# AN ABSTRACT OF THE THESIS OF

Kai Zeng for the degree of Master of Science in Computer Science presented on December 10, 2021.

Title: Optimizing Interconnection Topology using Deep Reinforcement Learning

Abstract approved: _____

Lizhong Chen

As the number of nodes in high-performance computing (HPC) systems continues to grow, it becomes increasingly important to design scalable interconnection network topologies. Prior work has shown promise in adding random shortcuts on top of an existing topology to reduce average hop count and network diameter, but has been limited to naïve and heuristic ways to add shortcuts. In this work, we propose a novel and systematic approach that combines deep reinforcement learning and Monte Carlo tree search. The proposed framework is able to explore the large design space of adding shortcuts effectively. Compared with state-of-the-art topologies for HPC systems, the solution found by our framework significantly reduces the network diameter by 15% and shortens the average hop count by 16%.

# Optimizing Interconnection Topology using Deep Reinforcement Learning

by

Kai Zeng

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented December 10, 2021
Commencement June 2022

Master of Science thesis of Kai Zeng presented on December 10, 2021.

APPROVED:

---

Major Professor, representing Computer Science

---

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

---

Kai Zeng, Author

# ACKNOWLEDGEMENTS

First of all, I would like to express my most sincere thanks to my advisor Lizhong Chen. During the research process, he was able to give me many enlightening suggestions to help me make progress in the research step by step; in the process of writing the paper, he carefully reviewed it with great patience, put forward suggestions for revision, and urged me to repeat Thinking has provided a great help to the completion of this article.

Secondly, I would like to thank my intern company, WeRide. After I entered the company, my colleagues taught me to gain more professional knowledge, broaden my vision, and greatly improved my work ability and creativity. The improvement of personal ability gained from the company has greatly helped my research ability.

In addition, I would like to thank my friend Junkun and Wenxuan, who taught me to look at and solve problems from the perspective of a doctoral student, which also greatly improved my research ability.

Finally, I want to thank OSU for providing such a beautiful and suitable campus environment for learning. Apart from research, I participated in school swimming training courses, which exercise my body well and also improve my work efficiency when doing research.

# TABLE OF CONTENTS

# TABLE OF CONTENTS (Continued)

# LIST OF FIGURES

# LIST OF TABLES

# LIST OF ALGORITHMS

# Chapter 1: Introduction

High Performance Computing (HPC) system refers to computers that can perform large amounts of data and high-speed operations that personal computers cannot handle. This type of cluster mainly solves the calculation of large-scale scientific problems and the processing of massive data, such as scientific research, weather forecasting. Since the switching delay is relatively large compared to the line and microchip injection delay, Therefore, the structural optimization of interconnection topology is very important for HPC systems. Our goal is to generate better network topology.

Since the design space of generated graph is very complicated, it is difficult to generate a good topology. This article investigates the methods of generating topologies, such as common HPC system topologies, Distributed Loop Networks (DLN), complex networks, and random shortcut topologies algorithm. This article also investigates the methods explored in the huge design space, such as machine learning and deep reinforcement learning methods.

This paper choose to add shortcuts to the base topology, to generate better topology. The model draws on the application of Deep Reinforcement Learning (DRL) Framework on Routerless NoC Case, and implements a model based on DRL and MCTS, and explained various details in the model. This model is dedicated to solving the optimization problem of interconnection topology. That is,

for an initial interconnection topology graph, we hope to add some shortcuts to optimize the graph structure and reduce the diameter of the graph and the average hop count between all point pairs.

For the results, we compare the DRL model with two random models. By adjusting the size of the network, the maximum degree of each node, and the maximum number of edges in the network, the DRL model will produce better results. We also showed some examples of topology generation. At the end of this paper, we also discuss the derivative work that this model can complete, such as optimizing the topology if edges have weights.

This article is divided into 7 chapters, and the content of each chapter is roughly as follows:

- Chapter 1 is introduction, which roughly describes the task to be completed in this thesis;
- Chapter 2 is background, which mainly talked about the technology and principles of DRL and MCTS, and also discusses the complexity of design space;
- Chapter 3 investigates some reference methods and techniques, these methods directly or indirectly inspire the model of this article, some methods will be compared with the model of this article as a comparison method in the follow-up;
- Chapter 4 proposed the framework design of our model, and specifically describes the implementation methods of each module;

- Chapter 5 briefly describes the parameters and methods of program operation;
- Chapter 6 exhibits the results of our model and compares them with the methods described in Chapter 3;
- Chapter 7 discusses the scalability of our model, that is, more problems can be solved after modifying the constraints;
- Chapter 8 summarizes the work of this article.

# Chapter 2: Background

In this chapter, first expounds the extensiveness of interconnection topology in HPC system, and then explains the complexity of design space. After that, the main techniques designed in this article are introduced: deep reinforcement learning and Monte Carlo tree search

## 2.1 Design Space Complexity in HPC system topology

High Performance Computing (HPC) systems refers to computers that can perform large amounts of data and high-speed operations that ordinary personal computers cannot handle. The basic components are not much different from the concept of a personal computer, but the specifications and performance are much more powerful. Most of the existing supercomputers have a computing speed of more than one trillion (trillion, not one million) times per second. This type of cluster mainly solves the calculation of large-scale scientific problems and the processing of massive data, such as scientific research, weather forecasting, computational simulation, military research, CFD/CAE, biopharmaceuticals, gene sequencing, image processing, and so on.

Large-scale parallel applications deployed on next-generation high-performance computing (HPC) systems will suffer communication delays that can reach hun-

dreds of nanoseconds. Therefore, the construction or development of low-latency networks is particularly important in these systems. Since the switching delay is relatively large compared to the line and microchip injection delay (for example, about 100 nanoseconds in InfiniBand QDR), in order to achieve low delay, the switch should have a better topology (for example, smaller diameter and lower average shortest path length), both are measured by the number of switch hops. Fortunately, high-cardinality switches with dozens of ports are now available. Compared with traditional high-diameter topologies, these switches can be designed with low-latency topologies that use more links per switch.

