were generated by trained network heuristics, $H(\theta)$, and A*. Thus generating a two-dimensional array representing the distance of each node pairs by extracting all nodes by n_i and n_j, which form the optimal path, P_θ, as Equations 4.6, 4.7. X_{P_θ} in Equation 4.6 is the set of each

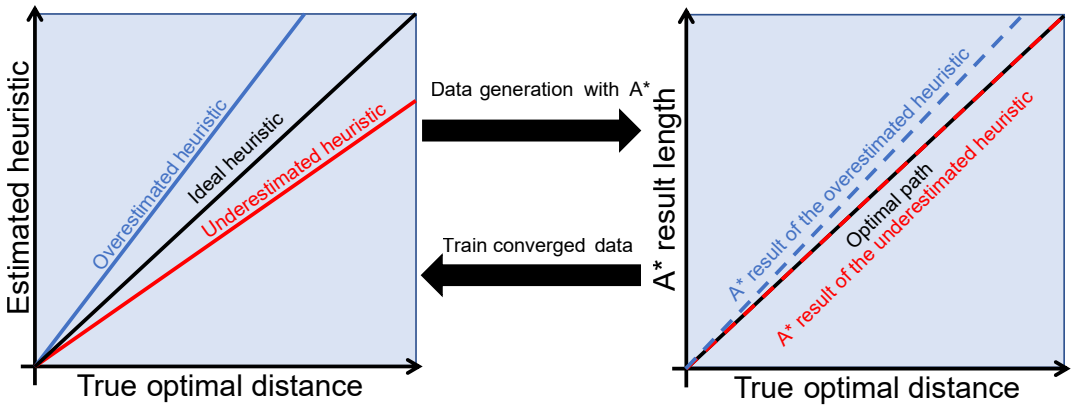


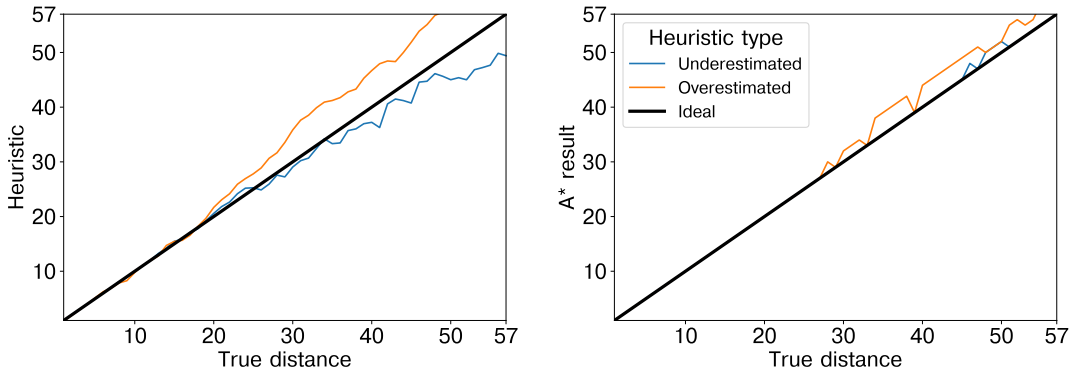
Figure 4.3: Converge process of heuristics; Overestimated heuristic tends to be greedy as a result of A* through the heuristic, however with some breadth-first search characteristics, A* results tend to be smaller than expected, and the underestimated heuristic will naturally converge to an optimal path, even if it takes a lot of time.

pair of nodes in the optimal path, P_θ. Y_{P_θ} in Equation 4.7 is the distance between each pair of nodes in the optimal path, P_θ. Finally, the parameter, θ, will be trained through Equation 4.8, in order to output a distance between the collected pair of nodes, when a pair of nodes generated in Equations 4.6 and 4.8 were entered to the heuristic function.

Theoretically, this allows us to generate data utilizing trained network, and using the iteration method for network heuristics by conducting a reinforcement on itself by training the network again. However, there are two problems with these approaches.

4.1.2 Optimal convergence

The first problem is that heuristics composed of neural networks configured in this way cannot be completely trusted. A* Search is a compromise between a breadth-first search and a greedy search. Although heuristic functions make a search more efficient by playing the role of leading a breadth first search to take a greedy direction. However, if the predicted heuristic value is higher than the ideal heuristic value, the greedy tendency of the search becomes larger than that of the breadth-first search. Therefore, only when heuristics are admissible (there is no case where the calculated heuristics are higher than ideal heuristics in any results), the result of A* guarantees optimal solving. However, a heuristic through a neural network that approximates through data cannot be admissible. This means that naturally, datasets generated by heuristics composed with neural networks, cannot be completely optimal path-based. Although classical heuristics weren't a problem for collecting ideal heuristics samples, they were



(a) (b)

Figure 4.4: Example of converge process of heuristics; Optimal convergence experimented on an optimal path, (a) the prediction of a heuristic function measured on the optimal path, (b) path of the optimal path obtained by each heuristic function with the length of the A* search result to the target point

designed as admissible heuristics. There is a possibility that neural networks cannot be trained by the ideal heuristic value, which is our goal, because neural networks retrain through a dataset that cannot be ensured as composed with the optimal path when data was generated with the neural network. This suggests the possibility that the dataset for training is contaminated, thus cannot train functions that approximate the ideal heuristic.

However, even if heuristics are overestimated and inadmissible, due to the tendency of the breadth-first search of the A* algorithm, the path obtained by A* tends to be a bit closer to the optimal one than the one that heuristics calculated. At this moment, because the network will train data lower than estimation, it will gradually converge to a lower estimation value. On the other hand, the result of A* obtained when it is underestimated and admissible is an optimal solution, thus, converging to ideal heuristics. Therefore, it can be expected that iteration through the network will have a form that converges to the ideal value as the lower bound, and this process can be seen in Figure 4.3. This suggests that the training process can have optimal characteristics when it has sufficient iterations and a reasonable network size.

To check whether this hypothesis is possible we evaluate each nodes: one with a heuristic function that trained the underestimated, another with a heuristic function that trained the overestimated, both in one real optimal path, and check whether it appears as intended when we proceed with a real A* search, as shown in Figure 4.4. The A* search result using the heuristic function, which trained the overestimated, shows

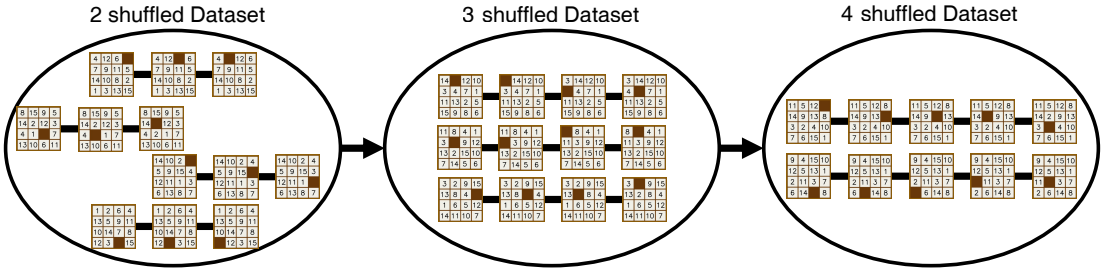


Figure 4.5: Example of curriculum approach

to be closer to the optimal path than our hypothesized prediction, and although there were some parts of the length of the A* search result that were partially different from the optimal path. This was because the heuristic functions that were trained as underestimated have overestimated parts, and are identical with most of the results. Thus, we confirm the possibility that heuristics consisting of networks cannot fully guarantee optimal paths, but as the training progresses, it converges on ideal heuristics rather than diverging into overestimation.

4.1.3 Curriculum approach

The second problem with Deep A*iteration is that networks initialized with random values without training do not produce meaningful results when they operate as heuristics.

When using a randomly initialized network on the heuristic, A* searches without any tendency of solving problems, consuming enormous time and memory just to solve a slightly complex problem. This can be problematic as it takes nearly an infinite amount of time to obtain a dataset for the first training with A* results obtained over neural networks.

To address this problem, we designed a method to gradually increase the scope of problem samples for data generation to collect data in a sufficient time, even with the randomly initialized network. The number of nodes generated and time taken for the A* search solving problem is inversely proportional to the extent of the heuristic's training, and also exponentially proportionate to the length of the optimal path. To solve the problem, and also to prevent the number of generating nodes and time taken from increasing too much, we limit the maximum length of the optimal path by only using shuffled results, which are only shuffled for K times from starting point, s, according to the following Equation 4.9 when generating t as the length between starting point s

to point t.

$$t \leftarrow \textbf{Shuffle}(s, K) \tag{4.9}$$

In this method, we proceed with only one shuffle at first. The first training interaction can be performed without any problems with this easiest difficulty, because we generate data at once even if it becomes a breadth-first search due to that A* heuristic is not trained. In addition, we increase the number of shuffles, K, of problem samples that generate data every time, the training accuracy of the performed iterations exceeds a certain value. This is because if the neural network reaches the anticipated accuracy on the dataset of the current number of shuffles, the next range of samples can also be solved in a short time. Thus, the increasing training range increases to a global range, such as a random sample of the entire environment, so that the overall trend of the environment can be trained.

4.1.4 Algorithm

Algorithm 1 Deep A* Iteration with curriculum approach

Input: I: Training_iterations, C: Curriculum_threshold

initialize network parameter θ
Set $K \leftarrow 1$
for i=1 to I **do**
  Random sample s
  $t \leftarrow \textbf{Shuffle}(s, K)$
  $P_\theta = A^*(s, t, H(\theta))$
  $(X_{P_\theta}, Y_{P_\theta}) \leftarrow \bigcup_{n_i, n_j \in P_\theta} \{((n_i, n_j), |i - j|)\}$
  $\theta \leftarrow \textbf{argmin}_\theta Loss(Y_{P_\theta}, H(\theta, X_{P_\theta}))$
  if Accuracy $\geq C$ **then**
    $K \leftarrow K + 1$
  end if
end for

Return: θ

Through the methodology described above, Algorithm 1 has been designed. The algorithm sets the number of needed sample shuffles, K, when generating data as 1, after initializing network parameter, θ, first. By doing so, we first set up a sample that is easy enough to be solved by an unprepared network, and which slowly increases the maximum length of the shortest path of the sample. In the case of every iteration, while commencing data generation, a heuristic function is provided with an updated

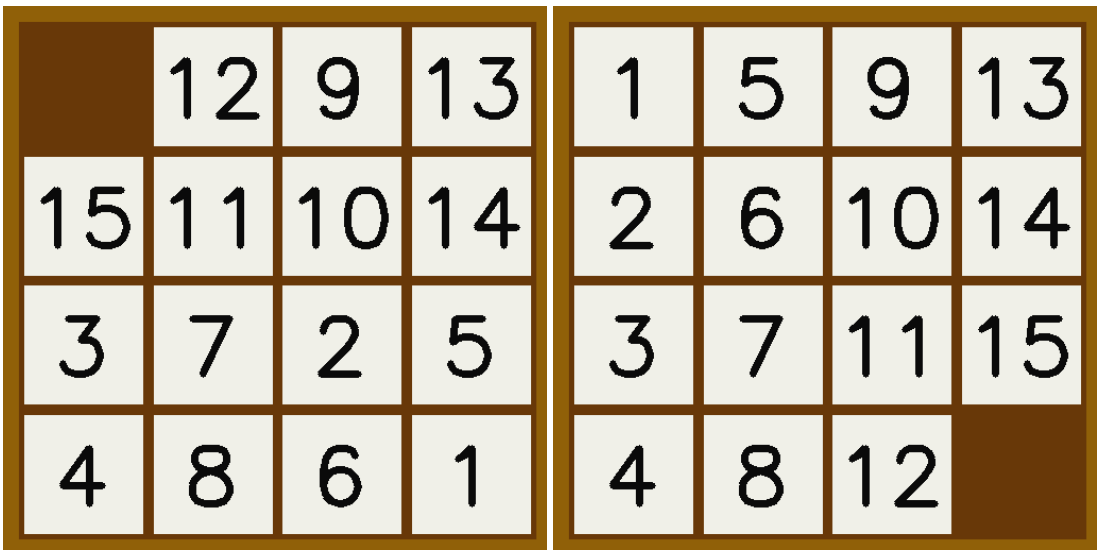
neural network parameter, θ. A* will solve samples of shuffle value, K, which involves the processing path of solved A* value with ideal heuristic dataset to generate X_{P_θ} and Y_{P_θ} values for the network's input and target output respectively. Based on this data, the neural network will train through each iteration by gathering data as much as the training batch size, B.

The data collected through this obtains information on paths closer to the optimal path than the one expected by the network itself. Afterwards, the algorithm will measure the accuracy of the output from parameter heuristic as a same ratio with target output Y_{P_θ}, increase K value by 1 when the value is higher than the Curriculum threshold value C because securing enough accuracy can enable data gathering in a short time although shuffling has increased. Finally, training proceeds as a series of tasks, as defined above, and will progress as much as the number of the iterations, I.

4.2 Experimental results

Finally, we confirm results by adjusting hyper parameters of methods presented above. This is to determine values that produce optimal results, changes made in training time, and the improvement in their performance. This experiment has been conducted for training, using the AMD Ryzen 95950X 16-Core Processor and two NVIDIA RTX3080 as well as four multi-processes to accelerate training data generation. For swift experiments, networks used for training and experiments were Net_S, Net_M, and Net_L, which are described in Table 4.1.

Also, devices running A* heuristics were adapted to operate most efficiently, based on the network size of the model when the data was generated for training data generation and performance measurements. CPU was used for Net_S and GPU was used for Net_M, and Net_L. We collected 200 random samples and optimal paths for the overall performance measurement, to verify how efficiently the proposed network heuristic finds the optimal solution while measuring other performance indicators. The sample includes the most complex case with an optimal path length of 80 for the 15-puzzle, and diagonal inversion case Figure 4.6, where the optimal path length is 72 but the heuristic has difficulties measuring. In both cases, as shown in Figure4.66, it takes two days to solve using heuristics designed in a fairly efficiently optimized C++ implementation, and requires a significant amount of search even in a database-based method. In both cases, Figure 4.6 that takes two days to solve using heuristics designed in a fairly efficiently



(a) (b)

Figure 4.6: Two most difficult cases prepared for evaluation; (a) that has optimal path size of 80, which takes tremendous time using conventional heuristics and database-based method. (b) a case of heuristics, hard to understand, although having an optimal size of 72.

optimized C++ implementation, and requires a significant amount of search even in a database-based method. Therefore, if the trained heuristic solves these problems in a short time, it can be regarded as an indicator that it is a sufficiently practical solution.

4.2.1 Optimal convergence

First, we investigate the number of nodes generated in the A* search and the length value of the path that varies depending on the pattern in which a heuristic appears. Then, we investigate the results of A* search to find the shape of the heuristic and the shortest distance between the starting point and the target point in the shortest paths with a length value of 34 that reach the target point using four classical heuristics composed of human-designed laws. The number of nodes created in the search and the length of the obtained path were thereby measured, and the results are shown in Figure 3.2. The heuristics used are Hamming distance to count the number of missed tiles, Manhattan distance to sum the distances between each target tile, linear conflict to consider the positional collision between each tile, and weight Manhattan distance, which artificially induces the heuristics by multiplying the top and left of the target state by a multiple of 2. As shown in Figures 3.2 (a), (c), and (e) on the left, the heuristic is an admissible one that approaches the length of the optimal path in the order of

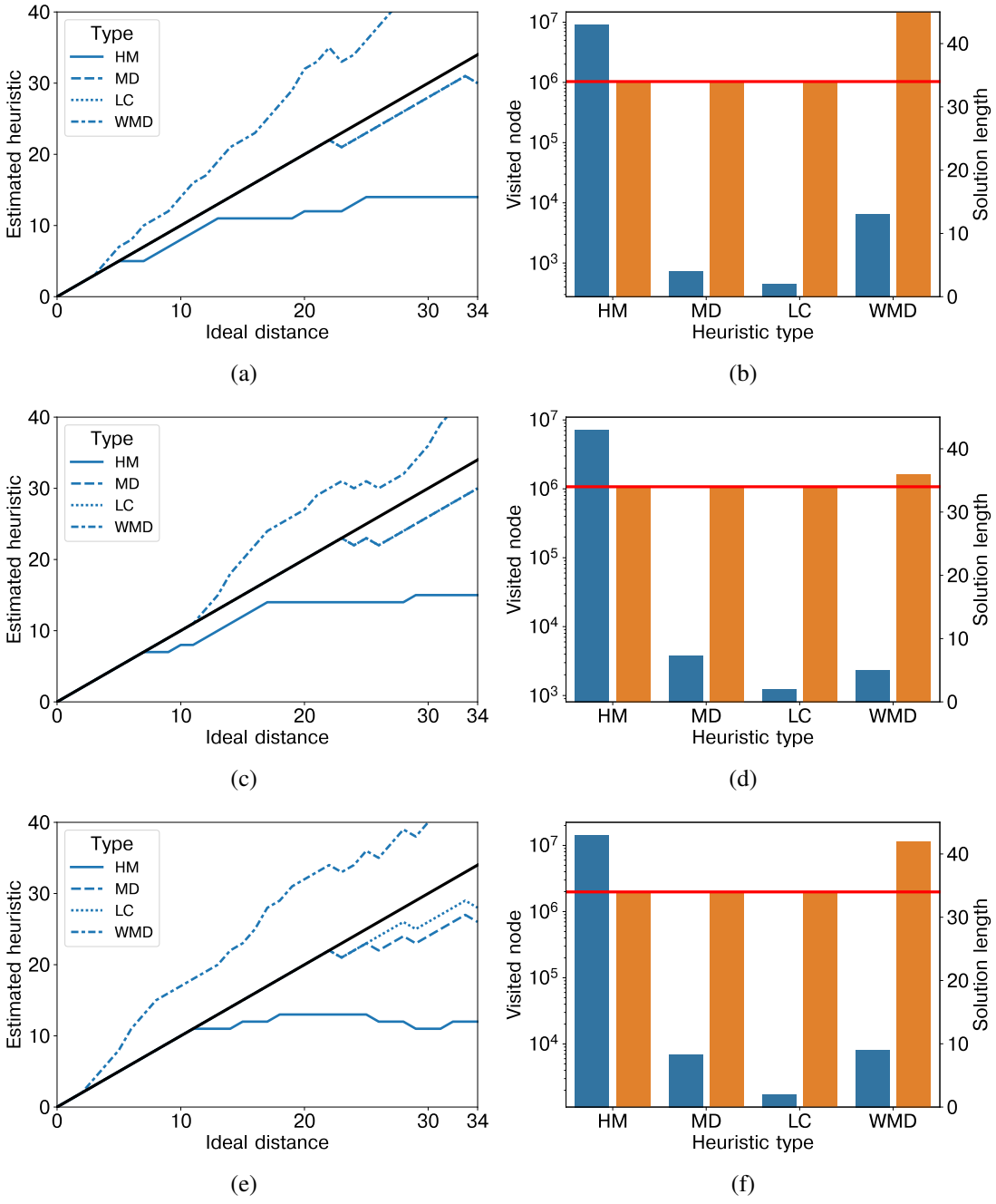


Figure 4.7: Effects of overestimate and underestimate in conventional heuristics: The distance between the start node and optimal route predicted by each heuristic (left); The distance between generated nodes and routes when A* has been applied to each heuristic (right).