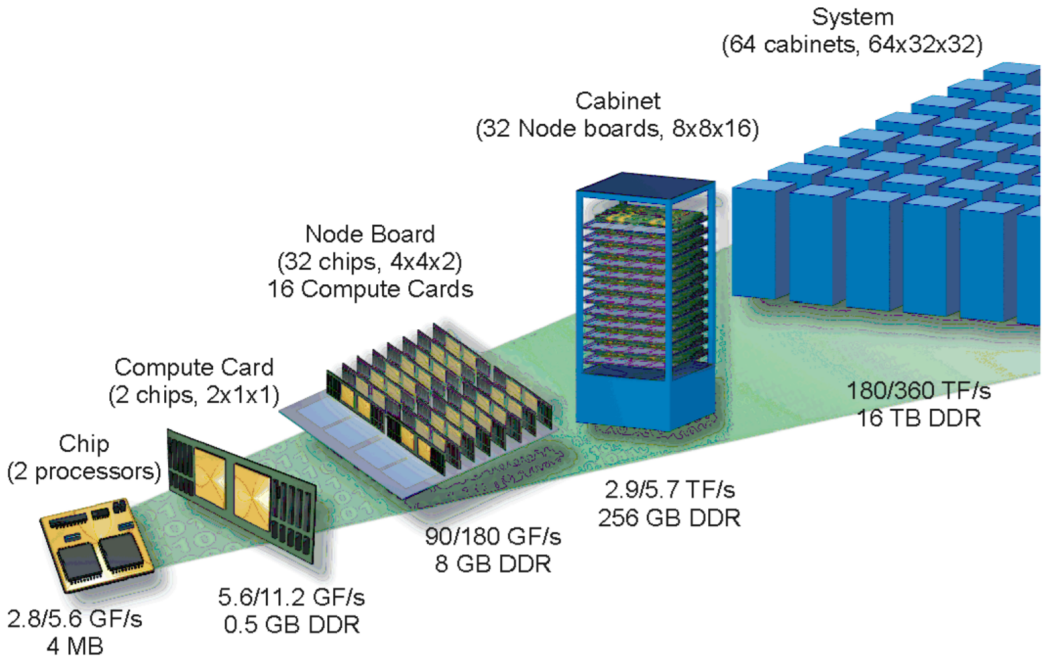


Figure 2.1: High-level organization of the Blue Gene/L supercomputer.

Figure 2.1 shows the structure of the IBM Blue Gene/L Supercomputer[3]. Two

chips share a compute card that also contains SDRAM-DDR memory. Sixteen compute cards can be plugged in a node board. A cabinet with two midplanes contains 32 node boards for a total of 2048 CPUs. The complete system has 64 cabinets and 16 TB of memory. Generally speaking, topologies are widely used in HPC systems. In a cabinet, 32 node boards will be connected to each other according to a certain topology; in Blue Gene/L, 64 cabinets will also be connected according to a certain topology. The widespread topology connection makes the delay also widespread in the HPC system. Therefore, the structural optimization of interconnection topology is very important for HPC systems.

For the HPC system, we optimized its topological structure, thereby reducing the transmission time of information in the system, thereby increasing the calculation speed of the HPC system. Then for the same amount of calculation, the HPC system can get results in less time, which can save a lot of electricity bills. Taking weather forecast as an example, the optimized HPC system can analyze more weather data and run more complex algorithms to get more accurate weather forecast results.

Design space complexity in HPC system problem poses a significant challenge requiring efficient exploration. The topology problem in this article can be abstracted into a graph theory problem. There are $n$ switches in the HPC system, and each switch has a total of $d$ ports and a maximum of $m$ shortcuts; that is, there are $n$ nodes in the graph, and the degree of each node is at most $d$, and there are at most $m$ edges. Then there can be at most $C(n,2) = \frac{n(n-1)}{2}$ edges in the graph, so there are a total of $C(\frac{n(n-1)}{2}, m)$ available graphs. Taking IBM

BlueGene/L Supercomputer as an example ($n = 64$ and assume $m = 128$), there will be $C(64, 2) = 2016$ available edges and $C(2016, 128) \approx 10^{205}$ available graphs. Suppose we use IBM Blue Gene/L to verify a feasible solution, and its computing speed is about 360 Teraflops of peak computing power[3]. Assuming that the time complexity of verifying each feasible solution is $O(1)$, the processing time is $\frac{10^{205}}{360 \times 10^{12}} \approx 2.7 \times 10^{194}$ seconds $\approx 8.5 \times 10^{186}$ years. Therefore, it is completely impossible to use this exhaustive method.

## 2.2 Reinforcement Learning

Machine learning (ML) has come a long way in the last two decades, from a laboratory curiosity to a real technology with extensive commercial use. Machine learning has emerged as the preferred approach for producing practical software for computer vision, speech recognition, natural language processing, robot control, and other applications in artificial intelligence (AI).

**Reinforcement learning:** Reinforcement learning (RL) is the study of how an agent can interact with its environment to learn a policy which maximizes expected cumulative rewards for a task. Recently, RL has experienced dramatic growth in attention and interest due to promising results in areas like playing Go (Silver et al. 2016)[13]. Recent deep reinforcement learning (DRL) techniques, in particular, enable efficient exploration in vast design spaces where conventional design strategies may be inadequate.

Reinforcement learning is also a learning, forecasting, and decision-making ap-

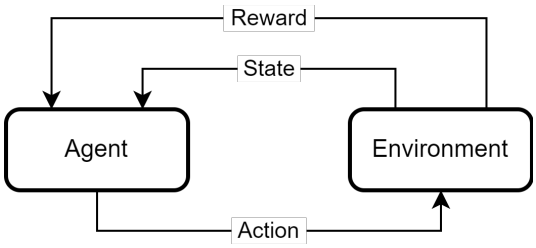


Figure 2.2: Reinforcement Learning Framework

proach framework. Reinforcement learning is likely to help solve an issue if it can be stated or translated into a sequential decision problem with defined state, action, and reward. Reinforcement learning, in general, has the potential to assist in the automation and optimization of humanly created strategies. Supervised learning often addresses one-time difficulties, focuses on short-term benefits, and considers immediate returns, whereas reinforcement learning considers sequence problems, has a long-term viewpoint, and considers long-term returns. This long-term perspective on reinforcement learning is crucial for discovering the best solutions to a variety of issues. For example, if only the nearest neighbor nodes are considered in the shortest path problem, the shortest path may not be obtained.

**Deep Reinforcement Learning:** Deep learning breakthroughs have prompted academics to reconsider the uses of deep neural networks (DNNs) in a variety of fields. Deep reinforcement learning is one of the outcomes, which combines DNNs and reinforcement learning techniques to solve complicated problems. Through efficient data-driven exploration based on DNN output, this synthesis mitigates data reliance without introducing convergence difficulties. These ideas have recently been applied to Go, a grid-based strategic game in which players position

stones. A trained policy DNN learns optimal actions in this model by exploring a Monte Carlo tree that records DNN-suggested actions during training[13, 14]. By developing a sequence of actions with superior cumulative rewards, deep reinforcement learning can beat traditional reinforcement learning.

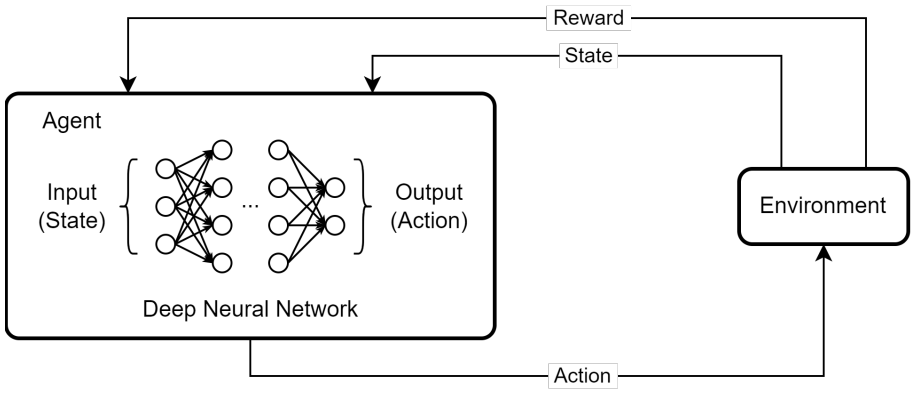


Figure 2.3: Deep Reinforcement Learning Framework

## 2.3 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a method of making optimal decisions in artificial intelligence problems, generally in the form of move planning in combinatorial games. It combines the generality of stochastic simulation and the accuracy of tree search.

The rapid attention of MCTS is mainly due to the success of the computer Go program and its potential application to many difficult problems. Beyond the game itself, MCTS can theoretically be used in any field where (state, action) is used to define and use simulation to predict output results.

The basic MCTS algorithm is very simple: according to the output results of the simulation, a search tree is constructed according to the nodes. The process can be divided into the following steps:

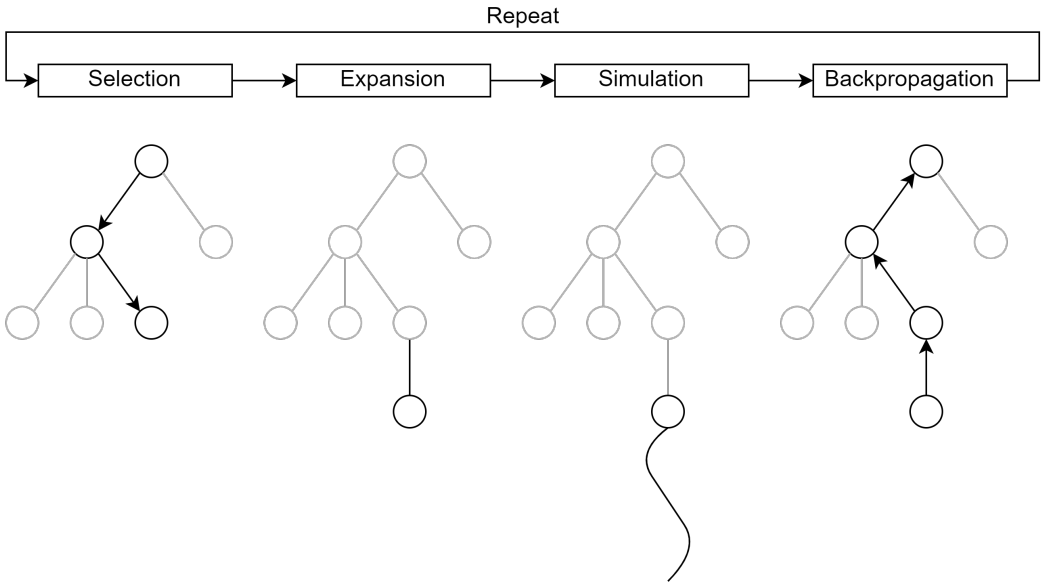


Figure 2.4: Monte Carlo Tree Search Framework

- **Selection:** Starting from the root node R, select the best child nodes in a edge (explained) until the leaf node L is reached.
- **Expansion:** If L is not a termination node (that is, leading to the termination of the game game selection), then create more word child nodes, one of which is C.
- **Simulation:** Run the simulation output from C until the end of the game.