Hamming distance, Manhattan distance, and linear conflict. However, weighted Manhattan distance is an inadmissible heuristic as designed. In Figures 3.2 (b), (d), and (f), the A* search, through allowed heuristics, is shorter than the length of the optimal path, showing that the optimal path was found in all cases. The artificially designed weighted Manhattan distance of inadmissible heuristics, on the other hand, did not produce an optimal path. The number of nodes created in the search decreased in the order of heuristic results closest to the length of the optimal path, and an A* search using Hamming distance only found the optimal path after generating 10 million nodes. However, in the case of the weighted Manhattan distance of inadmissible heuristics, the nodes generated did not decrease significantly, even if the heuristics were overestimated and a greedy search was performed. This can be interpreted as meaning that an incorrect greedy search hinders path generation efficiency, and even if it is a heuristic that is not allowed by overestimation, it does mean that the number of search nodes cannot be efficiently reduced if the heuristic does not show the state relationship of the environment well. To sum it up, in this experiment, it can be confirmed that the basis for the heuristic to determine the efficiency of A* search lies in how well the environment is interpreted and is linearly proportional to the actual distance.

As suggested, we confirmed that the overestimated heuristics and the underestimated heuristics converge to ideal values as the iteration continues. First, we apply two pretrained network parameters to the models Net_S, Net_M, and Net_L, one trained to prioritize overestimation and the other trained to prioritize underestimation, and then randomly sample the start and target nodes to proceed with deep A* iteration. The results are shown in Figure 4.8 and 4.9. In this experiment, we collect 38400 data points per iteration, performing 300 mini-batch trainings each of size 128, and finally training four cases per iteration. The estimation error mentioned in the experimental results is the difference between the actual optimal path distance and the heuristic values predicted by the trained network in the 200 random samples mentioned above. With this value, we can check whether the trained network can approximate the actual heuristic.

First, looking at the result in Figure 4.8, when the value of the error is trained to be positive, the distribution of heuristic errors in all networks is biased to positive numbers. This means that most searches proceed with greedy searches, so it is less likely that the searched path is the optimal path. Furthermore, at iteration zero, as shown in the graph of Figure 4.8 (d), the results of A* search find only 20% of optimal paths in models of all sizes. When training is carried out in the manner presented,

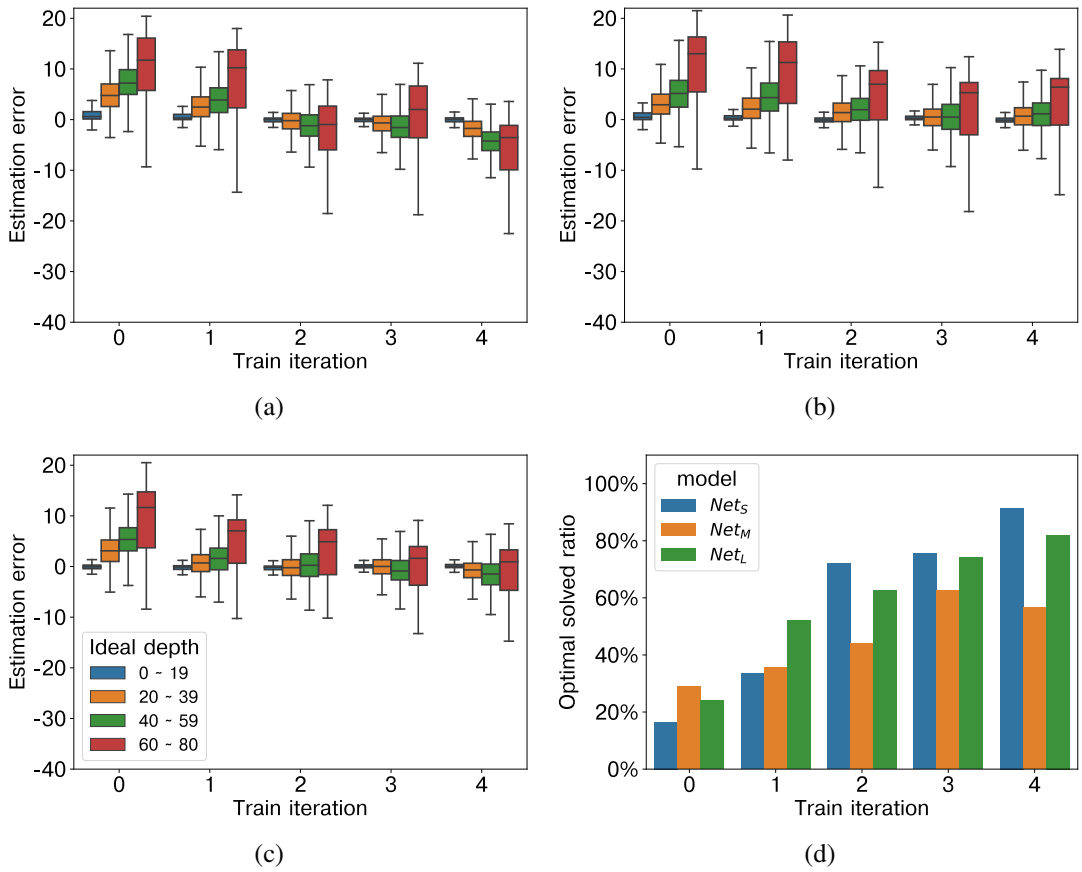


Figure 4.8: Convergence process of overestimated heuristics: An error with the ideal heuristic was measured. (a) Net_S, (b) Net_M, (c) Net_L, (d) ratio of the optimal path increases each time the iteration proceeds.

the distribution of heuristic errors approaches zero with every increase in duration in networks of all sizes. Therefore, the chances that the searched path is the optimal path in A* search tend to increase continuously as the iteration increases. Thus, finally in four iterations, about 80% resulted in the optimal path.

When the result of Figure 4.9 is underestimated, it is clear that the distribution of heuristic errors has shifted to negative on all networks at the zero-iteration interval where training has not progressed further. This means that most searches consist of a breadth-first search, so the number of nodes created is very large until the search is completed. As shown in the graph of Figure 4.9 (d), it can be seen that a very large number of search nodes occur at iteration zero. At this time, if training is carried out in the manner presented, it can be confirmed that the distribution of heuristic errors continues to increase from negative values every time the iteration increases in networks of all sizes, approaching zero. As a result, the number of nodes created while

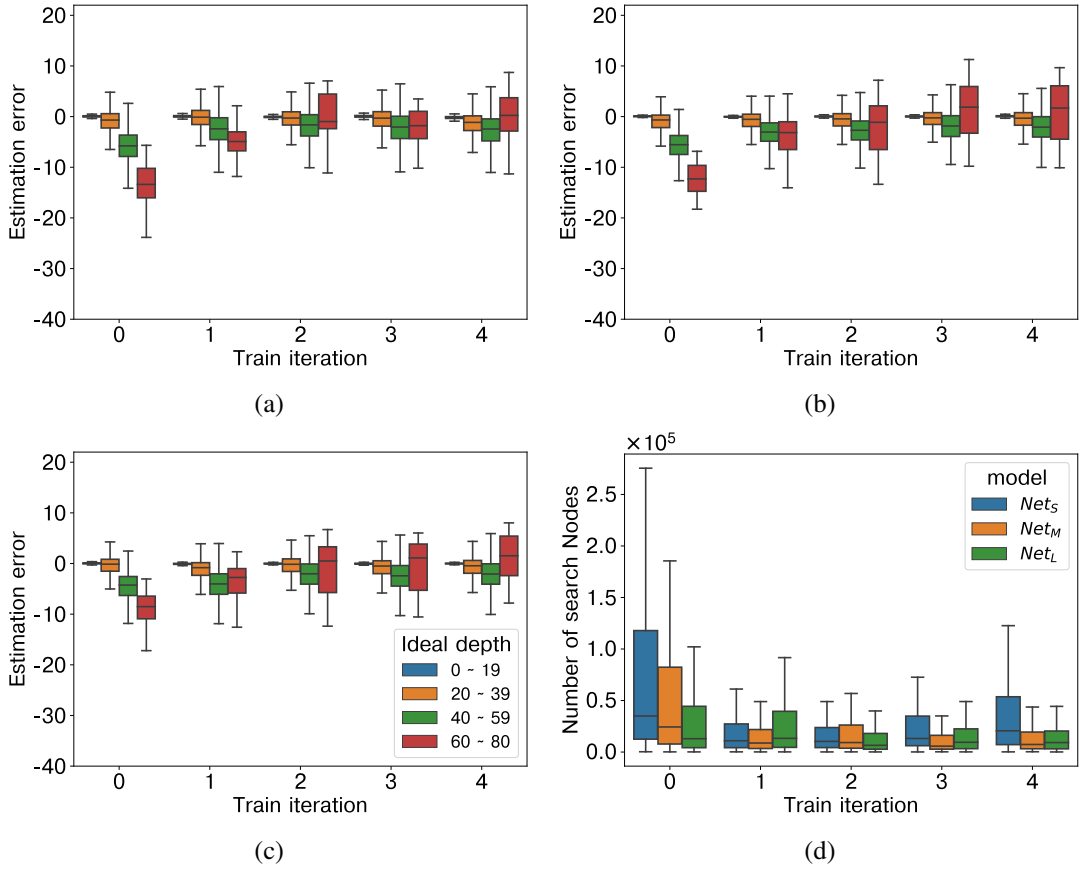


Figure 4.9: Convergence process of underestimated heuristics: An error with the ideal heuristic was measured. (a) Net_S, (b) Net_M, (c) Net_L, (d) the number of nodes searching to find a path to the goal decreases each time the iteration proceeds.

searching for the optimal path in A* search tended to decrease continuously whenever it was automatically increased.

To confirm this tendency, we aim to solve 200 problem samples fixed in Figure 4.10, proceed with training in two cases, overestimated and underestimated networks, and analyze the optimal path of the target sample and the process of changes in the predicted heuristic. In this experiment, 200 pairs of fixed start and target points per iteration were solved, and then the generated heuristic dataset was divided into a mini-batch of size 128 to finally train 100 iterations. Figure 4.10 (a) and (b) show how each network changes as training progresses when 80 slides are the optimal path, and (c) and (d) show how each network changes as training progresses when 72 slides are optimal paths.

First, in Figure (a), when 80 slides were the optimal path, the network that was overestimated at iteration zero without additional training showed a much higher prediction

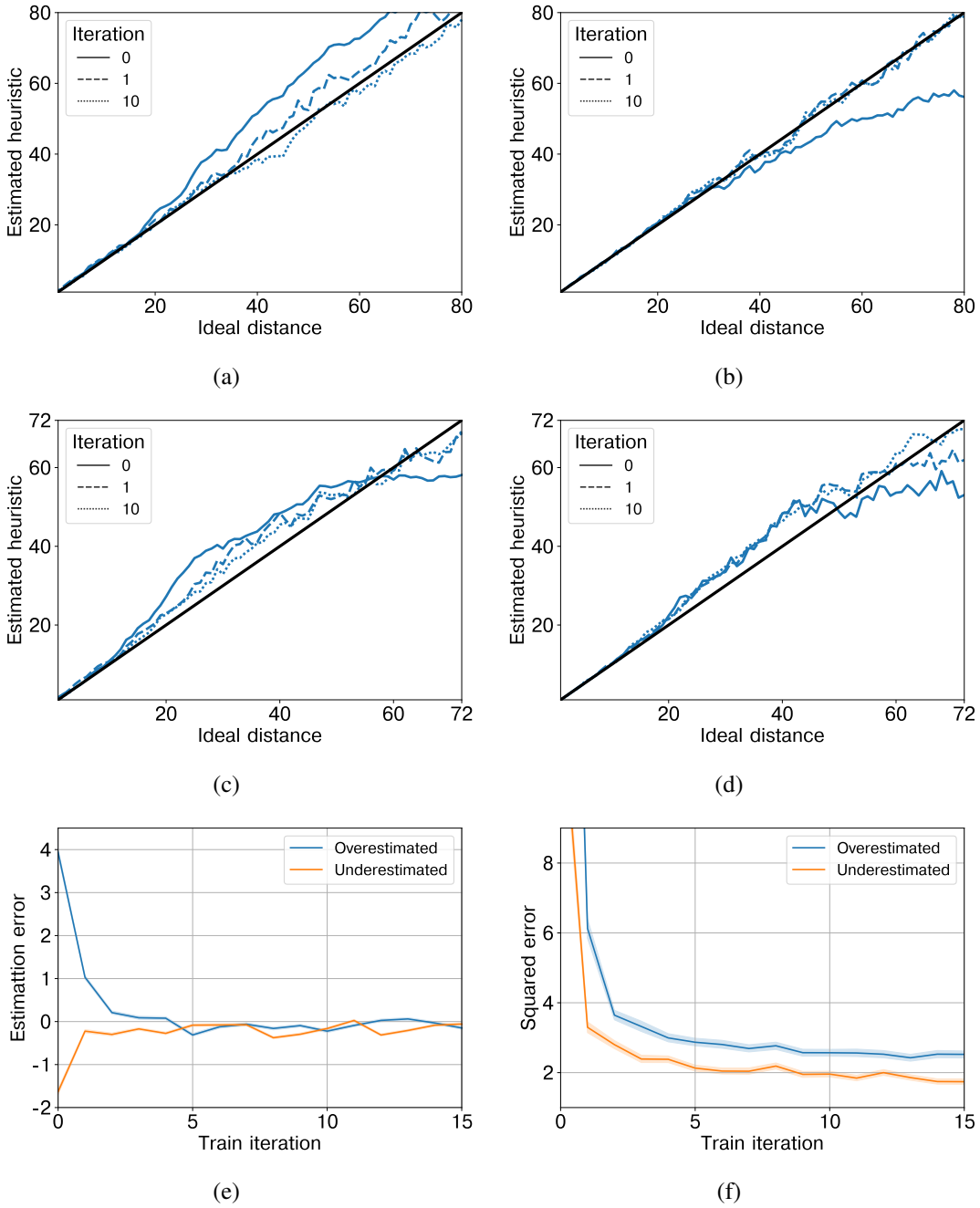
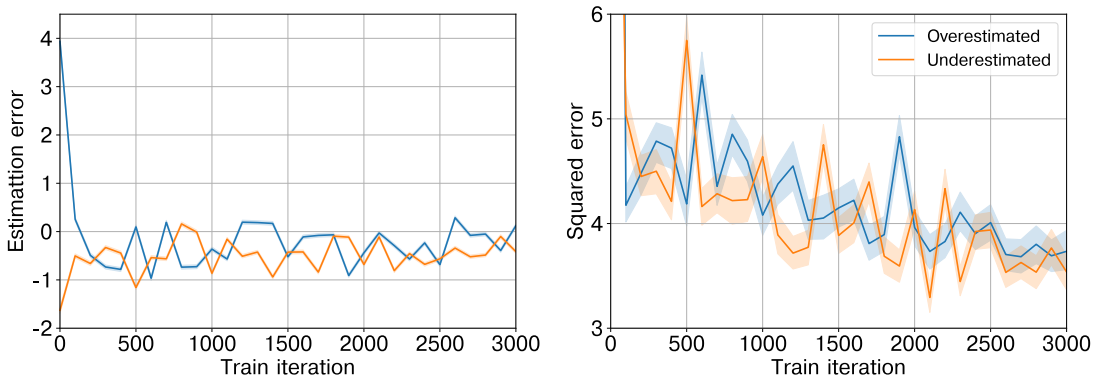


Figure 4.10: Predicted heuristic values converge on each iteration; training is performed on 200 fixed samples. Overestimated heuristics converge at optimal path lengths (left), Underestimated heuristics converge at optimal path lengths (right); (e) Average of heuristic prediction errors per training with overestimated and underestimated heuristics, (f) Average of heuristic prediction squared errors per training with overestimated and underestimated heuristics.



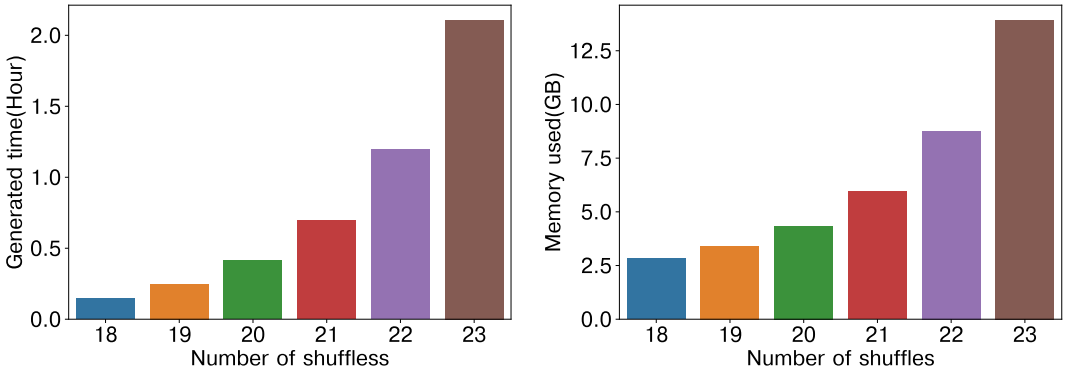
(a) (b)

Figure 4.11: Predicted heuristic values converge on each iteration; training is performed on 200 fixed samples with overestimated and underestimated heuristics; (a) Average of heuristic prediction errors per training session, (b) Average of squared errors per training session.

than the actual optimal path distance, the ideal distance. However, after one iteration, it trained closer to the ideal distance, and after completing 100 iterations, predicted heuristics were almost the same as ideal distance. In addition, in Figure (b), when 80 slides were the optimal path, the network that was underestimated at iteration zero, where no additional training was performed, predicted a much lower distance than the actual optimal path distance, the ideal distance. Heuristics, on the other hand, were similarly trained in ideal distance from the first iteration, and after 100 iterations, errors in the form of spikes that existed slightly in the first iteration were reduced.

Next, in Figures (c) and (d), when slide 72 were the optimal path, neither the overestimated network nor the underestimated network has successfully progressed in the prediction. This may be due to the number of cases with heterogeneous patterns that differ greatly from the data but had previously trained both networks. However, as the training progresses, the underestimated part increases and the overestimated part decreases, and as the iteration progresses, it clearly shows that it becomes a straight line close to the ideal distance. In Figures (e) and (f), we measure and plot the error of the predicted heuristic per iteration in all samples as well as the above two samples in Net_S. As shown in the figure, it is clear that both the overestimated network and the underestimated network converge to a value of zero. This shows that the Deep A*iteration we present tends to converge to an ideal heuristic.

In addition, experiments were conducted (see Figure 4.11) to check whether heuristics converge even when training is conducted in random cases rather than fixed samples.



(a) (b)

Figure 4.12: The resources required to generate data for the first iteration in each number of shuffles; (a) Time and (b) Memory for data generation.

In this experiment, 38400 data points were collected per iteration, which a mini-batch training of 128 sizes was performed 300 times to finally train 3000 iterations, and the average and squared average of estimation errors were recorded in the network trained every 100 iterations. The results in (a) show that the estimation error is vibrating due to the influence of random samples, but the square average value of (b) converges to a constant value every time the iteration proceeds, and an ideal heuristic is expected to be predicted if the training is sufficiently repeated.

4.2.2 Curriculum approach

We see that the Deep A*iteration presented in the experiment above converges into an ideal heuristic, which can be used for the A* algorithm, regardless of the training form of the network. However, in actual training, whether the required training time and memory usage are sensible is a very different story. Since the method presented above can generate data only after completing the A* search, the path to the goal should be generated without any hints if a heuristic has not been trained. These searches must find a number of nodes exponentially proportional to the number of shuffles of the start and target nodes, which consumes enormous time and memory in the search, making the training impossible to proceed.