- **Back-propagation:** Update the current action sequence with the simulation result output.

MCTS estimates will be unreliable at the beginning of the search, and will eventually reach a fixed time estimate, and the best estimate can be reached under infinite time.

# Chapter 3: Related Work

## 3.1 Graphs with Low Diameters

For decades, graph theorists have been studying the problem of maximizing the number of vertices in a graph with a given diameter and degree, and trying to get close to the famous Moore boundary[5]. Several graphs with easy-to-handle hierarchical structure and good diameter properties have been proposed for interconnection networks, including the well-known De Bruijn graph[12], $(n, k)$-star graph[1] and so on. These diagrams are rarely used for interconnection topologies in current HPC systems. However, their diversity indicates that there is a lot of room for the design of interconnect topologies.

## 3.2 Topologies of HPC Systems

Direct topologies: each node has direct point-to-point link to a subset of other nodes in the system[4]. Popular direct topologies include $k$-ary $n$-cubes, with a degree of $2n$, which lead to tori, meshes, and hypercubes. Each topology leads to a specific trade-off between degree and diameter. These topologies are regular, which means that all switches have the same degree, because each switch is connected to the same number of switches. (3-ary 2-cube (2-D torus) Graph)

Indirect topologies: each node is connected to an external switch, and switches

have point-to-point links to other switches[4]. They have low diameter at the expense of larger numbers of switches when compared to direct topologies. The best known indirect topologies are Fat trees, Clos network and related multi-stage interconnection networks such as the Omega and Butterfly networks. (Fat-Tree)

More recently, variations of these networks, such as the flattened butterfly[9], have been proposed as a way to improve cost effectiveness. The flattened butterfly gives lower hop count than a folded Clos and better path diversity than a conventional butterfly.

## 3.3 Distributed Loop Networks (DLN)

The distributed loop network (DLN) consists of a simple ring topology, to which chord edges or shortcuts are added. The purpose is to reduce the diameter of these shortcuts without causing a significant increase in the degree. One option is to add a set of "evenly spaced" shortcuts, the other option is to add shortcuts in a less regular way. It turns out that it is more efficient to add shortcuts in a less regular manner. For instance, [6] shows an example in which adding only five shortcuts for a 36-vertex ring can reduce the diameter from 18 to 9.

## 3.4 Complex Networks

The usefulness of random or seemingly random shortcuts for complex networks such as social networks and Internet topologies has been noted. The small-world

nature of these networks has received considerable attention in the literature. Watts and Strogatz [15] proposed a small-world network model based on probability parameters, which smoothly transforms a one-dimensional lattice into a random graph, in which a small number of long sides are used to greatly reduce the diameter. In addition, the scale-free and clustering characteristics of complex small-world networks lead to small diameters and average shortest path lengths, as well as robustness to random edge removal.

In order to go beyond traditional topologies in current HPC systems and achive a small-world effect, Michihiro Koibuchi et al. [10] propose an approach that adds random shortcuts to a base topology. In this algorithm, random shortcut topologies are generated by augmenting classical topologies with random links. Specifically, when $d$ shortcuts are added to the node, $k \times d$ random feasible target nodes will be found, where $k \geq 1$. Calculate the length (in hops) of the shortest path to each of these vertices. The $d$ vertices with the longest shortest path are selected as the destination of the shortcut. Continue to find points where the degree does not reach the upper limit and add edges Until the total number of edges reaches the upper limit.

Since $dn \gg m$, the complexity of this algorithm is $O(mk + \frac{mn^3}{d})$.

This algorithm has a certain reference: it can produce a reasonable shortcut addition method in a fast time. But the paper only discusses the application of some specific initial graphs. The structure of these graphs has certain rules, but it is not general. That is, the algorithm may only have some optimizations for specific graphs and constraints. This method will be used as a method to be compared in

the future.

## 3.5 Network-on-Chip (NoC)

The network on chip is a router-based packet switching network between SoC modules. NoC technology applies the theory and methods of computer networking to on-chip communication and brings notable improvements over conventional bus and crossbar communication architectures. The topology of NoC has a profound effect on the overall network cost and performance. It significantly influences the latency and power consumption. It also affects the network traffic distribution, and hence the network bandwidth and performance achieved.[1]

3D NoC Design is an emerging technology that has the potential to achieve high performance with low power consumption for multi-core chips. Sourav Das et al. [7] propose a robust design optimization method to intelligently explore the design space by combining the advantages of small-world (SW) networks and machine learning technologies. This model optimize the placement of both planar and vertical communication links for energy efficiency, which improves the energy efficiency of the 3D NoC architecture.

Routerless NoCs use a network of buses in a sophisticated way and typically need some sort of switching that earlier bus systems do not need[2]. Ting-Ru Lin, Drew Penney et al.[11] proposes a novel deep reinforcement framework, taking routerless NoC as an evaluation case study. The new framework successfully

[1] `https://en.wikipedia.org/wiki/Network_on_a_chip`

resolves problems with prior design approaches, and learns (near-)optimal loop placement for routerless NoCs with various design constraints. A deep neural network is developed using parallel threads that efficiently explore the immense routerless NoC design space with a Monte Carlo search tree.

Specifically, the method proposed by this model is:

- DNN provides a suggestion loop;
- Add subsequent loops by MCTS based on some strategies;
- Use the results to train DNN and MCTS;
- Repeat the above steps until the limit condition is reached;

# Chapter 4: Materials and Methods

## 4.1 Objective

Our goal is to optimize interconnection network topology. That is, for an initial topology, some shortcuts will be added to optimize this structure. In order to simplify this problem and make sure that a feasible solution can be generated in the end, we ensure that the given initial graph is fully connected, and all points in the graph must be connected to each other. That is, there are a total of n points in the graph, and the degree of each point is at most d. On the basis of the initial graph, we can add some edges, so that there are at most m edges in the graph, so that the average hop count and diameter of the final graph can be as small as possible.