To this end, we initialized the smallest network Net_S, fixed a constant number of shuffles of samples to be trained, performed an A* search, and recorded the maximum memory size increasing as nodes were created, as shown in Figure 4.12. The time and memory size at which the first iteration is trained according to the number of shuffles

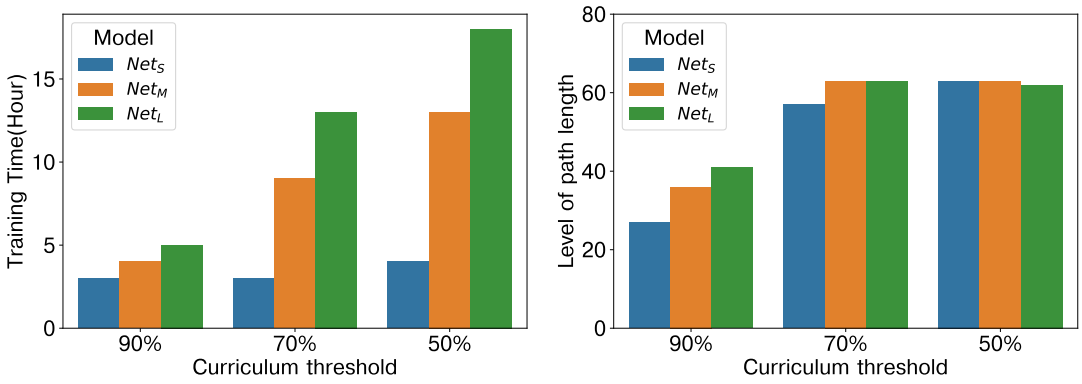


Figure 4.13: Training information according to curriculum threshold; (a) Training time for each model according to the curriculum threshold value, (b) Highest difficulty of sample that was generated using the curriculum threshold value.

showed an exponential increase, as we predicted. The time and memory size required for the first iteration increased by about 1.7 times for each increase in the number of shuffles, and it took about 2 hours for the number of shuffles to reach 23, and the memory size occupied a capacity of 13 GB when the number of shuffles reached 23.

In all cases, the difficulty of 15-puzzle has an average value of 52. Assuming that the time increases at the same rate as in the experiment above, it takes about 35 million hours to collect data from a sample with an average difficulty of 52 times, and 200 petabytes of memory is expected. For this reason, it is virtually impossible to solve and collect data with difficulty enough to easily solve 15-puzzle in all cases. As a result, it is nearly impossible to solve and collect data with enough difficulty to easily solve a 15-puzzle problem in all cases. In fact, based on the number of shuffles over 23, as shown in the experiment, training was not possible with the current experimental equipment due to training time and memory usage.

With this experiment, we can confirm that there is a limit on obtaining the full range of data to be solved without underlying knowledge, such as trained networks or classical heuristics. That is why, as we suggested in the Method part, we can estimate the sampling range that we think can be solved even in the currently trained network based on accuracy, an objective indicator of how well we trained the obtained data within the generation range of the current data, thus confirming the need for a technique to gradually increase the sampling range.

We collect a sample of training data in an easily solvable range of 15 puzzle, which

reduces the time it takes to collect data across the entire range as much as possible. Methodically, K is used to gradually increase the range of trained data samples in an easily solvable range by the currently trained network, investigating how much the training time has been reduced and how much the range of data gathering has increased.

First, in Figure 4.13, we investigated how long it takes to train according to the parameter curriculum threshold (C) used by the method, and what is the sampling range used for training in the last iteration. In this experiment, we collected 38400 data points per iteration and performed 300 mini-batch training of 128 sizes to finally train 3000 iterations, measured the highest target value from the sampled data in the last iteration, and the time it takes to train up to 3000 iterations. The curriculum thresholds (C) used in the experiment were set to 90%, 70%, and 50%, respectively.

The smallest-sized network, Net_S, completed a training of 3000 iterations within 4 hours if the curriculum thresholds (C) were 90%, 70%, and 50%; the highest target value of 90% in the last iteration was 27, 57 for 70%, 62 for 50%. Next, training 3000 iterations on a large network, Net_M, took 4 hours when curriculum threshold (C) was 90%, 8 hours when 70%, and 13 hours when 50%. Furthermore, the highest target values in the last iteration were 38 at 90%, 62 for both 70%, and 50%. Finally, training 3000 iterations on the largest network Net_L took 5 hours when the curriculum threshold (C) was 90%, 70% took 13 hours, and 50% took about 18 hours. And the highest target values in the last term were 42 at 90%, 70% and 50% are both higher than 60.

Overall, this gradual increase in the sampling range of the curriculum approach can result in a maximum target value of above 60 on average. Since the optimal path length of the 15-puzzle problem is a normal distribution with an average value of 52.2, sampling the problem in the same range as completely randomized sampling requires a maximum target value of 60. It can be said that our method was able to complete the training within a reasonable time while training by collecting global samples of the environment called 15-puzzle.

Additionally, while analyzing the increase in training time for each model a large variation was found, but the lower the curriculum threshold value, the higher the time required to perform 3000 iterations. The smallest network Net_S used different CPU cores to calculate heuristics of A* in the data generation process, resulting in less bottlenecks and very little evaluation time for heuristics; thus, the training time did not

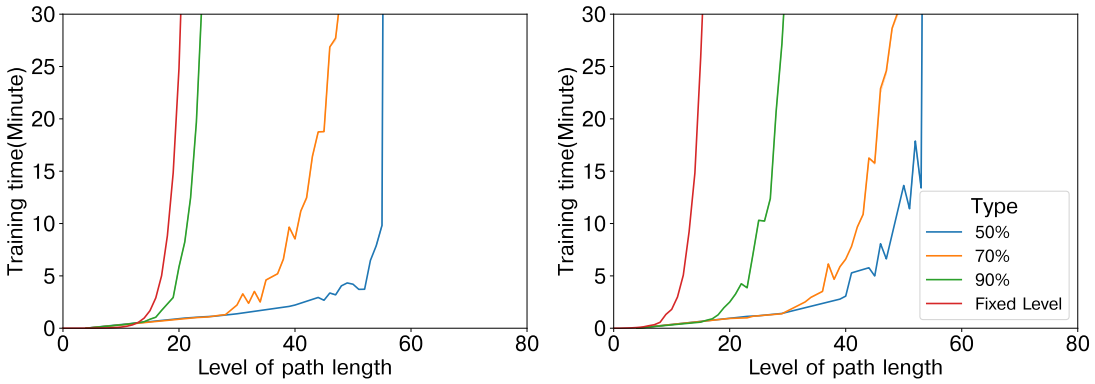


Figure 4.14: Time takes to generate data up to a certain path length; (a) Net_S, (b) Net_M

increase significantly. However, in the other two networks Net_M and Net_L, the training time increased significantly when the curriculum threshold value became small due to inter-process interference using GPU and long time for evaluating GPU bottleneck and individual heuristics.

Also, to clearly show the difference between fixed-level training and curriculum method training, data collected from the networks Net_S and Net_M was used to compare the time taken to generate data of a certain maximum target value in both cases in Figure 4.14. The case of training with a fixed level in both Figure (a) resulting from a small-sized network, Net_S, and Figure (b) resulting from a medium-sized network, Net_M, shows that although the maximum target value does not exceed a value of 20, the training time shows a very exponential increase. However, the cases in which training was conducted through the curriculum method increase to a relatively high maximum target value, and it was confirmed that there was a very large improvement in the training sample range and sample creation time ratio.

We then draw a heuristic plot by which the network Net_S is trained on a 50% curriculum threshold to determine if the network is trained to be the same as the optimal path as suggested above, and the result is shown in Figure 4.15. If we analyse each case: (a) when the optimal path length is 56, the estimated value of neural network heuristic increases to approach the optimal path as the iteration increases. At the time of 100th iteration, the graph is almost identical to the length of the optimal path except for the last section of 50 or more. Subsequently, even in (b) with an optimal path length of 62, as the iteration increases, the predicted value of the neural network heuristic that is being trained increases to approach the optimal path, but is trained to be partially

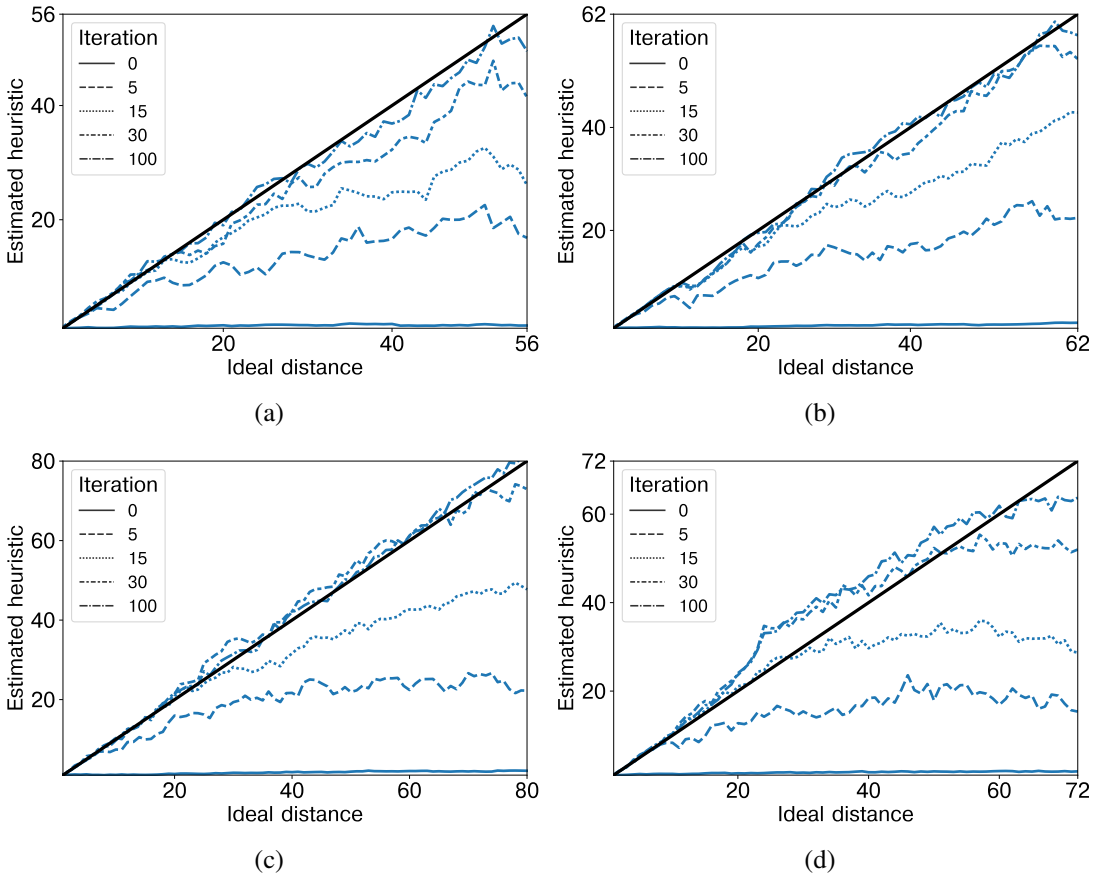


Figure 4.15: Processes that heuristic increases to ideal heuristic; results are measured in four optimal paths (a) length 56 case, (b) length 62 case, (c) length 80 (longest) case, (d) length 72 case

overestimated at 30 iterations.

However, at the time of the 100th iteration at the end of the training, the overestimated part decreases to be confirmed as drawing an almost similar graph to the length of the optimal path. Subsequently, in (c) when the optimal path length was 80, the predictive value of the neural network heuristic increases, as the duration was increased and showed a graph similar to the length of the optimal path, but became slightly overestimated at the time of the 100th iteration. Lastly, in (d) when the optimal path length was 72, the graph of the neural network heuristics trains gradually increases as iteration increases. At the time of 30th iteration, the first half of the evaluation by heuristics was very overestimated and the second half was very underestimated, and then at the time of 100th iteration, a convex shaped graph was seen as overestimated. However, in both cases, much straighter graphs have been drawn than in the previous ones, which suggests that if we have sufficient network size and training iterations, it's possible to

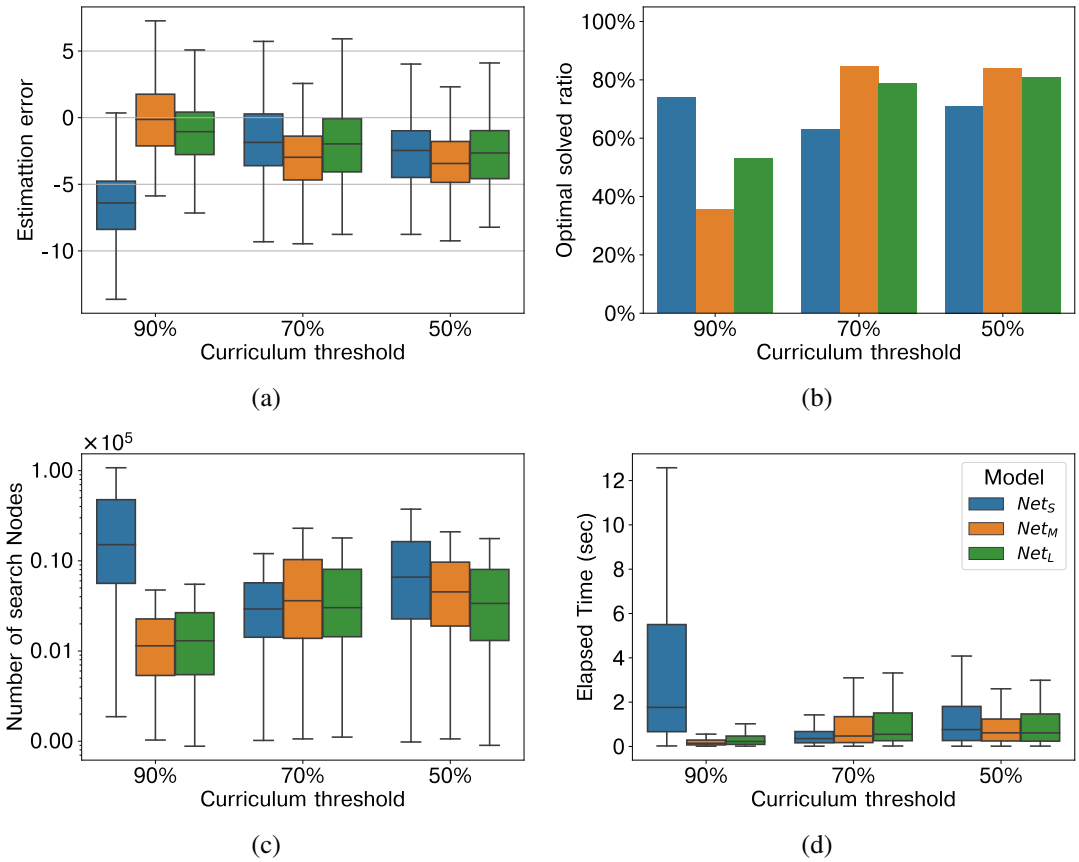


Figure 4.16: Performance comparison on the curriculum threshold parameter; (a) A error distribution between estimated heuristics and shortest path. (b) A rate of heuristics trained with estimated heuristics and shortest path. (c) A number of nodes when A* search proceeds based on each heuristic. (d) A time elapsed on searching A* search based on each heuristic.

converge at the length of the optimal path. As training proceeds in the aforementioned cases, we can confirm that the untrained network can converge to a suitable heuristic by performing Deep A* iteration, normally through the curriculum approach, thanks to heuristics graphs that changes as training progresses.

4.2.3 Comparison

Finally, we investigate the error distribution of the trained networks in 200 samples and the actual correct answers, the ratio of optimal paths to come out when running A*, the number of nodes searching for paths to come out, and the time distribution to come out of paths, and the result is Figure 4.16. First, if we look at the error distribution of expected values and actual answers, if Curriculum thresholds are 90% and 70% and

network size is Net_S, the difficulty of the problem sample used for training is not sufficiently increased, as shown in Figure 4.13 (b) above. Trained from Net_S to 90%, curriculum thresholds tend to be very largely underestimated, and the results in (c) show that the number of nodes generated has the highest distribution, and that the elapsed time distribution is also extremely high. In comparison, cases where Net_M and Net_L were trained with a curriculum threshold of 90% showed a slightly higher expected value distribution compared to other cases being slightly underestimated. Thus, as can be seen in (b), cases, where Net_M and Net_L were trained with a curriculum threshold of 90% estimated the optimal path at a very small rate compared to other cases. However, looking at the results in (c), the number of nodes generated showed the smallest distribution, and the solution time distribution in (d) was also extremely small. Considering these points as a whole, a high curriculum threshold has the advantage of less training time, but it is too underestimated or overestimated by training only low difficulty compared to the actual difficulty distribution of the game. We can confirm that this is difficult to become a heuristic to find the optimal path quickly that we actually want. However, a too-low curriculum threshold can significantly increase training time due to the too-quick difficulty increase, and it can be seen that the training time of Net_M and Net_L using GPUs increased by about 4 hours while the curriculum threshold was reduced from 70% to 50%. However, it was found that there was no significant change in the ratio at which the optimal path came out, compared to the increased training time width or in the distribution of the number of nodes generated in the search. This means that the curriculum threshold value must be appropriately reduced, ensuring that the network is well trained and that the training time is not increased too much. This is because the complexity of environment is too high, so accuracy above certain point is not possible, because a too-high curriculum threshold causes a reaction that difficulty cannot increase, stop still. Therefore, for proper convergence of training time and network, it is necessary to find the necessary curriculum threshold values according to the difficulty of the environment and the size of the network.

In conclusion, in this experiment, if you want fast path-finding performance and want to find the optimal path as often as possible, a curriculum threshold of 70% or less with a generally high optimal path ratio and sufficiently acceptable path computation time may be recommended. However, it is difficult to say that a value is exactly the best, because the optimal path ratio of 70% and 50% of the curriculum threshold does not change that much, but the number of nodes required for search tends to increase

as the curriculum threshold decreases. On the other hand, the 90% curriculum threshold value showed side effects for each model size, because it was difficult to obtain heuristic data evenly for the entire environment with very small training time. In the smallest network Net_S, only very small heuristics were trained, resulting in very large underestimation, which required a relatively large number of nodes to solve the puzzle; conversely, Net_M and Net_L failed to collect and converge real heuristics, resulting in very low rates for optimal path-finding ratio.

4.3 Summary of Chapter 4

In this chapter, we aimed to self-train a heuristic without human knowledge through enhanced training without using conventional heuristics to train a network. The data collection method described in the previous chapter utilised A* search, which used a conventional heuristic. However, traditional heuristics utilising A* to solve a harder sample (which needs a minimum of more than 60 slides) requires substantially longer time and larger memory. Thus, a neural heuristic can be used to solve harder samples faster, as explained in the last chapter.

By utilising this technique, we gathered data to train the network through a neural heuristic to present the Deep A* Iteration (DAI) algorithm using an enhanced training method to converge neural heuristics. However, because of the conditions for admissible heuristics, heuristics comprising networks are not guaranteed to produce an optimal path. This fails to lead to a possibility that data close to the ideal heuristic, which is the data of the optimal path that we aimed for, can be generated through the overestimated data by the overestimated network heuristic. We thus presented the theoretical DAI algorithm that converges to the ideal heuristic and provided experimental evidence to validate this theory. The DAI algorithm can confirm whether a network is overestimated or underestimated compared to the ideal heuristic as well as demonstrate whether the results will converge to the ideal heuristic value.

Another challenge in the DAI algorithm is that if the neural heuristic is in a state wherein nothing has been trained, it takes a much longer time and more memory to solve harder samples. The efficiency of A* search is defined by how well the heuristic predicts the real cost to go, so it takes far longer to solve harder samples and generate data in the untrained network. To overcome this, we suggested a curriculum approach to increase sample complexity in situations where the neural heuristic has

trained enough generated samples, starting from the easiest. Our approach showed that the time taken to generate a training sample was considerably reduced to the time taken to generate specific complexity of the samples when the curriculum was applied versus when it was not. Finally, we confirmed that the neural heuristic after training could generate data and train from very complex samples.

In this chapter, we proposed the DAI Algorithm and then verified its convergence to the ideal heuristic sample. Next, we proposed a method by applying the curriculum method approach. We suggested a training method to generate samples of the overall range, which does not take a long time to generate without any kind of training.

Chapter 5

Deep A* Iteration with self-organizing local view

In the previous chapter, we have shown that Deep A* Iteration converges to the optimal solution, and to solve the problem of the global case where the target point is all random, it takes nearly infinite time to collect training data. By sampling a relatively short-depth local case problem, we were able to considerably reduce the expected training time to train a global problem in a curriculum manner that gradually increased its range.

However, this method manually changes the local range for sampling by specific rules. Data collecting time increases when the local range that the current network tries to train is set too large, whereas too small a value can lead to hardness on gathering data for global problems, which is problematic because it is difficult to determine how to increase training time and local performance sampling range.

Therefore, in this chapter, we collect samples of steps as difficult as the network is well-structured, constructing appropriate sampling ranges on our own and then retraining them to progressively collect training data, eventually extending them to a global range. We also propose a way to increase sample efficiency by increasing the number of trained data samples, which were very small compared to the number of heuristics evaluated for data collection.

5.1 Method

When solving the 15 puzzle problem, we note that the collection of data from existing methods does not occur eventually if the problem sample fails to solve the A* algorithm. We limit the number of shuffles because when the number of shuffles increases to the point where it becomes difficult to solve, the A* search cannot finish owing to insufficient memory, and the time to collect data needed for training increases exponentially. This was essentially because with the heuristics we now have, tens of thousands of branches have to be created to find the only path we will use for training. Furthermore, the more precise the heuristic is, the fewer the nodes needed for A* search to get results, but nevertheless, in the very well-trained heuristic from the previous method, the number of nodes evaluated and the number of data generated is close to a ratio of 500:1. We need a way to generate meaningful data, even if A* does not solve the problem provided, to solve the fundamental problem in which only the final path is used in these A* algorithms. Therefore, we aim to design an algorithm that increases the efficiency of data sampling using branches that have not been used in the previous study and self-optimizes the distribution space with f. This is the combined value of the distance between the represented start nodes, g, and the expected distance between the individual nodes and the target nodes, h.

5.1.1 Expending Nodes and Full Path

We tried to modify the equation of the ideal heuristic sample after obtaining a single long path to proceed with the training of the Deep A* Iteration. As mentioned above, generating data after the final route is calculated with all searches having been completed takes an excessive amount of time. Consequently, we designed a method that allows a portion of the ideal heuristic sample to be filled out each time whenever the node included in the path is expanded.

$$R_k = \{n_0, n_1, n_2 ..., n_{k-1} | \text{ all root node from } n_k\} \tag{5.1}$$

$$X_k = X_{k-1} + \bigcup_{n_i \in R_k} (n_n, n_i) \tag{5.2}$$

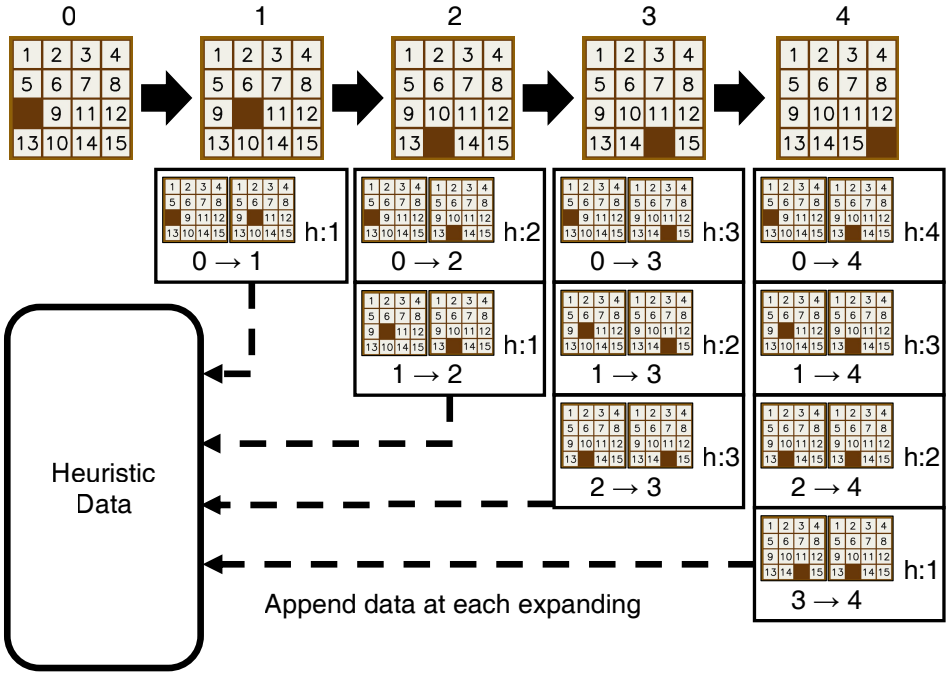


Figure 5.1: Collection of data from every node expansion

$$Y_k = Y_{k-1} + \bigcup_{n_i \in R_k} (n_n.g - n_i.g) \tag{5.3}$$

As the search for the best-expected path for data generation deepens, as given in Equation 5.1, the newly obtained nodes can obtain all the root nodes from which they are derived, and the set of these nodes is called R_k. Using the relationship between the obtained Rk and the newly obtained node, we can extend the ideal heuristic sample as presented in Equations 5.2 and 5.3 and Figure 5.1.

X_k, defined in Equation 5.2, is the set of root nodes that the newly obtained node nk has, which is called Rk and is a source, target state pair with elements. Y_k, defined in Equation 5.3, is the set of root nodes that the newly obtained node nk has, which is called Rk and the distance from the elements. They can be easily obtained through the difference in g values, the distance between the nodes, and the start nodes accumulated.

The collected X_k and Y_k are subsets of the ideal heuristic sample used by Deep A* Iteration obtained in the final path, and when k equals the length of the final path, Xk and Yk become the same as the ideal heuristic sample. Additionally, X_k and Y_k are subsets of the ideal heuristic sample used in Deep A* Iteration, thus maintaining the optimal convergence property as well. This allows us to obtain a data sample that is

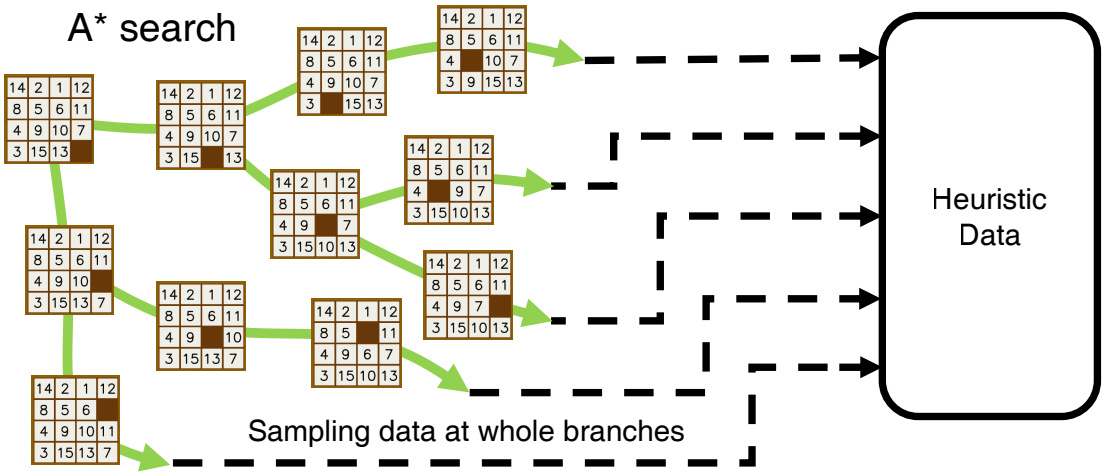


Figure 5.2: Overview of the whole-branch sampling. All branches calculated by minimal f searched while using A* are optimal paths and can be sampled as a heuristic dataset.

meaningful for training even if A* is not fully solved.

5.1.2 Whole-Branch Sampling

Using the way presented above, we were able to obtain data in the process of nodes expanding one by one, but in order to apply the above method, we need to select only nodes in the optimal path.

When we select only the nodes of the optimal path we want, we may eventually get some of the ideal heuristic samples before the end of the A* search, but we cannot solve the problem that only one branch is used and others are discarded in the process of generating data, which is a fundamental problem. Therefore, we present the whole-branch sampling method as a way to include other branches in the dataset and configure them using the same ideal heuristic sample as before.

The A* search finds the optimal path by sequentially checking the sample with the smallest f value, i.e., the sum of the calculated heuristic and the actual distance between the current node and the starting node (when the heuristic is allowed). It should be noted that A* search is one of the specialized versions of Dijkstra's algorithm that finds the optimal path between all states from the initial state. Therefore, the node that has the minimum f value chosen in the A* search has an optimal path towards the node itself, not the optimal way for solving the problem presented by us. To be more precise, A* has a tendency to extend to a desired goal to solve a desired problem because of

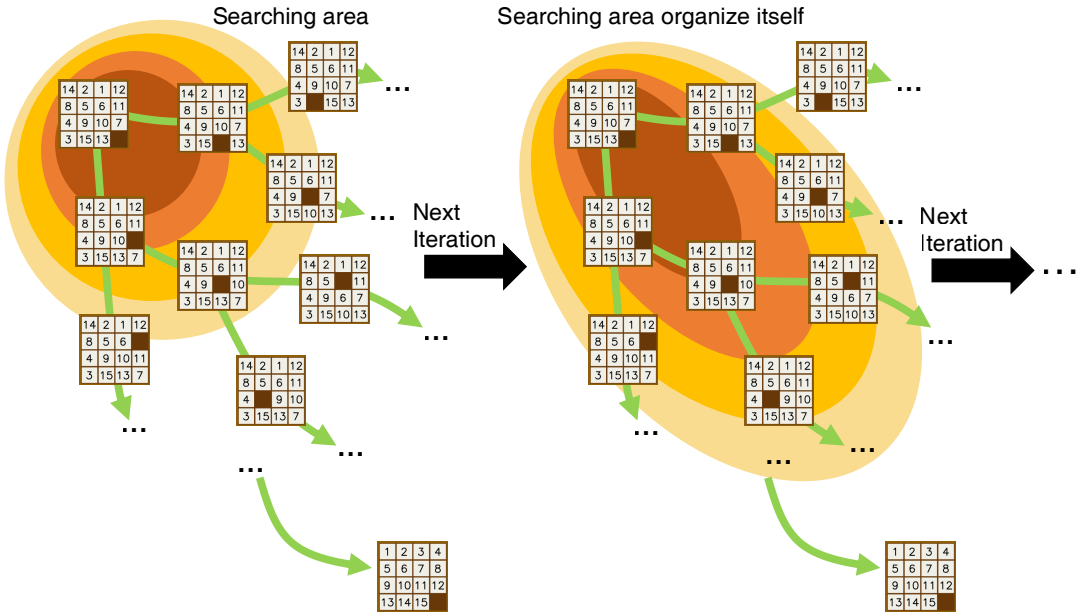


Figure 5.3: Search area becomes deep owing to the heuristics becoming more accurate as training proceeds.

heuristics, but in reality, a series of processes continues creating nodes till the optimal route for target comes by.

The ideal heuristic sample we defined in the previous chapter consequently generates the optimal path to the problem we present as data, but this is not the only result that is available to use. As mentioned in the earlier section, by combining this with the method of collecting data each time the proposed node is created, a dataset can be collected from any branch that is expanding within the A* search.

This allows us to generate a dataset tens of thousands of times, including data from previously calculated ideal heuristic samples from the results of A*, as illustrated in Figure 5.2. In addition, it is possible to generate wide and diverse data rather than only one directional data in a single search process, and the number of data that can be generated in a single search can be increased exponentially.

5.1.3 Organizing Searching Area

A method of collecting data through branches generated in A* search has been presented in the previous section. In this section, we now present a method to proceed with data sampling that extends to a global range even in an untrained network through the presented method. In the above, a method of collecting data through branches gen-

erated in A* search has been presented. The range of A* search, i.e., the space, can be represented as f, a contour based on combination of h and g, a distance between starting node and individual node, and the same thing that heuristic predicted. Here, a space made with such f values is defined as the search area. As mentioned above, when data is collected through branches generated in the A* search, the distribution of data generated is sampled sequentially along the search area, which is the distribution of branches generated in the A* search.

An A* search with a heuristic in which nothing has been trained makes the search similar to an area-first search because the heuristic cannot accurately identify the direction of the search. Solving a problem in a very large state space using a width-first search is excessively time-consuming, but using the above approach, we can collect datasets with information during the search and organize the network more precisely. By continuing training, the heuristic, which has become more precise, can present the direction of search, and the search area searched by A* search spreads in an oval shape in the direction of the target node compared to the previous one. As the search area spreads in an oval shape, it allows data to be collected up to nodes at a longer distance, and the data between nodes at a longer distance come closer to the global sampling of the data, which means the trained network converges into a more precise heuristic. Finally, when this training is repeated, the trained heuristic forms a search area in a straight line that connects the start and the target node. In short, data is generated locally according to the degree to which the network is trained, whether we are sampling globally without setting the nodes in a local range. As training is repeated, the generated data reaches the target node, that is, close to global sampling.

5.1.4 Search Node Limitation

As described above, the search area naturally spikes and generates more global data when training the branches that are generated by A* search, by repeating the A* search again. However, searching along the search area using an A* search with inefficient heuristics has the limitation that the generated dataset does not reflect the searching area towing to the immediately trained heuristics, because we must wait until there are branches having a strong searching tendency to exit the queue.

Heuristic training proceeds, but the update delay becomes very large as the distribution of generated data is trained. The delay arising from this data distribution is a repetition of training and data generation within advanced ranges, which makes the training cycle

very slow.

Consequently, in order to ensure that the distribution of the data is updated according to the trained heuristic, when the number of nodes exiting the queue in the A* search exceeds a certain number, we quit the current search and perform a new and more efficient A* search. The data generated by this method is periodically generated in the search area based on the latest and most efficient heuristic. As training progresses, performance increases rapidly.

5.1.5 High-Depth Sampling

In the ideal heuristic sample, the number of data in short-depth samples is directly proportional to the length of the path. The node sampling method being identical to the ideal heuristic sample, short-depth samples are generated at a high rate. In particular, while traditional ideal heuristic samples only select the path that corresponds to the highest difficulty in A* searches, the node sampling method uses all the searched paths, even the short-depth paths used for data generation, increasing the proportion of short-depth samples. This increases the likelihood that the network will focus on short-depth samples that have already been trained, rather than longer-depth ones.

To overcome this problem, we present the high-depth sampling method, which increases the number of the most challenging samples (high-depth samples) in the training data and reduces the ratio of the short-depth samples. We then only collect the data from the samples of nodes that are deeper than the multiples of the percentage sample ratio γ at the deepest search size D. If the sample rate variable γ is 1, the number of data generated in the course of A* once is equal to that of the final path only, as described in the previous chapter. If 0, the data generated in the course of A* once is designed to be the same as sampling the data in all paths generated in A*. This allows the ratio to be adjusted based on how reliable the data is to calculate the number of data generated at a time of the A* search.

5.1.6 Weighted A* Search

$$f(c,t) = g(c) + h(c,t) \tag{5.4}$$

In order to increase the proportion of deep searches among the searches carried out, we use weighed A* search to collect trained data.

When the variable c is defined as the current node, t is defined as the target node, the function g is a heuristic function that predicts the distance that the current node extends from the start node, and h is the distance between the current node and the target node. In the existing A* search, a function f that determines the priority of an extended search node is defined as in Equation 5.4. At this time, given that the node with the smallest f function value is expanded preferentially, the greediness of the A* search is determined by the degree to which g increases and h decreases each time the node expands.

This is because the h value decreases larger than the degree to which g increases unconditionally because the heuristic mentioned above cannot guarantee the optimal path, so using the overridden heuristic makes a greener search beyond the standard to find the optimal path. Therefore, the ratio of this increase in the g function and the decrease in the h function acts as an important factor in the efficiency of the search to find the path to the goal and whether the path obtained by the search is the optimal path.

$$f(c,t) = \omega \cdot g(c) + h(c,t) \tag{5.5}$$

However, weighted A* techniques promote greedy search by artificially adjusting the ratio of g to h in the calculation of f, as given in Equation 5.5. At this time, the closer the variable ω is to 0, the more the greedy search is driven; the larger the value ω, the less greedy the search is; if ω is 1, the search is the same as a regular A*. It is assumed that the heuristic function is constructed very linearly so as to be proportional to the actual distance from the target node. At ω values less than 1, deep greedy searches are performed at very high speeds to find the path to the target node. We also confirmed that the trained neural heuristics from experiments in previous chapters predict some linear ideal heuristics. It can be seen that a deep greedy search can be made with less searching.

This deep greedy search increases the ratio of the longer path and can help to train evenly over a wide range of heuristic values. In fact, as can be seen in Figure 5.4, when weighted A* is applied, the distribution of nodes searched in A* search is evenly distributed over a wide range. Furthermore, even after training is done, adjusting the ω value as you proceed with the search can help you find a greedy and fast path, or reduce greedy search to increase the probability of obtaining an optimal path.

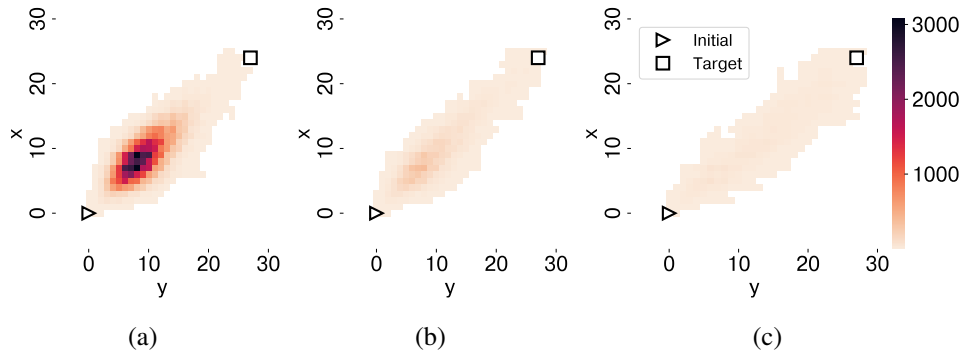


Figure 5.4: Distribution of searched nodes varies according to the ω value; Initial state and Manhattan distance of X,Y axis. (a) ω: 1.0, (b) ω: 0.9, (c) ω: 0.8

5.1.7 Algorithm

Algorithm 2 DAI-WBS

Input: B: Batch size, I: Iterations, M: Maximum search, γ: Near max depth ratio, ω: Weighted A* weight

initialize network parameter θ
Set $i \leftarrow 1$
Set Empty X, Y
while $i \leq I$ **do**
  Random sample s, t
  Initialize D as 0
  for m=1 to M **do**
    $c \leftarrow$ get minimum f node from that not visited samples
    $D \leftarrow \max(D, \text{c.g})$
    if $c.g \geq \gamma \cdot D$ **then**
      $R \leftarrow \{n_0, n_1, n_2 ..., n_{n-1} |$ all root from $c\}$
      $X \leftarrow X + \bigcup_{n_i \in R}(c, n_i)$
      $Y \leftarrow Y + \bigcup_{n_i \in R}(c.g - n_i.g)$
    end if
    if $\text{len}(X) \geq B$ **then**
      $\theta \leftarrow \text{train}(\theta, X, Y)$
      Update A* heuristic as $H(\theta)$
      Clean training data X, Y
      $i \leftarrow i + 1$
    end if
    if $c == t$ **then**
      break
    end if
    Add neighbors to not visited samples, and calculate f with ω
  end for
end while
Return: θ

Algorithm 2, i.e., Deep A* Iteration with Whole-Branch Sampling(DAI-WBS), is designed by combining the comprehensive methods presented above. As data must be collected in the course of the A* algorithm, the training process works by repeating A* searches until the number of iterations i is higher than I and updating the network whenever the collected data exceeds the training batch size B. The algorithm continues to run A* searches with a completely random start and target point by default. An algorithm with high-depth sampling was added to collect and train data when samples that satisfy the conditions under minimum f value, D, and γ are presented.