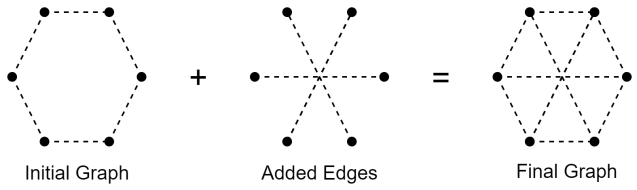


Figure 4.1: Topology Modification Example

## 4.2 Framework Overview

The deep reinforcement learning framework modified according to Routerless NoCs is shown in the Figure 4.2. The framework starts with an empty tree and a neural network with no prior training to initialize Monte Carlo Tree Search (MCTS). The whole process consists of many iterative processes. Each iteration starts with an initial graph (for example, a ring connecting all nodes to ensure that the graph is fully connected), and then continuously implement actions to modify the design. For each iteration, DNN first suggests a good initial action to direct the search to a specific area in the design space; follow the MCTS in that area to take multiple actions. Starting from the current design, MCTS uses a greedy strategy to explore and select actions until it reaches the leaf node (a feasible solution). Finally, the overall reward ("evaluation index") is calculated and combined with information about state, action, and value estimates to train the neural network and update the search tree. Repeat the exploration cycle to optimize the design until the limit conditions (such as the upper limit of the number of edges, etc.) are reached. After the search is complete, we will evaluate the design (calculate the average number of hops and diameter for this graph).

In this framework, DNN generates rough designs, and MCTS effectively refines these designs based on prior knowledge to continuously generate more optimized configurations. Unlike traditional supervised learning, this framework does not require a training data set; on the contrary, DNN and MCTS gradually train themselves from past exploration cycles.

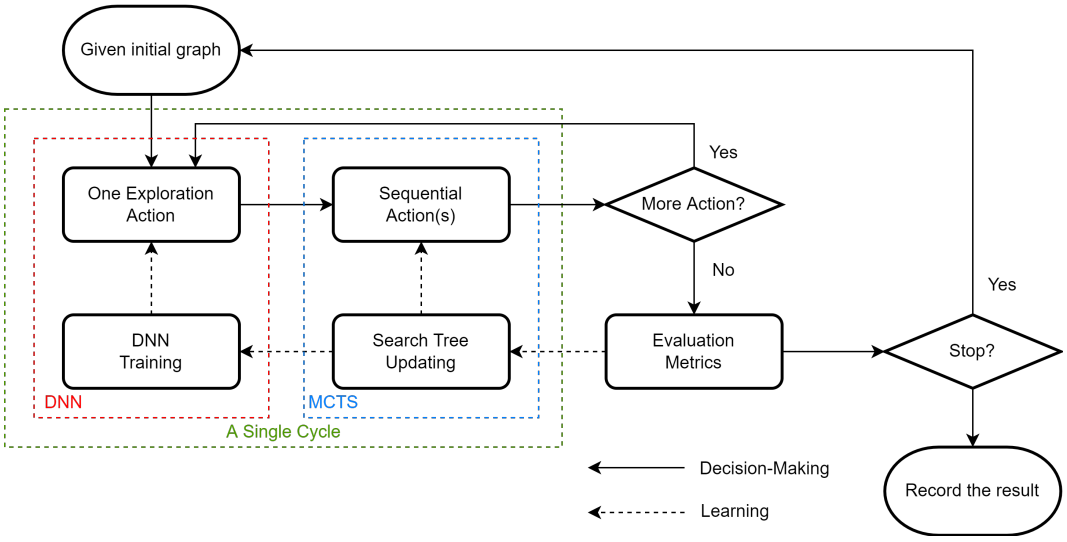


Figure 4.2: Deep reinforcement learning framework

## 4.3 Representation of graphs (States)

State representation in our model uses a hop count matrix to encode current graph state as shown in Figure 4.3. Assuming that there are a total of $n$ nodes in this topology, then the overall state representation is an $n \times n$ matrix. Among them, the element in the $i_{th}$ row and the $j_{th}$ column represents the shortest path length from the $i_{th}$ node to the $j_{th}$ node. In particular, the element in the $i_{th}$ row and the $i_{th}$ column is always 0. Given the initial graph, all nodes are connected, so the initial value of the matrix can be set to $n$.

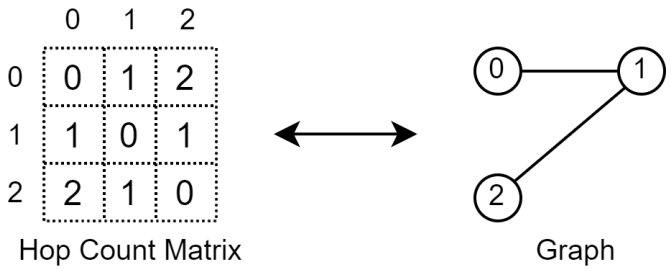


Figure 4.3: Graph Hop Count Matrix Example

## 4.4 Representation of edges (Actions)

Actions are defined as adding an edge to an graph with $n$ nodes. Since an edge connects 2 nodes, the actions can be encoded as:

$$(x, y) \tag{4.1}$$

where $x$ and $y$ represent the sequence number of the node. To avoid ambiguity, we force $x \leq y$ (if $x > y$, exchange them).

## 4.5 Returns After Edge Addition

The reward function encourages exploration by rewarding zero for all valid actions, while penalizing repetitive, invalid, or illegal actions using a negative reward.