Finally, we designed algorithms that can enable the neural network to solve complex cases by itself by collecting samples for real cases over the network and re-training the network through samples.

5.2 Experimental results

Finally, we investigate the results by adjusting the hyperparameters of the methods presented above. This is to see what values produce optimal results, how they change training time, and how much performance has been improved. This experiment has been conducted for training, using an AMD Ryzen-9 5950X 16-Core Processor and two Nvidia RTX3080, and four multi-processes have been used to accelerate training data generation. For swift experiments, the networks used for training and experiments were Net_S, Net_M, and Net_L, which are described in the previous chapter's Table 4.1.

Furthermore, devices running A* heuristics were adapted to operate most efficiently, based on the network size of the model when the data was generated for training data generation and performance measurements. CPU was used for Net_S, while GPU was used for Net_M and Net_L. This experiment used 200 random samples and optimal paths collected to measure performance in the previous chapter, to verify how efficiently the proposed network heuristic finds the optimal solution while measuring other performance indicators.

5.2.1 Organizing Sampling Area with Whole-Branch Sampling

In order to show that whole-branch sampling automatically collects higher-difficulty samples during training, we collect 10000 nodes generated from A* search using the trained neural heuristic for each training iteration. The parameters used in the exper-

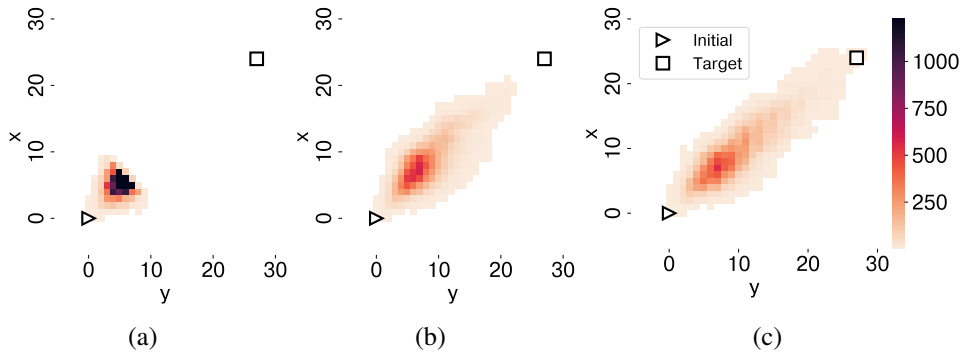


Figure 5.5: Example of self-organizing; Initial state and Manhattan distance of X,Y axis to solve the same problem according to the number of training iterations of the network, the distribution and density of 10000 nodes collected from A*. Color depicts the number of nodes that have a corresponding node. (a) 0 iteration, (b) 10 iteration, (c) 80 iteration.

iment were NetS for model size, 10^5 for search node limitation, 0.8 for γ, and intermediate results of training without Weighted A*. The distribution of the nodes created by separating and measuring the x and y of the Manhattan distance of the nodes was generated as a heatmap, shown in Figure 5.5.

In Figure 5.5, the triangle point in the heatmap is the initial state in which A* search is executed, and the square point is the target state. First, in Figure 5.5 (a), showing 0 iterations, nodes with relatively small distances were created based on the Manhattan distance. This seems to be because the 0th iteration is almost completely a breadth-first search because nothing is trained. After training the data collected with this distribution at the 0th iteration and then continuously training up to 10 iterations, the A* search has a heuristic index to reach the target state, so a deeper and more efficient search is made.

In Figure 5.5 (b), we can see the distribution of search nodes at this time, and you can see that the distribution of search nodes approaches the target state. Thus, by training the deep search data that has been trained in this way, the neural heuristic becomes more accurate. In Figure 5.5 (c), the distribution of search nodes finally reaches the target state. It takes approximately 5 minutes to reach this training process of 80 iterations.

These experimental results allow us to collect samples of biased elliptical data directed toward the goal as accurately as heuristics by using whole-branch sampling techniques. As heuristics become accurate, we show that the distribution of collected data also

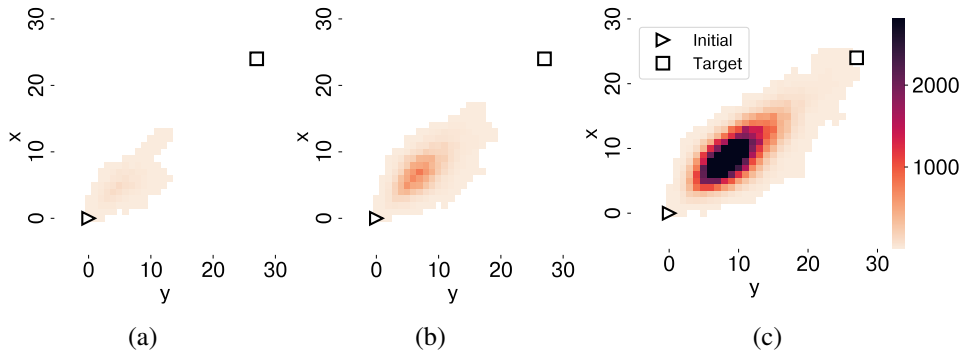


Figure 5.6: Effect of search node limitation; A* search node distribution by maximum search node limit. The maximum number of search nodes in each 40-iteration trained heuristic was tested as follows. (a) 10^3 node, (b) 10^4 node, (c) 10^5 node

grows, and furthermore, we can generate a virtuous cycle structure that makes the heuristics more accurate. These results imply that sufficiently efficient heuristic training is possible by iterating to collect and train data from attempts to solve random samples without any guidance in the end.

5.2.2 Search Node Limitation

Subsequently, we investigate the efficiency of training according to the size of the maximum number of expansion nodes to proceed with A*search in one sample when collecting data. As shown in Figure 5.6, where the maximum number of search nodes in A* in trained networks is increased, samples of longer distances can be collected as the maximum number of search nodes in A* increases. However, as the number of samples searched and trained increases, it takes longer for the trained network to be reflected in data generation. The longer it takes for training to be reflected in data generation, the longer the network settles into an ideal heuristic; therefore, we need to properly adjust the maximum number of search nodes with two goals: short network training settlement time and long distance sample collection.

5.2.3 High-Depth Sampling

As shown in Figure 5.6, samples of longer distances can be collected at higher frequencies than samples of longer distances as the maximum number of search nodes in A* increases. The data we need for training requires a large data distribution of nodes with a long distance from the starting node and the target node to effectively train the overall heuristic of the environment. As most nodes that are searched are focused on

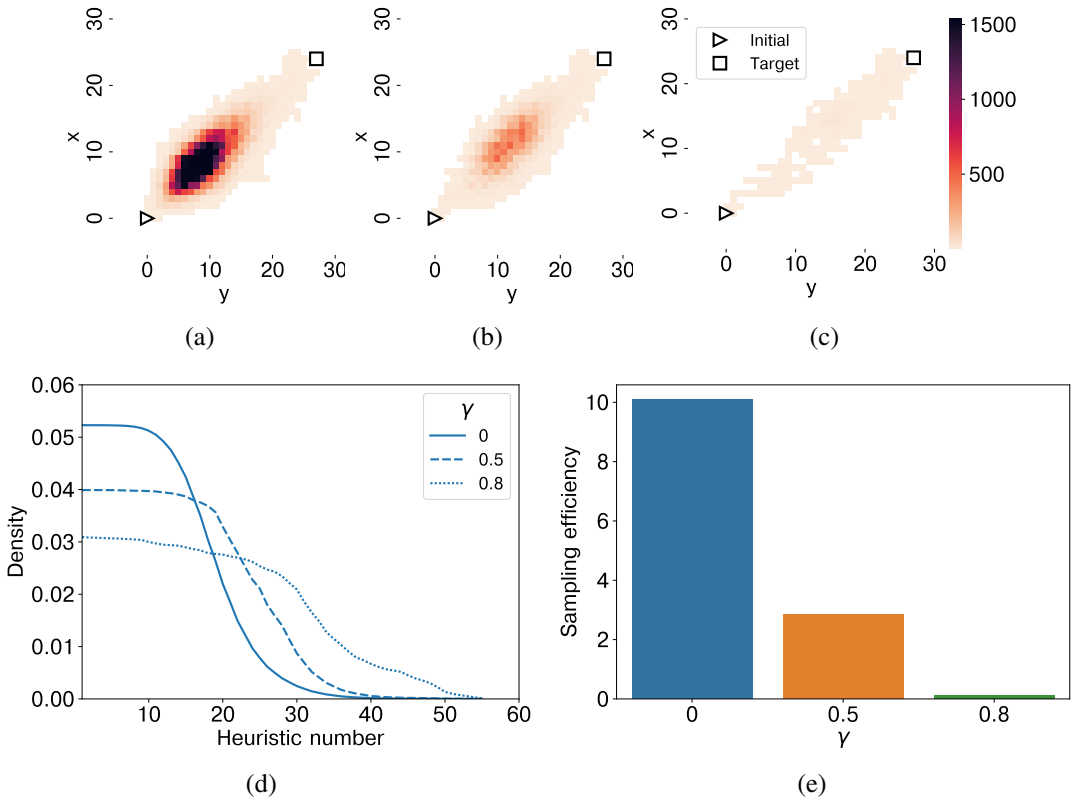


Figure 5.7: Effect of high-depth sampling in train data collection. (top) Distribution of nodes that are collected as data according to the γ value. (a) γ: 0, (b) γ: 0.5, (c) γ: 0.8, (bottom) Actual training data density and sample efficiency. (d) Heuristic data density, (e) Sample efficiency (number of data collected / evaluated nodes)

nearby nodes rather than distant nodes, a method is needed to collect some safe and long-distance samples intensively. To achieve this, we present high-depth sampling, where we can adjust the number of missing data in training according to the γ value. However, an in-depth analysis is needed on how the distribution changes in datasets that actually change according to high-depth sampling and what γ values should be selected for effective training.

Theoretically, the high-depth sampling we present does not collect branches below γ, which is a certain proportion of the length of branches that have been extended to the maximum before. As the theoretically expected sampling efficiency, it can be expected that having a γ value of 0 will have a very high sampling efficiency because of pure whole-branch sampling, and that the γ value of 1 will have a sampling efficiency similar to that of collecting only the results of A*. Similarly, the theoretically expected distribution of sampled data can be expected to be overwhelmingly large; with a value of γ of 0, less difficulty than the sample provided using purely whole-branch sam-

pling, whereas a value of γ of 1 would result in larger samples collected from relatively higher-length branches.

We investigated the total number of nodes collected as data and the distribution of nodes collected within a limited number of searches 10^5 according to γ values in one trained network heuristic to determine how much the high-depth sampling method reduces sampling efficiency and changes the distribution of the collected data. In this case, the number of nodes evaluated heuristic for the search is 1.8×10^5.

The result of Figure 5.7 is. If the γ value is 0, then high-depth sampling is the same as the unapplied state and samples a very large number of nodes, but the distribution of most sampled nodes, such as Figure 5.7 (a), shows that nodes are sampled at the start state rather than at the maximum length of a given sample. In Figure 5.7 (b), the γ value is 0.5, and the distribution of nodes being sampled close to the start state is reduced. In Figure 5.7 (c), the γ value was set to 0.8, and the distribution of the sampled nodes was shown to be an overall even distribution, unlike before. However, as these set γ values increased, the number of nodes sampled decreased from 1.0×10^5 when the γ value was 0, to 2.0×10^4, and 7.0×10^2.

These results show that high-depth sampling allows us to filter out excessively generated, near-distance branches from the start state, and conversely, some sampling efficiency can be reduced as a side effect.

Subsequently, we investigate the distribution and efficiency of heuristic data that is actually collected rather than sampled nodes. Our whole-branch sampling method collects heuristics for sampled nodes and all root nodes of sampled nodes as data. Therefore, sampling from a branch with a length of N collects data with heuristic values of $1, 2, 3, ..., N$ for numbers below N. This leads to the problem that samples with low heuristic values increase exponentially, and samples with relatively high heuristic values decrease. In fact, the factors that influence our training are heuristic data collected in this manner, so it is necessary to investigate the number and distribution of heuristic data collected that ultimately vary as the γ value varies.

In Figure 5.7 (d), γ values of 0 show that heuristic samples of less than the length of the branch are collected in addition to the characteristics sampled at nodes close to the starting point. When the γ value becomes 0.5, the phenomenon of bias toward the low heuristic value is alleviated because the distribution of the sampled nodes has become distant from the starting node. Finally, when the γ value was set to 0.9, the distribution

of sampled nodes was much closer to the target node than before, indicating a higher proportion of high heuristic values. As these set γ values increased, the number of heuristic data sampled decreased from 1.9×10^6 when the γ value was 0 to 5.3×10^5 and then to 2.4×10^4. Finally, in Figure 5.7 (e), as the number of nodes evaluated heuristically during the search is 1.8×10^5, the sampling efficiency (number of collected data / evaluated nodes) is 10 when γ is 0, 2.84 when 0.5, and 0.12 when 0.8.

Through this experiment, we can collect a considerable amount of heuristic data if we just use pure whole-branch sampling, but we can clearly confirm that most are made up of samples with low heuristic values. Furthermore, we confirm that by properly filtering the data using our proposed high-depth sampling technique, the proportion of data with high heuristic values can be increased. Data with low heuristic values can be very easily approximated, but as data with high heuristic values are difficult to easily approximate, lowering the weight of low heuristic data and increasing the weight of high heuristic data can be crucial to approximate the heuristic function well over a wide range.

5.2.4 Weighted A*

However, when the data are filtered to increase this high heuristic ratio, the sampling efficiency is significantly reduced. This is because, even if high-depth sampling is performed, the A* search itself is not well performed by deep greedy search, and thus it searches for nodes in a range close to the start node. To mitigate this issue, we apply Weighted A* search techniques to A* search collecting training data, thereby reducing the number of searches itself to solve the sample by promoting a greedy search and seeking to quickly collect data and converge training. Weighted A* claims that the reduction in sampling efficiency due to high-depth sampling can be reduced by operating to reduce the number of branches of short paths, as it increases the greedy search, resulting in many branches of long paths.

In order to analyze the benefits of using weighted A* in practice, it is necessary to investigate how the distribution of collected data varies and how the sample efficiency varies. Therefore, we investigate the distribution and sampling efficiency of heuristic data collected by applying Weighted A* to the data collection process in the following experiments.

First, we investigate how the distribution of nodes collected as data varies when Weighted

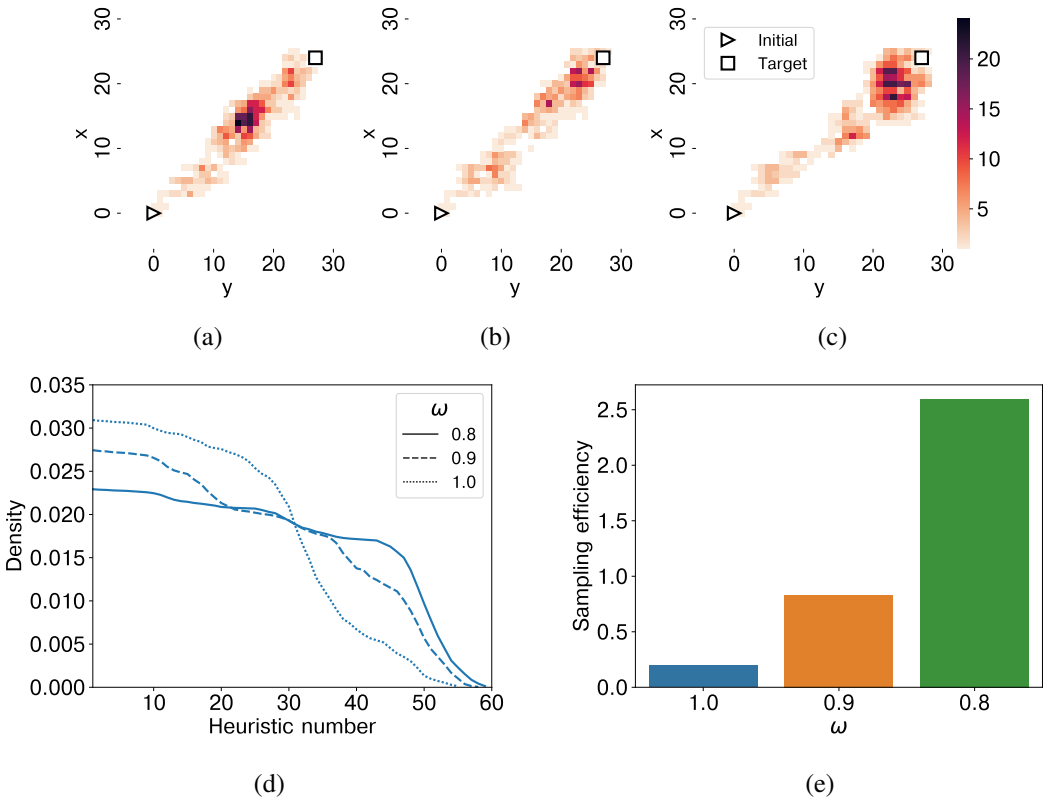


Figure 5.8: Effect of weighted A* in training data collection; (top) Distribution of nodes that are collected as data according to the ω value. (a) ω: 1.0, (b) ω: 0.9, (c) ω: 0.8, (bottom) Actual training data density and sample efficiency. (d) Heuristic data density, (e) Sample efficiency (number of data collected / evaluated nodes

A* is applied. The experiment was conducted in three ways, from γ values of each fixed 0.8, ω values of 1.0 to 0.9, 0.8, and the result is Figure 5.8.

The results of these experiments are as follows. When the ω value is 1.0, the distribution of sampled nodes, as shown in Figure 5.8 (a), is shown to be concentrated in the middle, as the weighted A* is not applied. Figure 5.8 (b) is a case where the ω value is slightly reduced to 0.9, and we see that the distribution of the nodes being sampled has shifted a little closer to the target state and farther from the start state. Finally, in Figure 5.8 (c), we see that most of the node distributions are distributed close to the target state, with ω values decreasing even further to 0.8. These results clearly show that weighted A* techniques are advantageous for collecting samples of relatively long distances as data. Furthermore, after adjusting ω values, the number of nodes sampled changed only slightly to 716, 575, and 785 in the order of 1.0, 0.9, and 0.8, whereas the number of heuristics evaluated during the search decreased significantly to 1.7×10^5, 2.5×10^4, and 9×10^3.