- Repetitive action: refers to adding a duplicate edge, receiving a -1 penalty;
- Invalid action: refers to adding a self-loop, receiving a -1 penalty;

- Illegal action: involve additions that violate the degree limit of each node, a penalty of -1 is obtained;

The agent receives a final return to characterize overall performance by subtracting average hop count in the generated topology from average mesh hop count. Minimal average hop count is therefore found by minimizing the magnitude of cumulative returns.

## 4.6 Deep Neural Network

**Residual Neural Networks:** Adequate network depth is essential, but high network depth can lead to overfitting of many standard DNN topologies. Residual networks provide a solution by introducing additional shortcut connections between layers, allowing robust learning even if the network depth is 100 or more layers. The building blocks of the residual network are shown in Figure 4.4. Here, the input is X, and the output after two weighting layers is F(X). Note that both F(X) and X (via shortcut connection) are used as inputs to the activation function. This shortcut connection provides a reference for learning the best weights and alleviates the problem of vanishing gradients in the back propagation process[8]. Figure 4.5 depicts a residual box (Res) composed of two convolution (conv) layers. Here, the numbers 3x3 and 16 represent a 3x3x16 convolution kernel.

**DNN architecture:** The proposed DNN uses the dual-head architecture shown in Figure 4.6, which learns both the strategy function and the value function at the same time. We use convolutional layers because edge placement analysis is

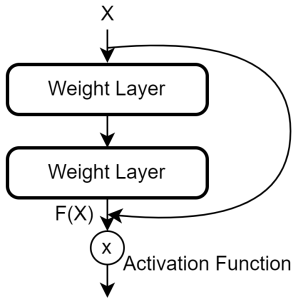


Figure 4.4: A generic building block for residual networks

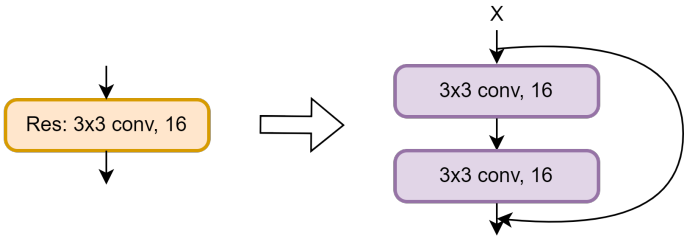


Figure 4.5: A building block for convolutional residual networks

similar to spatial analysis in image segmentation, and it performs well on convolutional neural networks. Use batch normalization after the convolutional layer to normalize the value distribution, and use maximum pooling (denoted as "pooling") after a specific layer to select the most important features. Finally, both the policy and the value estimate are produced in the output as two independent heads. The strategy discussed in Section 4.5 is the action with two dimensions $(x, y)$, which are generated by following the ReLU function. Refer to Figure 4.6, the softmax input after ReLU is $\{a_{ij}\}$ where $i = 1, 2$ and $j = 1, \cdots, N$. Dimensions x and y are $max_j(exp(a_{1j})/\Sigma_j exp(a_{1j}))$ and $max_j(exp(a_{2j})/\Sigma_j exp(a_{2j}))$. The value head

uses a single convolutional layer followed by a fully connected layer without an activation function to predict the cumulative return.

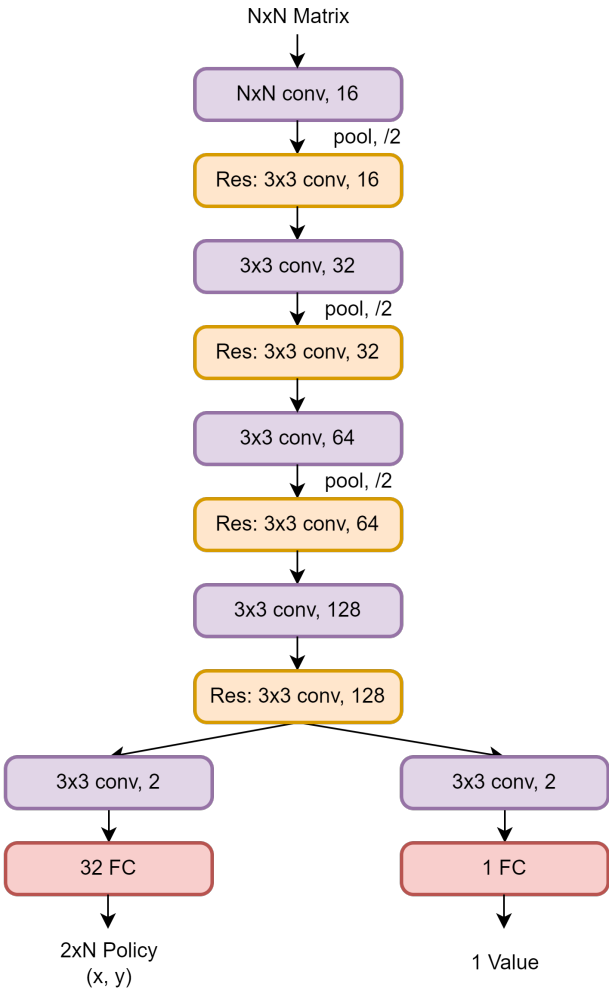


Figure 4.6: Proposed deep neural network

## 4.7 Topology Design Exploration

**MCTS:** In our implementation, each node s in the tree represents a graph (state), and each edge represents an additional edge (action). In addition, each node $s$ stores a set of statistical information: $\overline{V}(s_{next})$, $P(a_i; s)$, and $N(a_i; s)$. $\overline{V}(s_{next})$ is the average cumulative return from $s_{next}$, which is used to approximate the value function $V^{\pi}(s_{next})$. $P(ai; s)$ is the prior probability of taking action $ai$ based on $\pi(a = a_i; s)$. Finally, $N(a_i; s)$ is the access count, which represents the number of times $a_i$ is selected in $s$. The exploration starts from the state $s$, and then the best action $a^*$ is selected according to the expected exploration reward given by the following formula[11]:

$$a^* = \arg\max_{a_i}(U(s, a_i) + \overline{V}(s_{next})) \tag{4.2}$$

$$U(s, a_i) = c * P(a_i; s) * \frac{\sqrt{\sum_j N(a_j; s)}}{1 + N(a_j; s)} \tag{4.3}$$

where $U(s, a_i)$ is the upper confidence limit and $c$ is a constant. The first term in Equation 4.2 encourages extensive exploration, while the second term emphasizes fine-grained development.