The results in Figure 5.8 (d) show that the heuristic value distribution of the generated data increases as the ω value decreases to 0.9 and then 0.8. In addition, in Figure 5.8 (e), the sampling efficiency (number of data collected / evaluated nodes) was 0.12 when ω was 1.0, 0.83 when 0.9, and 2.5 when 0.8.

These results show that applying weighted A* in the data collection process can increase sampling efficiency while helping to distribute the heuristic size of the collected data highly.

5.2.5 Convergence and Performance

The above experimental results confirmed the effectiveness of our proposed parameters and theoretically showed that our algorithm can train heuristics. In the following experiments, we confirm that the algorithm presented in this chapter, the DAI-WBS algorithm converges with the ideal heuristic as in the previous chapter. In addition, we confirmed whether the training time differs from the results of actually training the network according to the identified effects. At last, we confirmed that the previous curriculum learning method and the heuristic plot were compared to show similar patterns in practice. The performance of the trained network is evaluated in performed on 200 random samples that already know the optimal path.

Figure 5.9 investigated whether the DAI-WBS algorithm converges to ideal heuristics like DAI in the previous chapter. The experiment was conducted by preparing an overestimated, underestimated network, training it from DAI-WBS, and investigating the distribution of differences between the actual heuristic values of 200 samples already obtained and those predicted by the trained network for each iteration. The parameters of DAI-WBS used in training are search node limitation: 10^5, $\gamma : 0.8$, and $\omega : 1.0$. The network used in the experiment is Net_S.

The results in Figure 5.9 (a) show that the overestimated network converges close to the ideal heuristic value. The results in Figure 5.9 (b) similarly show that the underestimated network converges close to the ideal heuristic value, which is the same as the DAI algorithm in the previous chapter. Figure 5.9 (c) then shows that as the overestimated network converges to the ideal heuristic, the rate of solving given samples with optimal paths increases when performing A*. Conversely, Figure 5.9 (d) shows that the number of nodes searching to solve a given sample when performing A* is reduced as the underestimated network converges to the ideal heuristic.

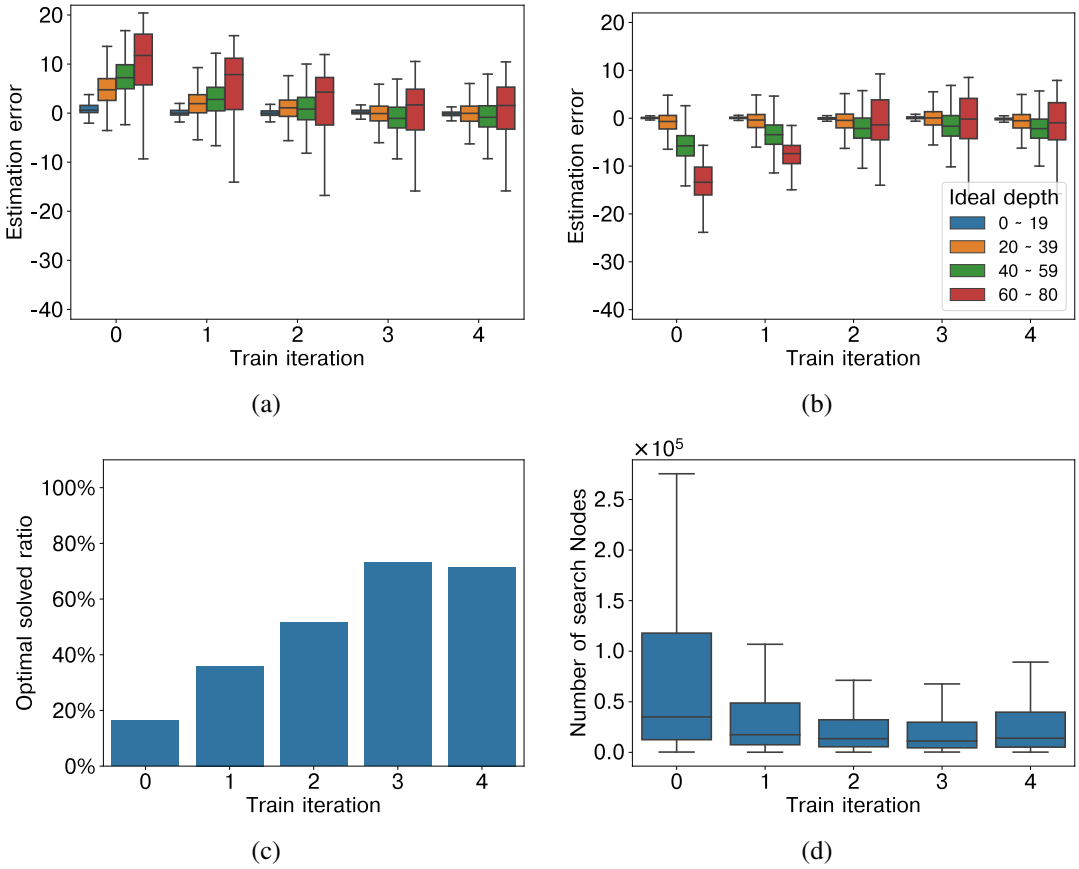


Figure 5.9: Convergence process and effect of neural network; the experiment used Net_S. (a) convergence process of overestimated network, (b) converge process of underestimated network, (c) convergence effect of the overestimated network, (d) convergence effect of the overestimated network

These results clearly show that DAI-WBS algorithms operate fundamentally as DAI algorithms, converging to ideal heuristics.

In Figure 5.10, we compare curriculum learning in previous chapters to train through DAI algorithms in difficult cases with untrained networks and training patterns from untrained networks through DAI-WBS algorithms. The experiments were conducted by training a randomly initialized network to investigate the predicted heuristic of two fixed optimal path samples with a network trained at each iteration. The parameters of DAI-WBS used in training are search node limitation: 10^5, $\gamma : 0.8$, and $\omega : 1.0$. The network used in the experiment is Net_S.

In Figure 5.10 (a) and (c), the method proposed in the previous chapter is the curriculum learning method, which shows that the overall heuristic increases as the duration passes and eventually converges to the ideal heuristic. In Figure 5.10 (b) and (d),

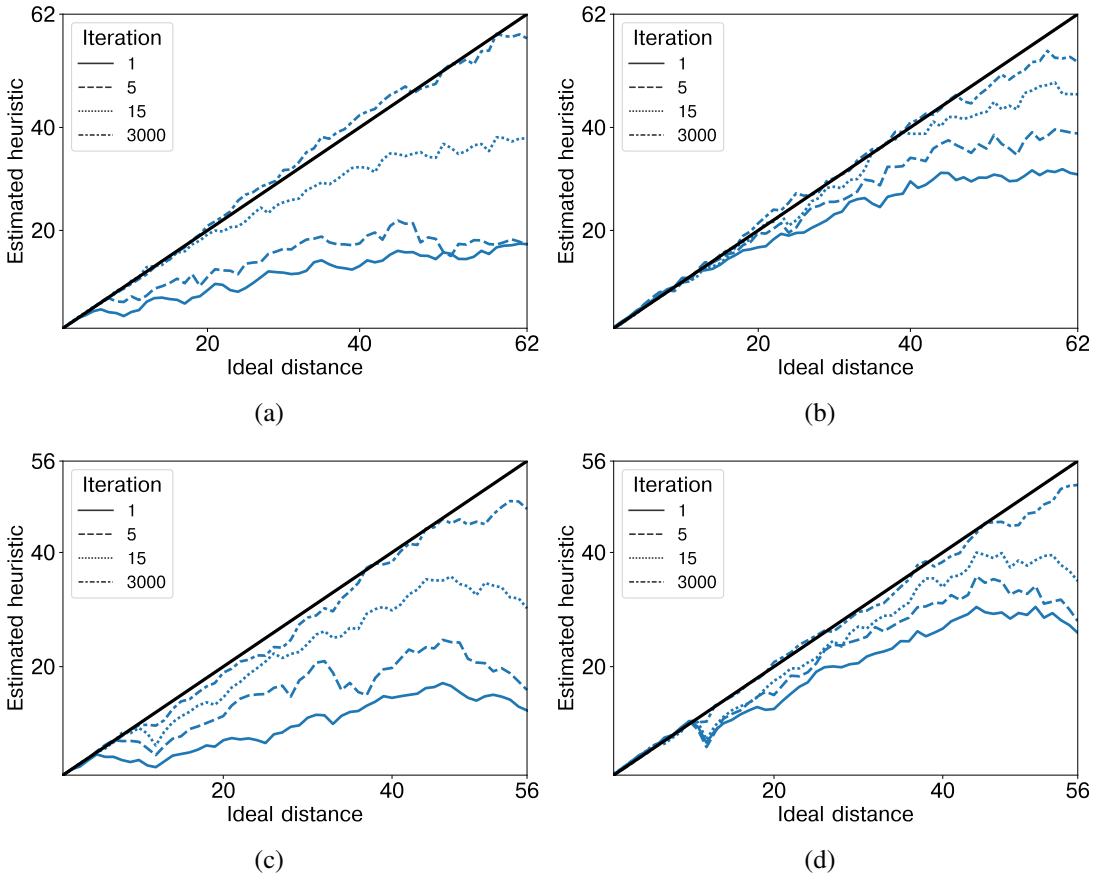


Figure 5.10: Increasing heuristic plot by iteration in two methods; (left) DAI with curriculum learning, (right) DAI-WBS

we see that the proposed DAI-WBS algorithm also increases as the overall heuristic increases over time and eventually converges to an ideal heuristic. Even in curriculum learning, the 1st iteration, i.e., heuristics are not properly predicted when they are trained only once, whereas DAI-WBS algorithms show an almost directly proportional increase in heuristics even when they are trained only once. These results clearly show that the proposed DAI-WBS algorithm allows us to sufficiently approximate heuristics for the entire environment without the curriculum learning method.

Subsequently, we conducted training for each suggested parameter and investigated the training time and the efficiency of the trained network accordingly. However, in these experiments, when the search node limit and the high-depth sampling variable γ are small values, only a small range of heuristic data is gathered, resulting in insufficient training. In Figure 5.11, based on the search node limit values to 10^5 and γ values to 0.8 for showing sufficient results, we investigate training time reduction and heuristic performance change for changes in variables ω. In the experiment, a total of 3000

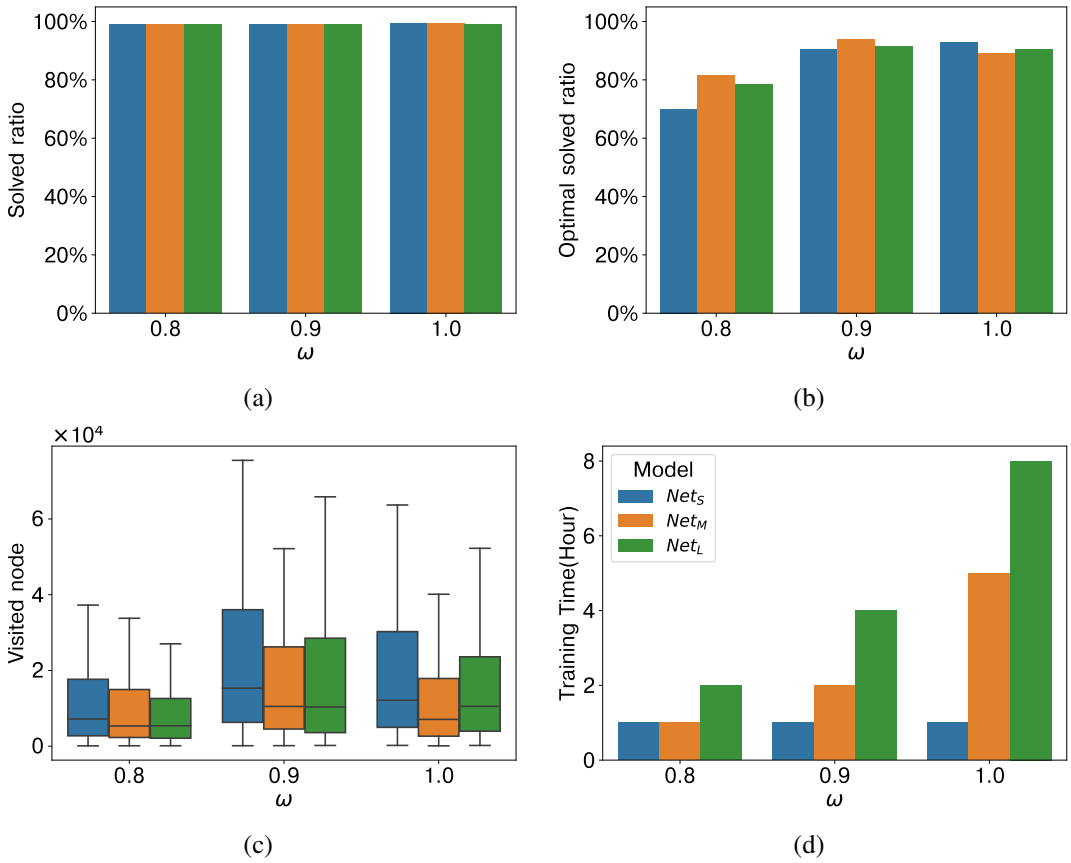


Figure 5.11: Comparison of last training results by ω value; (a) Solved ratio at performed 200 random samples, (b) Optimal solved ratio at performed 200 random samples, (c) Visited node for solving performed 200 random samples, (d) Training time

iterations were trained, and it was confirmed that all networks converged.

Figure 5.11 (a) shows the ratio of 200 random samples prepared to find a path to the target under a search node limitation condition of 10^5 when performing A* search with these three trained network heuristics. In Figure 5.11 (b), the ratio of the path found to whether the path is the optimal path and with a low ω value of 0.8 is relatively small, but it still has an optimal solving ratio beyond 70% in any size network. In contrast, in Figure 5.11 (c), which records the number of nodes visited while performing A*, it can be seen that 0.8 has the least visited nodes and finds the path. Finally, in Figure 5.11 (d), we find that as the variable ω decreases, the time it takes for 3000 iterations to proceed with training is significantly reduced, and even Net_L, which requires more than 13 hours of training when trained using curriculum method, can be trained in 2 hours.

These results show that there is almost no performance reduction while significantly re-

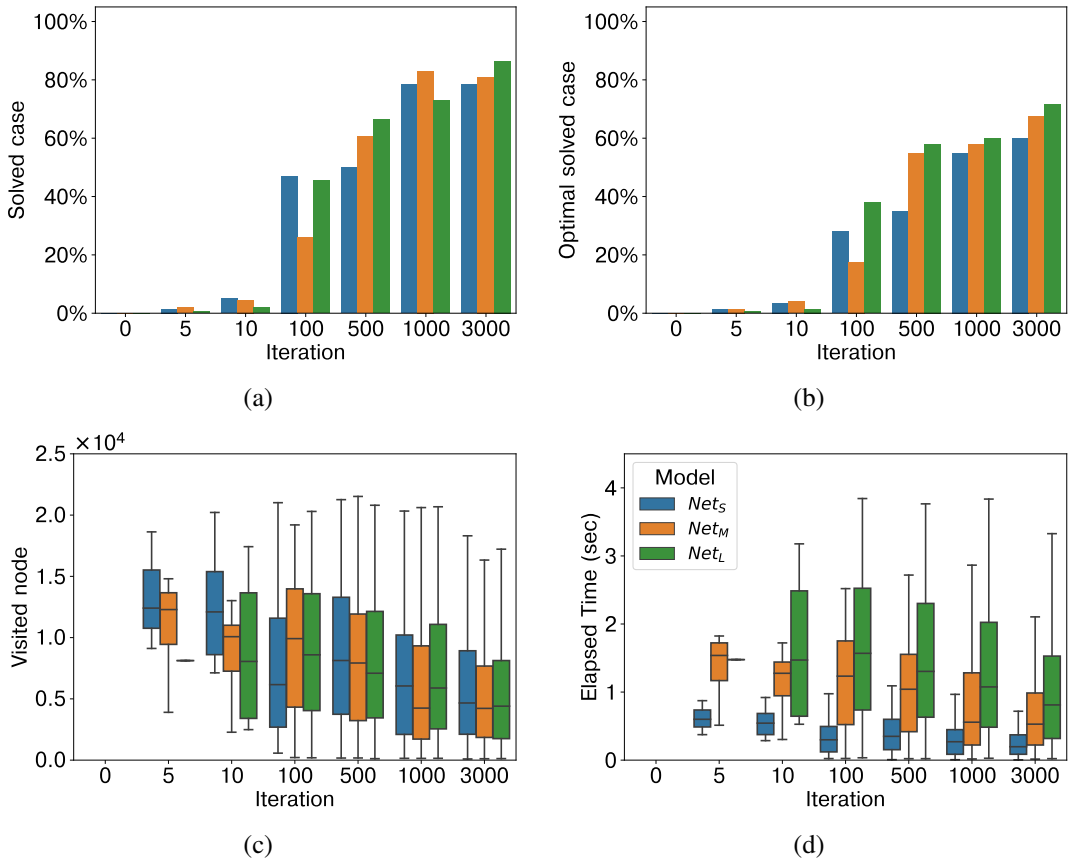


Figure 5.12: Improving the performance of the network according to the iteration; A* search results in search node limitation value 10^4 at performed 200 random samples. (a) Solved ratio, (b) Optimal solved ratio, (c) Visited node for solving, (d) Elapsed time for solving

ducing the time it takes to train through the regulation of the parameter ω for weighted A* as the training proceeds. The experiment also showed a slight decrease in the optimal solving ratio, but it can also be considered an advantage that the time it takes to solve the samples has also decreased. Therefore, we proceeded with the following parameters: the search node limit is 10^5, γ is 0.8, and ω is 0.8 in subsequent performance experiments.

Then, as shown in Figure 5.12, we investigated the performance by increasing iterations to determine how performance increases as training progresses in the DAI-WBS algorithm. Performance was measured while solving 200 samples prepared by A* search with a small search node limitation of 10^4.