The MCTS algorithm shown in Figure 2.4 is divided into 4 stages: selection, expansion, simulation, and back-propagation. In our model:

- Selection: the agent selects the best action (cyclic placement) by following Equation 4.2 with a probability of $1 - \epsilon$ or using a greedy search with a probability of $\epsilon$. Greedy search algorithm evaluates the benefits of adding

various edges and selects the edge with the highest return. Traverse the tree until reaching a leaf node without any child nodes.

- Expansion + Simulation: the leaf state is evaluated using the DNN to determine an action for rollout/expansion. Here, $\pi(a = a_i; s)$ is copied, then later used to update $P(a_i; s)$ in Equation 4.3. A new edge is then created between $s$ and $s_{next}$ where $s_{next}$ represents the routerless NoC after adding the edge to $s$.

- Back-propagation: after calculating the final cumulative gain, propagate the statistics of the traversal edge backward through the tree. Specifically, $\overline{V}(s_{next})$, $P(a_i; s)$, and $N(a_i; s)$ are all updated.

**Greedy algorithm:** During MCTS process, model will use the greedy algorithm to find suitable edges according to the same rules, and then add them to the graph.

When the program starts to execute, the program will generate a search sequence based on the initial graph, that is, sort all the nodes. The sorting rule is: take the sum of the hop count from this node to all other nodes as the first key, sort from big to small; if the first key is the same, the second key is the number of node, from small to big Sort. The larger the sum of the hop count from this node to all other nodes, the greater the distance from this node to other nodes, and the higher priority is to add edges for this node.

During MCTS process, there are two greedy methods.

- For the node currently being searched, find an another node with the longest

path between them, and then add a shortcut between them. Continue to search for the current point until the upper limit of the degree of the point is reached , and then continue to search for the next point in the search sequence. Until the upper limit of the number of edges is reached. The pseudo code of this method is as Algorithm 1:

- Select the two points with the largest shortest path. If the shortest paths are the same, the one with the smallest integer pair is selected.

Combining these two methods can have a good search effect.

**Algorithm 1** Greedy Algorithm used in MCTS

1: $x =$ next available node in search sequence
2: $y = \arg\max_z(\text{HopCount}[x][z])$
3: Add $(x, y)$ to the graph

# Chapter 5: Methodology

This model is implemented by Python 3, and DNN is implemented by TensorFlow. For the parameter setting of program operation:

Table 5.1: Framework Parameter Setting

| Parameter | Value |
|---|---|
| Constant value $c$ in Equation 4.3 | 1.0 |
| Probability of using greedy search | 0.1 |
| Discount factor for rewards | 0.8 |
| Gradient norm clipping | 40.0 |
| Maximum degree of each node | 10 |
| Number of nodes in graph | 64 |
| Maximum number of edges in graph | 128 |

Each time the program runs, the complete feasible solution generated each time will be recorded and the average hop count will be calculated. For those given parameters, after the program runs for a period of time, the average hop count will show a downward convergence trend. After convergence, the corresponding actions and the average hop count can be output.

# Chapter 6: Results

For a known graph, I proposed another random edge addition algorithm with a greedy strategy:

**Algorithm 2** Random edge addition algorithm that I proposed

1: Randomly generate t edges ($t \leq 10$)
2: For each edge, try to add it to the graph, calculate the contribution of this edge to the graph (such as the reduction of the shortest path between two nodes, or reduction of average hop count)
3: Choose the edge that makes the most contribution and add it to the graph
4: Keep adding until the upper limit of the number of added edges is met

If the reduction of the shortest path is used as a contribution, the complexity of this algorithm is $O(m(t+n^2))$. This method will be used as a comparison method later.

As the training process progresses, the average hop count will tend to converge and converge to the local optimal value. Take $n = 64$(number of nodes), $d = 10$(maximum degree of each node), $m = 128$(maximum number of edges) as an example:

By setting different parameters, we compare the test results with random methods (completely random adding edges and random shortcut topologies) to prove that the structure produces better results.

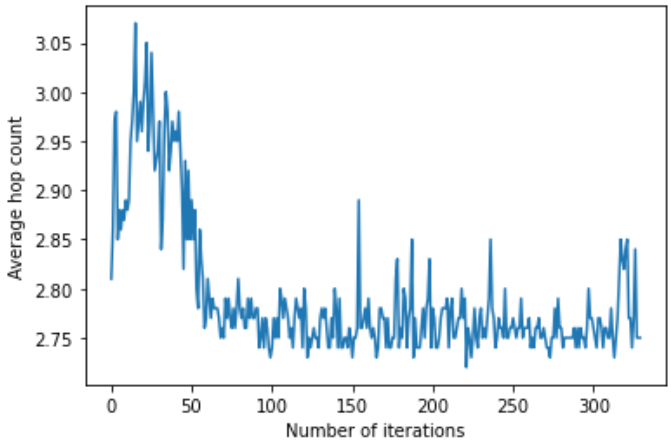


Figure 6.1: Model training process example

## 6.1 Different number of nodes in the graph

In this part, we will fix the initial graph as a ring, the maximum degree is 10, and the maximum number of edges is twice the number of nodes, and test different graph sizes (n). The test results are as follows:

Looking at the diameter again, each line in the graph shows an increasing trend, which is reasonable, because as the size of the graph increases, the internal structure of the graph will become more complicated, and more edges need to be added to reduce the diameter by 1. But we can also find that the diameter of the random shortcut topology algorithm has a very large increase, the full random algorithm and the reinforcement learning model have a very small increase, and the diameter of the reinforcement learning model is smaller than the other two.