Figure 5.12 (a) is a percentage of samples to find the path to the target at a given search node limitation value of 10^4. Figure 5.12 (b) is the percentage of samples found in the

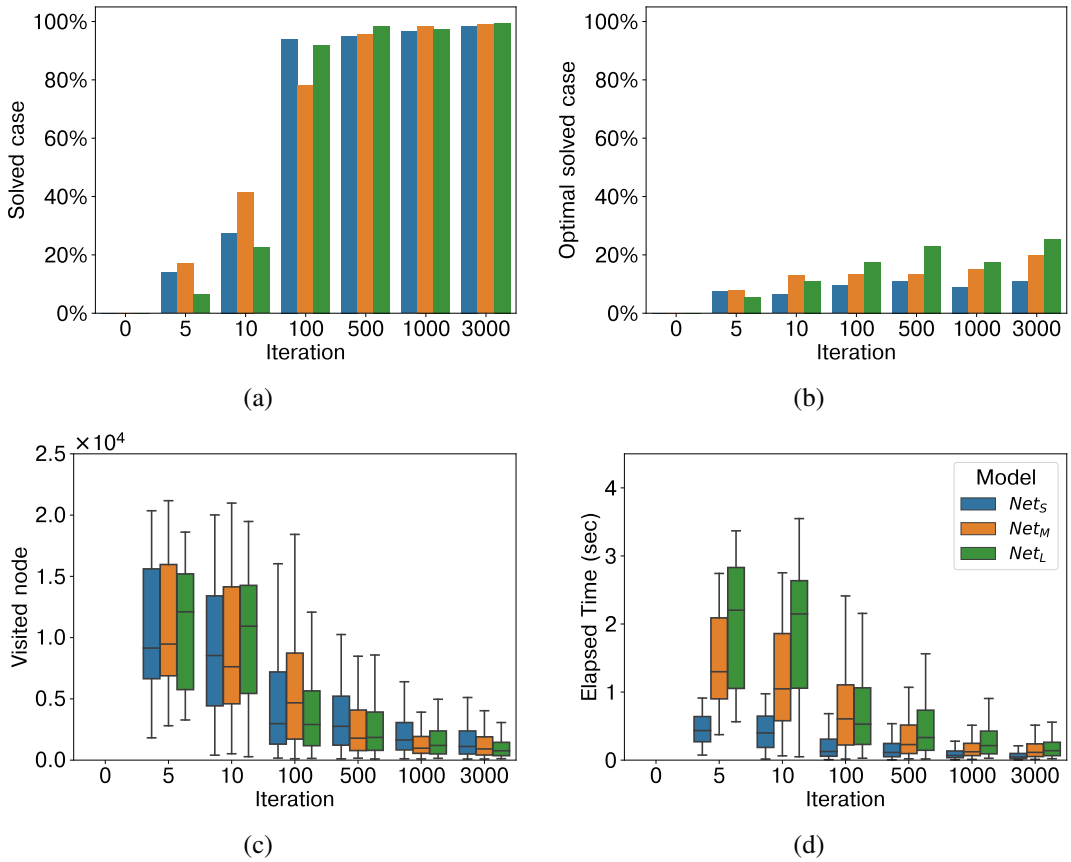


Figure 5.13: Improving the performance of the network according to the iteration; weighted A* search ($\omega : 0.8$) results in search node limitation value 104 at performed 200 random samples (a) Solved ratio, (b) Optimal solved ratio, (c) Visited node for solving, (d) Elapsed time for solving

optimal path at a given search node limitation value of 10^4. Figure 5.12 (c) and (d) investigate the generated visual node and the time it finally took until the path to the goal was found.

Through the above investigation, it can be seen that although not all samples were solved owing to the search node limitation value, the number of samples finding the path to the target and the number of samples finding the optimal path certainly increased as the duration progressed. It can also be observed that the number of visual samples created to find a path to the target and the time consumed are also decreasing.

We further investigate results from weighted A* searches under the same conditions as in Figure 5.13. For this investigation, ω for weighted A* was set to 0.8.

Figure 5.13 (a) showed that some samples were solved even at 10 iterations, i.e., rel-

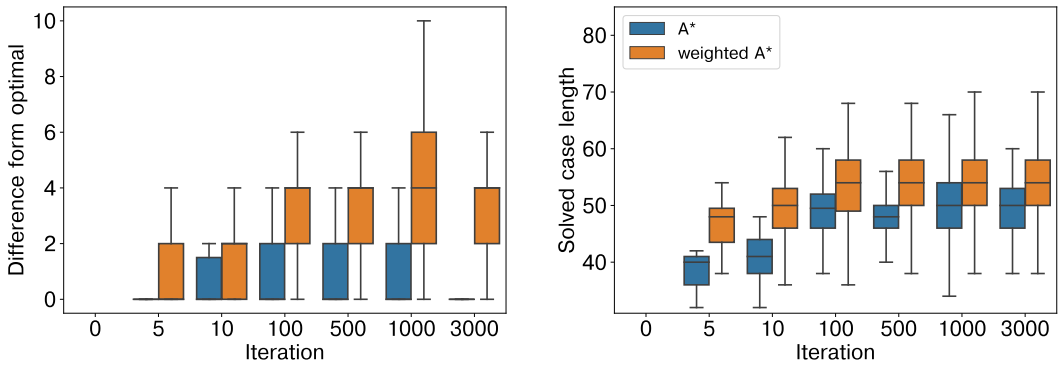


Figure 5.14: Comparison of solved path results in A* and weighted A*; (a) Comparison of the distribution of the difference between the lengths of the solved path and the optimal path, (b) Comparison of the distribution of the lengths of the solved paths

atively few iterations, as it focused on finding the path to the goal, not on finding the optimal path by applying weighted A*. Subsequently, more than 80% samples were solved at 100 iterations, and almost all samples were solved afterwards. In Figure 5.13 (b), the ratio of finding the optimal path was substantially reduced because weighted A* was focused on finding the path to the goal. Nevertheless, it can be seen that the rate of finding the optimal path increases slightly as the duration increases and that as the size of the network increases, the higher the likelihood of the optimal path being found. Subsequently, Figure 5.13 (c) and (d) clearly show that the number of nodes to be searched and the time for the search decrease very significantly as the duration increases. This shows almost all samples can be easily solved in one second at various sizes of all models. This shows almost all samples can be easily solved in one second at various sizes of all models.

Finally, Figure 5.14 shows how different the length of the path found in A* and weighted A* is from the actual optimal path. In Figure 5.14 (a), we investigate how much difference exists between A* and the path solved in weighted A* from the actual optimal path. Most of A* had only a difference of 0 and 2 from the length of the optimal path, and most of them had a difference of 0 in the final training. However, weighted A* usually had a difference of 2 to 4, and this difference did not decrease significantly in final training. Figure 5.14 (b) shows the length of the path solved in A* and weighted A* as a distribution. In this result, it can be seen that as training progresses, the length of the solved paths also increases as the number of solved cases increases. In this experiment, we can see that the distribution of the paths found by

weighted A* is relatively larger than the results of A* because weighted A* finds a longer path rather than the optimal path.

Overall, through these experiments, we can confirm that DAI-WBS has sufficient training from the parameters that work well, making it easy to solve many samples.

5.2.6 Comparison with DAVI

Finally, we experiment with Deep Approximate Value Iteration (DAVI), an algorithm for training heuristics presented in DeepCubeA (Agostinelli et al., 2019), on our proposed network, and investigate the performance differences with our final proposed algorithm DAI-WBS.

$$h(s,t) = \begin{cases} 0, & \text{if } s = t \\ \min\limits_{n \in \text{neighbor of } s} (h(n,t) + \text{cost}(n,s)), & \text{otherwise} \end{cases} \tag{5.6}$$

DAVI iterates the same expression as Equation 5.6 on a given trained sample to be trained, training heuristics to reach the goal in a Value Iteration method. This Value Iteration method has the advantage of being able to train a heuristic naturally without obtaining an actual optimal path. However, the Value Iteration method relies entirely on the trained approximation function for the target to be trained, so it can be settled naturally to override or underestimate, requiring that the approximation function to be trained be very precise. That is why in DAVI, a very large and complex network is proposed as an approximate function; conversely, there is a possibility that it will be very largely overridden or underestimated in a small network like the one we use.

Therefore, in the next experiment, we investigate whether overestimation and underestimation occur as predicted when DAVI is applied to our proposed network, and compare the performance when solving samples with real A* search with our proposed algorithm.

In Figure 5.15, we checked whether the presented training process trains the heuristic tendency with fewer iterations compared to the DAVI and whether it follows the actual heuristic tendency. Figure 5.15 (left) shows a plot of the heuristic trained by applying whole-branch sampling to our DAI. Figure 5.15 (right) is a plot of the heuristic trained through DAVI. In Figure 5.15, the heuristic trained by applying whole-branch sampling to DAI approached the ideal heuristic plot in almost 15 iterations, but the

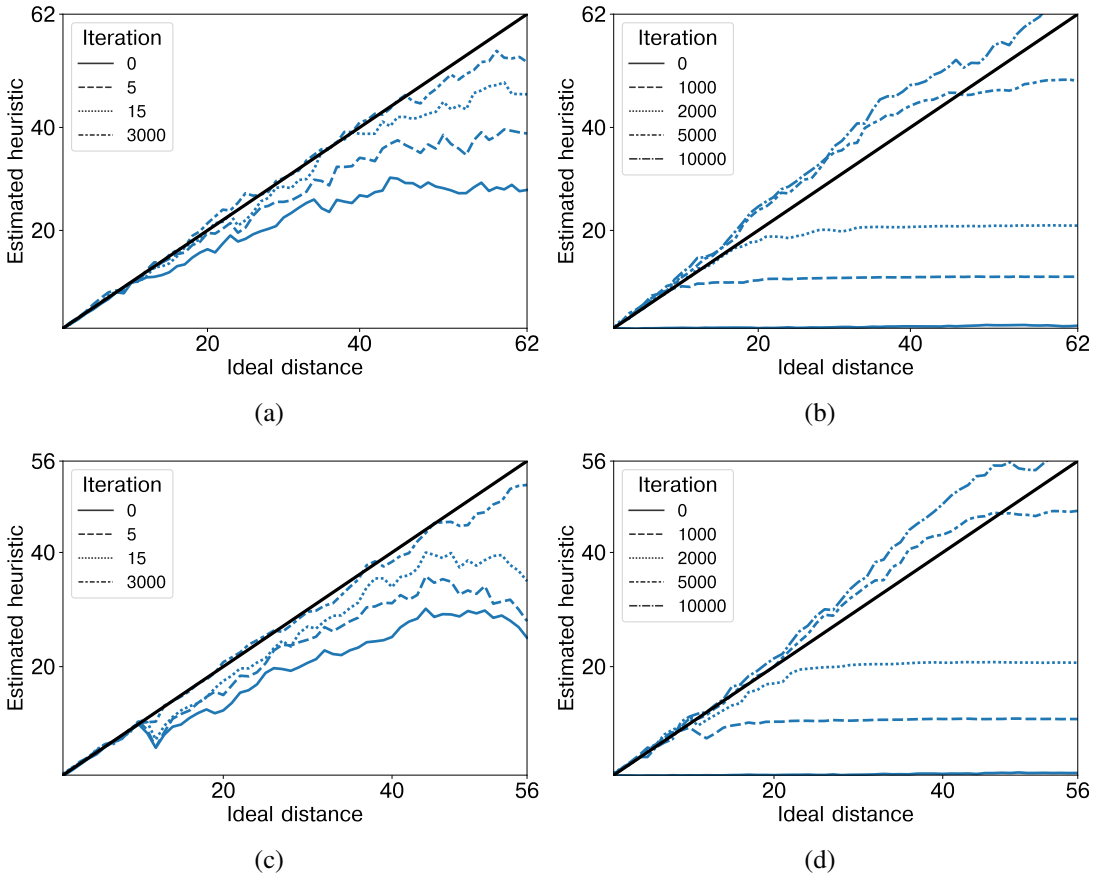


Figure 5.15: Increasing heuristic plot by iteration in two methods; (left) DAI-WBS, (right) DAVI

heuristic trained using DAVI follows the overall heuristic trend only after very many iterations. Furthermore, in the final trained result, it can be seen that DAVI tends to be overestimated as a whole, and the heuristic applied with DAI-WBS is less likely to be overestimated.

Subsequently, in Figure 5.16 and 5.17, we apply two heuristic training algorithms to compare performance. This experiment was conducted in the same 200 samples as the previous experiments, with the search node limitation value at 10^5, and the performance measurements were conducted in a general A* search and weighted A* search at $\omega : 0.8$.

Figure 5.16 is a performance comparison in a pure A* algorithm for finding the optimal path, resulting in solving a given 200 samples over each trained network. First, in Figure 5.16 (a), both DAI-WBS and DAVI solved most of the samples in a given search node limitation of 10^5, indicating that both training methods trained the heuris-

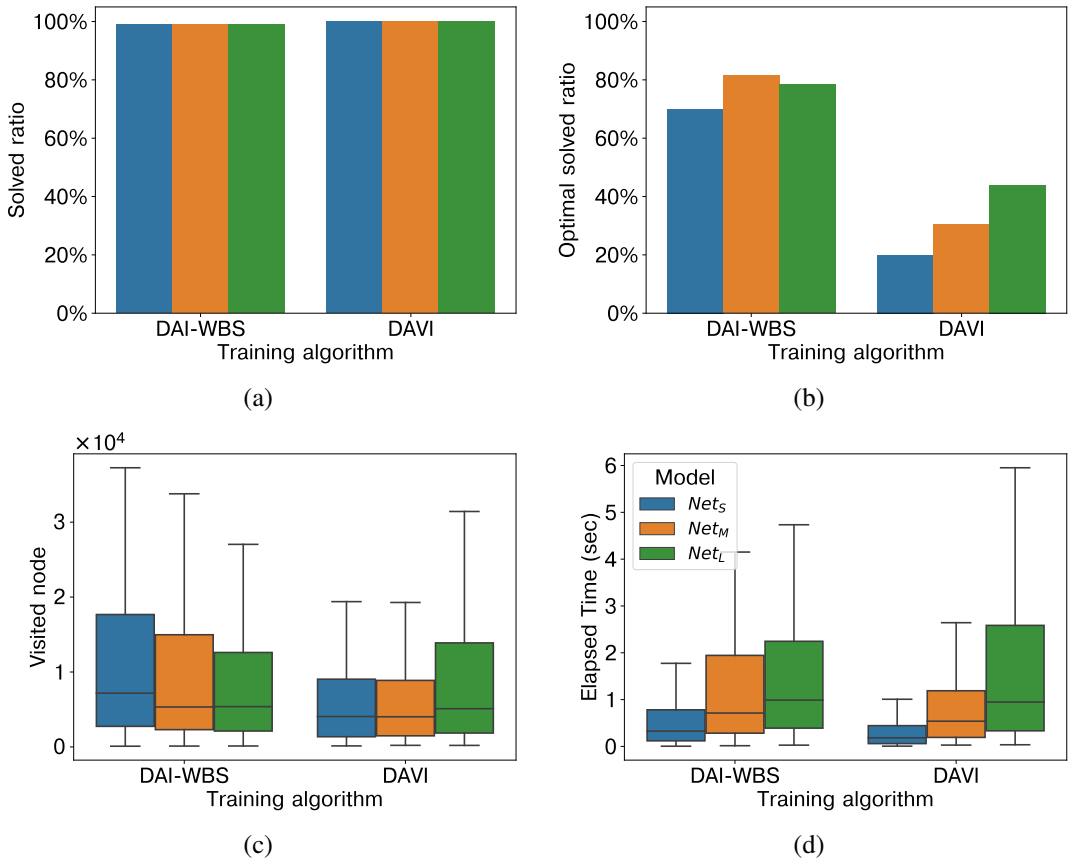


Figure 5.16: Optimal solving performance comparison of the network trained using the DAI-WBS algorithm and DAVI algorithm; A* search (searching for optimal path finding) results in search node limitation value 10^5 at performed 200 random samples. (a) Solved ratio, (b) Optimal solved cases ratio, (c) Visited node for solving cases, (d) Elapsed time for solving cases

tics of the entire environment. In Figure 5.16 (b), networks trained with DAVI show a relatively very low optimal solving ratio. This can be attributed to the fact that networks trained by DAVI are generally overestimated and do not exhibit smooth, directly proportional heuristics, as can be seen in Figure 5.15. In Figure 5.16 (c) and (d), as DAVI is very overestimated, a greedy search proceeds to solve the samples relatively faster.

Figure 5.17 is a performance comparison in weighted A* algorithm for fast path finding, resulting in solving a given 200 samples over each trained network. In Figure 5.17 (a), we solved most of the samples in both cases on the search node limitation of 10^5 given in the same way as the result of A*. In Figure 5.17 (b), both networks showed low optimal solving rates, but 95% or more of results from networks trained with DAVI

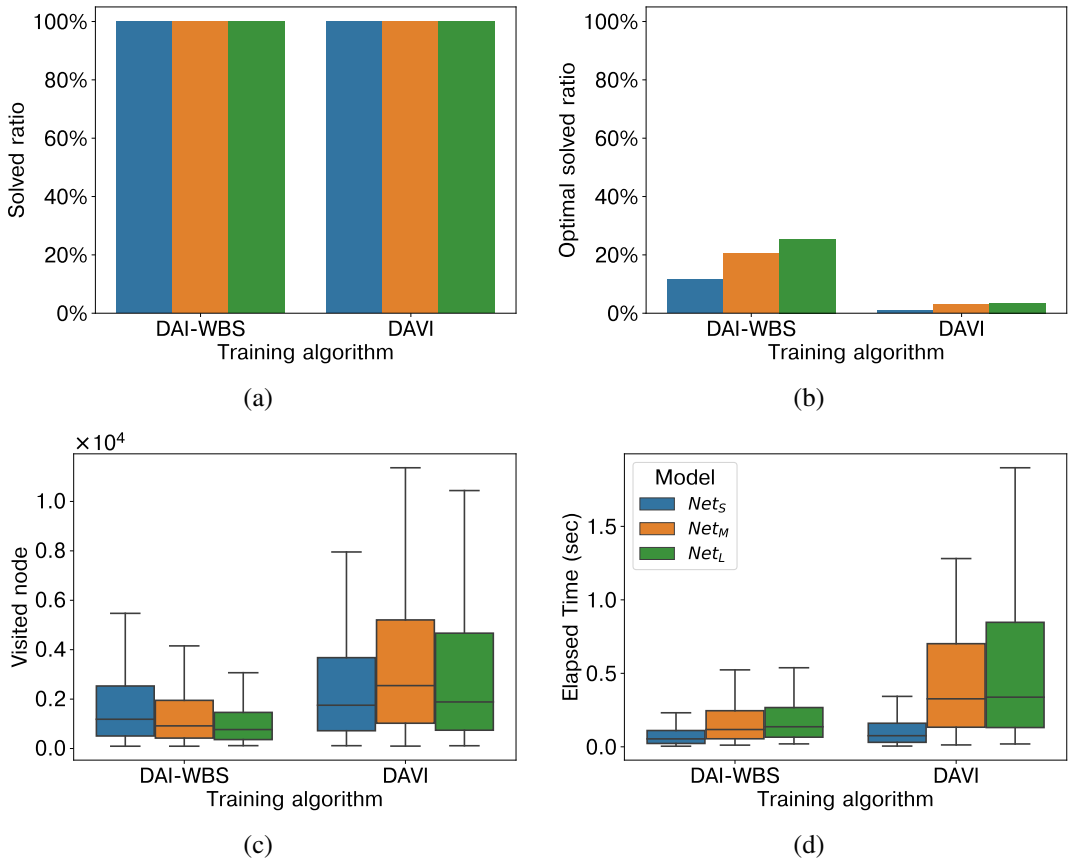


Figure 5.17: Fast solving performance comparison of the network trained on the DAI-WBS algorithm and DAVI algorithm; weighted A* search(searching for fast path finding) results in search node limitation value 10^5 at performed 200 random samples. (a) Solved ratio, (b) Optimal solved cases ratio, (c) Visited node for solving cases, (d) Elapsed time for solving cases

are not optimal paths. In Figures 5.17 (c) and (d), DAI-WBS also performs a greedy search, in which case networks trained with DAI-WBS clearly solve problems faster than DAVI. It can be thought that networks trained with DAI-WBS are more closely proportional than those trained with DAVI, so a more accurate greedy search is performed, resulting in faster search results.

Through these performance evaluations, we can confirm that DAI-WBS is higher in both optimal path search performance and general path search performance than DAVI algorithms in our current proposed network.

Finally, in Table 5.1, we measured the time required to proceed with the training enough to produce sufficient performance in each training method. DAVI took the longest training time because it had to collect a large amount of data for updating the

Table 5.1: Training time required for each method and network size

Methods	Net_L	Net_M	Net_S
DAVI	21 hour	13 hour	10 hour
DAI	18 hour	12 hour	3 hour
DAI-WBS	**2 hour**	**1 hour 30 min**	**1 hour 30 min**

network step by step using value iteration, and repeat the training until the loss was reduced before updating the network. Although DAI and curriculum learning methods in previous chapters were trained very quickly on Net_S, which easily collects data, Net_L and Net_M, which find it difficult to collect data using GPUs, have relatively significantly increased training time. Finally, it can be concluded that DAI-WBS works the same as DAI, while maximizing sampling efficiency so that networks of all sizes can finally be trained in two hours.

5.3 Summary of Chapter 5

This chapter describes how we aimed to increase the training speed by increasing the sampling efficiency of the DAI algorithm presented in the previous chapter. We also aimed to self-organise the data range through which the network would train itself.

The DAI's basic data collection feature, suggested in the previous chapters, used the results of A*. In most cases, the number of heuristics that need to be evaluated to produce a single A* result is tens of thousands, but the number of data generated is very small compared to hundreds, which is considered to have low sample efficiency. In addition, in the previous chapters, there was a limit for a sample range to perform the A* search, which is usually organised intentionally depending on the rules.

To overcome these limitations, we suggested the following methods. First, we increased the sample efficiency by collecting other branches created by performing A* search and making them available in training, rather than using only the results generated by A* search. Thus, as heuristics were trained, the process of collecting branches naturally led to the anticipation that the distribution of the collected data would lean in a skewed oval toward the target point of the sample. This was verified later through an experiment.

Second, we argue that using the branch collection method proposed in the first, the

number of low-level data generated becomes too large to proceed properly, and we then propose a method to properly adjust the ratio of the number of data generated in one sample to the difficulty.

We propose the above two methods, and then we test the methods proposed in the previous chapter and the methods proposed by other researchers to train heuristics, comparing the training time with the algorithm proposed in this chapter, the training degree of heuristics, and the actual puzzle-solving performance. From the results of this experiment, it was confirmed that while noticeably reducing the training time, several indicators showed good performance compared to other methods.

Chapter 6

Conclusions

In this chapter, we will be discussing the purpose, attempts and results of the learning method within the proposed graph structure environment. Existing puzzle resolution methods incorporating deep learning have limitations in that the size of the network used is very large, and there is a problem of overestimation of value iteration. To solve these challenges, a problem where the size of the network was above a certain level was to be guaranteed. We focused on extracting and learning high-quality data by analyzing the results obtained while searching the real environment. Based on the A* graph surge algorithm, we have presented an efficient method to collect all node-to-node heuristic data of the environment. The proposed method also enables efficient search with a data-learned network, allowing faster exploration of the entire range of the environment. The proposed approach that incorporates sub-methods can be divided into three chapters.

6.1 Neural Heuristic and Bucket priority queue

In chapter 3, our main aim was to improve the 15-puzzle solver, which uses the A* search. Studies that have explored the 15-puzzle solver have tried to introduce structural improvements in A* search or aimed at designing a heuristic to interpret the 15-puzzle a bit better. However structural improvements in A* search, just like the IDA*, sacrifice the search speed in return for suppressing memory increase usage, a trade-off just like weighted A*, where speed is increased but the optimal route cannot be guaranteed. An improvement plan for a heuristic design for the 15-puzzle has been suggested by utilizing databases rather than a function based on mathematical interpretation and

rules. Heuristics utilizing these databases specialize in calculating a fixed objective spot. Even the DeepCubeA(Agostinelli et al., 2019), which trains heuristic functions via a network, targets heuristic training aimed at a fixed objective.

We suggested other methods in this study that differs from these existing ones. First, we proposed a data structural optimization method for A* materialization that fits the 15-puzzle environment. To find an optimal route for the 15-puzzle, optimization of the number of slides is key. Further, all costs should be defined as positive integers By adopting these properties, a trick was suggested wherein all the search sample inputs and outputs were considered at the same time to complete, using the bucket priority queue rather than the heap data structure used to materialize the priority queue of most A* algorithms, enabling us to confirm a noticeable difference between the times taken in terms of the search frequency between the case of utilizing the bucket priority queue and the heap structure.

We suggested and utilized a method for extracting the heuristic of all node to node on the optimal path of conventional heuristic A* search for generating data to train the heuristic. This way, we generated universal heuristic data for higher sampling efficiency to represent all start and objective nodes, not just for one fixed objective node. However, the condition of starting and the objective point were defined as the square of a state space of a fixed conventional environment with a fixed objective point. In such cases, the problem of training a network can grow out of proportion. To fix this, we compressed the state space on the 15-puzzle problem and processed a length of the state space to more easily interpret form and input into the network. With this, distance prediction between the state spaces can be efficiently carried out even if only a very small network is used compared to conventional studies on heuristic training, leading to highly accurate heuristic training within the samples we gathered.

By adopting the aforementioned two improvements, we confirmed that the search was concluded quickly on the heuristic that trained the 15-puzzle solver, aiming for all the nodes via the network rather than conventional heuristics with the heap data structure on relatively harder samples.

The following contributions were made. First, a generation method for an efficient heuristic dataset to correspond to all node-to-node was suggested. Second, training for an incredibly small neural heuristic was made possible because of processing node-to-node data for better interpreting the state space of the 15-puzzle, which was increased

in square, compared to conventional studies.

6.2 Deep A* iteration

In chapter 4, we aimed to self-train a heuristic without human knowledge through enhanced learning without using conventional heuristics to train a network. The data collection method described in the previous chapter utilized A* search, which used a conventional heuristic. However, traditional heuristics utilizing A* to solve a harder sample (which needs a minimum of more than 60 slides) requires significantly longer time and larger memories. Thus, a neural heuristic can be used to solve harder samples faster, as explained in the last chapter.

By utilizing this technique, we gathered data to train the network through a neural heuristic to present the Deep A* Iteration (DAI) algorithm using an enhanced learning method to converge neural heuristics. However, because of the conditions for admissible heuristics, heuristics comprising networks are not guaranteed to produce an optimal path. This fails to lead to a possibility that data close to the ideal heuristic, which is the data of the optimal path that we aimed for,can be generated through the overestimated data by the overestimated network heuristic. We thus presented the theoretical DAI algorithm that converges to the ideal heuristic and provided experimental evidence to validate this theory. The DAI algorithm can confirm whether a network is overestimated or underestimated compared to the ideal heuristic and demonstrate whether the results will converge to the ideal heuristic value.

Another problem that can occur in the DAI algorithm is that if the neural heuristic is in a state wherein nothing has been trained, it takes a significantly longer time and more memory to solve harder samples. The efficiency of A* search is defined by how well the heuristic predicts the real cost to go, so it takes far longer to solve harder samples and generate data in the untrained network. To solve this, we suggested a curriculum approach to increase sample complexity in situations where the neural heuristic has trained enough of generated samples starting from the easiest. Our approach showed that the time taken to generate a training sample was greatly reduced to the time taken to generate specific complexity of the samples when the curriculum was applied versus when it was not. Finally, we confirmed that the neural heuristic after training could generate data and train from very complex samples.

Our contributions are as follows. First, we proposed the DAI and then verified its

convergence to the ideal heuristic sample. Second, by applying the curriculum method approach, we suggested a training method to generate samples of the overall range, which does not take a long time to generate without any kind of training.

6.3 Deep A* Iteration with self-organizing local view

In chapter 5, we aimed to increase the training speed by increasing the sampling efficiency of the DAI algorithm presented in the previous chapter. We also aimed to self-organize the data range through which the network would train itself. The DAI's basic data collection feature, suggested in the previous chapters, used the results of A*. In many cases, several heuristics need to be evaluated to obtain one A* value that can reach tens of thousands; however, large amounts of data generated can be thought to contribute to lower sampling efficiency because it is only a few hundred, which is far fewer compared to the former. Also, in the previous chapters, there was a limit for a sample range to perform the A* search, which is usually organized intentionally according to the rules.

To overcome these limits, we suggested the following methods. First, we increased the sampling efficiency by collecting the other branches made while performing the A* search and made them available in training rather than using only the results generated by the A* search. Thus, as heuristics were learned, the process of collecting branches naturally led to the anticipation that the distribution of the collected data would be leaning in a skewed oval toward the target point of the sample. This was verified later through an experiment.

Second, if we used the branch collection method suggested, the lower difficulty data generated increased exponentially, resulting in the iteration training not proceeding correctly. We thus suggested a method to control the ratio between the amount of data collected and sample difficulty.

Our contribution is as follows. First, we suggested the branch sample technique to exponentially increase the sampling efficiency of the DAI algorithm. Second, to suppress the negative impacts of the branch sample technique, we limited the exponential growth of the training data and suggested methods to increase data generation on the harder cases, thereby increasing the efficiency of iteration training.

6.4 Future Work

The network that we previously suggested, in chapter 3, used the parsing technique on the state space to solve the 15-puzzle The reason why these networks have relatively high precision compared to their size is that they play a key role in allowing human knowledge to interpret the location relationship of the tiles necessary to go from the current tile to the target tile in advance. Strictly speaking, human knowledge acts as interference when training a network. Thus, to train heuristics efficiently and quickly without human intervention, we will need a network to interpret the position and relations of the tiles between the current and target situation efficiently.

Additionally, in environments that have a complex state space compared to the 15-puzzle, for precise heuristic prediction, further research is needed on network structure to efficiently interpret relations within states inside their own structures. Honestly, our research is not the only one that investigates a network structure or the location order of certain objects through in-depth analysis In natural language processing research, an attention-based network structure was suggested to interpret the location and the meaning of each word in a sentence. These attention structures were confirmed to work effectively in terms of interpreting patterns inside a 2D image input or location relation of objects on image processing research. Following this trend, it is vital to design a small network using an attention structure to design a universal neural heuristic with high accuracy without requiring any input processing. Further research will be required to apply such methods on a complex environment rather than just the 15-puzzle one.

Appendix A

Figure A.1 is a array of optimal path states in case 1, mentioned in Table 3.1. This Figure A.1 is suggested to clearly show on what reason that each conventional heuristics widens the length between ideal heuristics.

When measuring heuristics, target state is set to start(0). Firstly Hamming distance, which counts whether tiles are identical in same location, shows heuristics increase from 1 to 5 in movement with same order, but in sixth the former moved number got moving so heuristic isn't able to increase. Secondly Manhattan distance, which calculate a combined value of x,y length between times of same numbers, shows heuristics increase from 1 to 14 in movement with same order, but in 15th the number 13 is returned to the same row with start(0), so heuristics aren't increased, rather decreased. Thirdly Linear conflict, which calculate values by counting additional conflicts on each row and axis on Manhattan distance, heuristic was increased on order 15 where Manhattan distance failed to increase.

The reason is in order 15, number 13 and 15 exist in the same row, conflict which is opposite to order of start(0) can be successfully sensed. However in order 29, even though number 2 came back to the same column, since there isn't any additional conflicts sensed, heuristics are decreased.

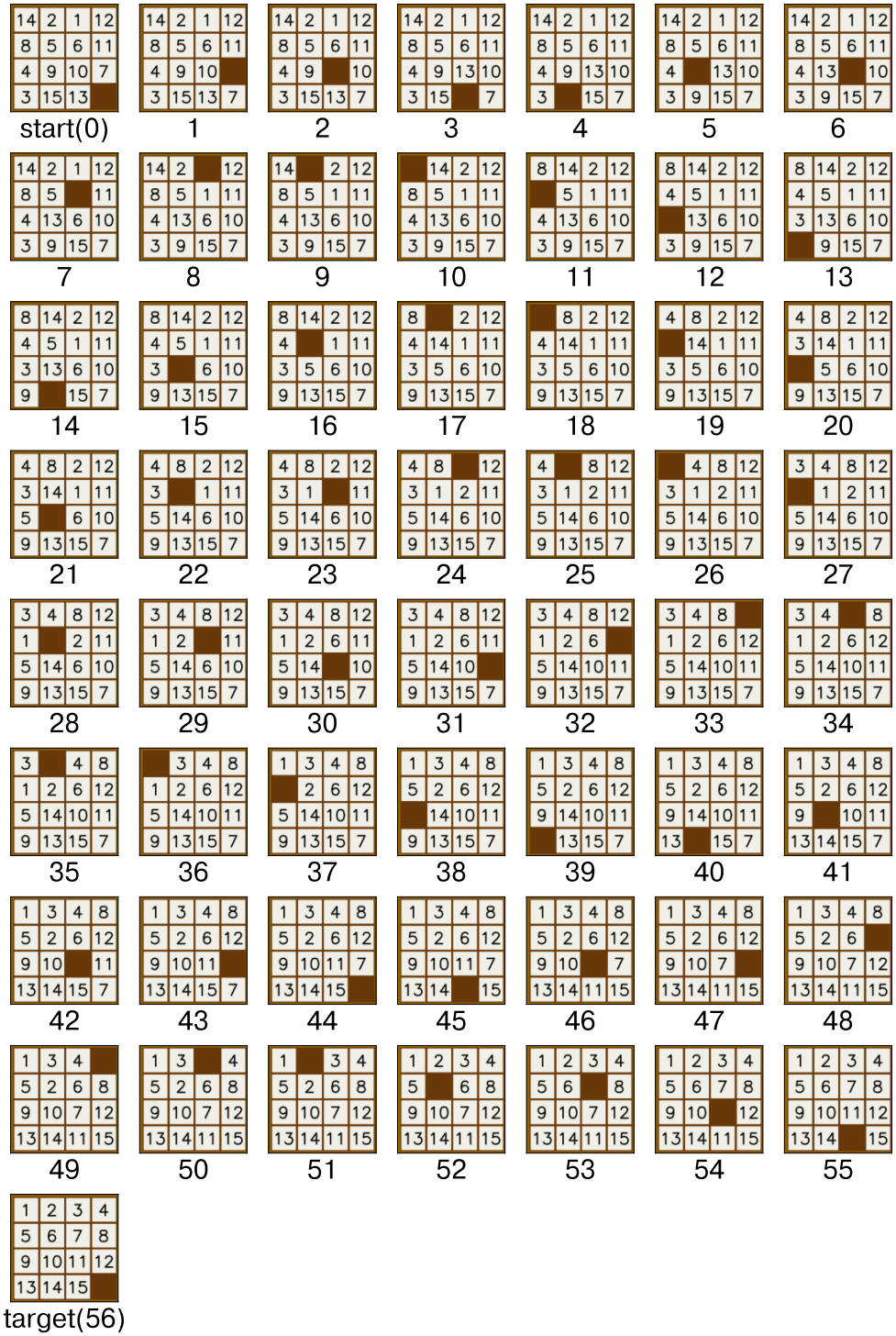


Figure A.1: Optimal Path of the Case1(56 Length)

Bibliography

Agostinelli, F., McAleer, S., Shmakov, A., and Baldi, P. (2019). Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1(8):356–363.

Agostinelli, F., Shmakov, A., McAleer, S., Fox, R., and Baldi, P. (2021). A* search without expansions: Learning heuristic functions with deep Q-Networks. *arXiv preprint arXiv:2102.04518*.

Bellemare, M. G., Dabney, W., and Munos, R. (2017). A distributional perspective on reinforcement learning. In *International Conference on Machine Learning*, pages 449–458. PMLR.

Bhasin, H. and Singla, N. (2012). Genetic based algorithm for N-Puzzle problem. *International Journal of Computer Applications*, 51(22).

Bischoff, B., Nguyen-Tuong, D., Markert, H., and Knoll, A. C. (2013). Solving the 15-Puzzle game using local value-iteration. In *SCAI*, pages 45–54.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. (2018). JAX: composable transformations of Python+NumPy programs.

Chaslot, G., Bakkes, S., Szita, I., and Spronck, P. (2008). Monte-carlo tree search: A new framework for game ai. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 4, pages 216–217.

Corli, S., Moro, L., Galli, D. E., and Prati, E. (2021). Solving Rubik's cube via quantum mechanics and deep reinforcement learning. *Journal of Physics A: Mathematical and Theoretical*, 54(42):425302.

Culberson, J. and Schaeffer, J. (1994). Efficiently searching the 15-Puzzle. *University of Alberta*.

Culberson, J. C. and Schaeffer, J. (1998). Pattern databases. *Computational Intelligence*, 14(3):318–334.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271.

Doran, J. E. and Michie, D. (1966). Experiments with the graph traverser program. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 294(1437):235–259.

Felner, A., Korf, R. E., and Hanan, S. (2004). Additive pattern database heuristics. *Journal of Artificial Intelligence Research*, 22:279–318.

Fortunato, M., Azar, M. G., Piot, B., Menick, J., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., et al. (2017). Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*.

Fujimoto, S., Hoof, H., and Meger, D. (2018). Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323. JMLR Workshop and Conference Proceedings.

Goldreich, O. (2011). Finding the shortest move-sequence in the graph-generalized 15-Puzzle is np-hard. In *Studies in complexity and cryptography. Miscellanea on the interplay between randomness and computation*, pages 1–5. Springer.

Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. (2018). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR.

Hamming, R. W. (1950). Error detecting and error correcting codes. *The Bell system technical journal*, 29(2):147–160.

Hansson, O., Mayer, A., and Yung, M. (1992). Criticizing solutions to relaxed models yields powerful admissible heuristics. *Information Sciences*, 63(3):207–227.

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107.

Hasselt, H. (2010). Double Q-Learning. *Advances in neural information processing systems*, 23.

Huber, P. J. (1992). Robust estimation of a location parameter. In *Breakthroughs in statistics*, pages 492–518. Springer.

Konda, V. and Tsitsiklis, J. (1999). Actor-critic algorithms. *Advances in neural information processing systems*, 12.

Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109.

Korf, R. E. and Felner, A. (2002). Disjoint pattern database heuristics. *Artificial intelligence*, 134(1-2):9–22.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25.

Lam, S. K., Pitrou, A., and Seibert, S. (2015). Numba: A LLVM-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–6.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

Lin, L.-J. (1992). *Reinforcement learning for robots using neural networks*. Carnegie Mellon University.

Loshchilov, I. and Hutter, F. (2017). Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*.

McAleer, S., Agostinelli, F., Shmakov, A., and Baldi, P. (2018). Solving the Rubik's cube without human knowledge. *arXiv preprint arXiv:1805.07470*.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement

learning. In *International conference on machine learning*, pages 1928–1937. PMLR.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Parberry, I. (2014). A memory-efficient method for fast computation of short 15-Puzzle solutions. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(2):200–203.

Pearl, J. (1984). *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc.

Pohl, I. (1970). Heuristic search viewed as path finding in a graph. *Artificial intelligence*, 1(3-4):193–204.

Ratner, D. and Warmuth, M. (1990). The (n2- 1)-puzzle and related relocation problems. *Journal of Symbolic Computation*, 10(2):111–137.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2015). Prioritized experience replay. *arXiv preprint arXiv:1511.05952*.

Scheiermann, J. and Konen, W. (2022). Alphazero-inspired general board game learning and playing. *arXiv preprint arXiv:2204.13307*.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural networks*, 61:85–117.

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., et al. (2020). Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609.

Schulman, J., Levine, S., Abbeel, P., Jordan, M., and Moritz, P. (2015). Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016).

Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44.

Sutton, R. S., Barto, A. G., et al. (1998). Introduction to reinforcement learning. *A Bradford Book*.

Van Hasselt, H., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double Q-Learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30.

Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., and Freitas, N. (2016). Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR.

Watkins, C. J. and Dayan, P. (1992). Q-Learning. *Machine learning*, 8(3):279–292.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256.

Willianms, R. (1988). Toward a theory of reinforcement-learning connectionist systems. *Technical Report NU-CCS-88-3, Northeastern University*.