Looking at the diameter again, each line in the graph shows an increasing

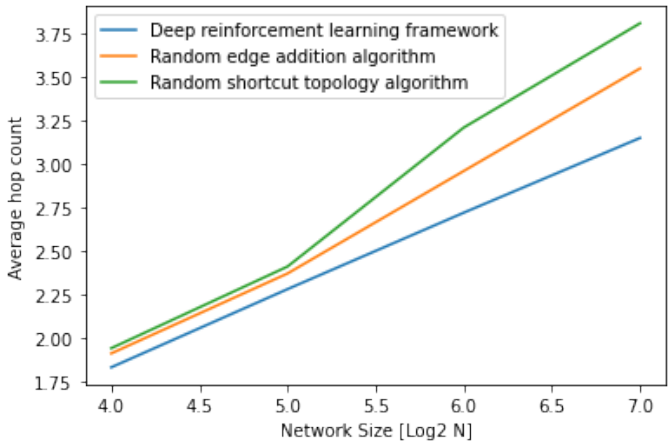


Figure 6.2: Avg hop count vs. network size (ring, d=10, ratio=2)

trend, which is reasonable, because as the size of the graph increases, the internal structure of the graph will become more complicated, and more edges need to be added to reduce the increase in diameter by 1 . But the random edge addition algorithm line completely coincides with our frame, and the diameter of the random shortcut topology algorithm is slightly larger. It can be seen here that compared to the other two algorithms, our algorithm can add edges more effectively and can generate a graph with a smaller diameter and average hop count.

Therefore, as the graph size increases, the superiority of the results produced by the reinforcement learning model will not change.

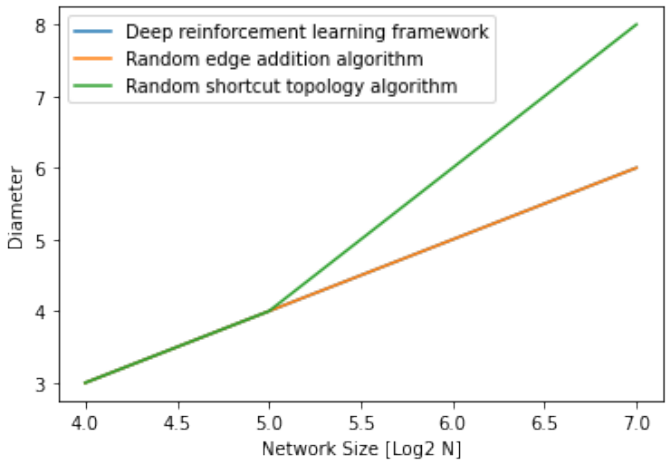


Figure 6.3: Diameter vs. network size (ring, d=10, ratio=2)

## 6.2 Different maximum degree limit of each node

In this part, we will fix the initial graph as a ring, the number of points in the graph is 64, and the maximum number of sides is 128 (2 times the number of points), and test different degrees of each node.

You can see that each line in the graph shows a downward trend. This is reasonable, because as the degree increases, even if the total number of edges does not change, each point is more likely to have a higher degree, that is, it becomes the local center of the graph, so the average hop count will decrease.

It can be seen from the Figure 6.4 that the average hop count of the reinforcement learning model is lower than that of the full random algorithm and the random shortcut topology algorithm, so the increase in degree will not change the betterness of the reinforcement learning model.

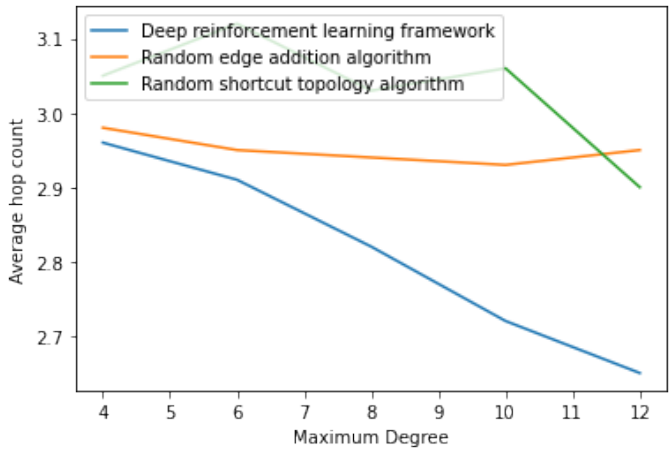


Figure 6.4: Avg hop count vs. maximum degree (ring, n=64, ratio=2)

## 6.3 Different maximum number of edges

In this part, we will fix the initial graph as a ring, the number of points in the graph is 64, the maximum degree is 10, and the different maximum number of edges are tested. As the graph size increases, the edges that need to be added also need to increase. So simple consideration, we assume that the maximum number of edges and the size of the graph are linearly positive, then we adjust the ratio in this comparison.

All the lines in the graph show a decreasing trend, and the decreasing range gradually becomes smaller. This is reasonable, because as the number of edges increases, the graph will be closer to a fully connected state, that is, the average hop count is 1. It can be seen from the Figure 6.5 that the superiority of the results produced by the reinforcement learning model will still not change with the increase of the maximum number of edges.

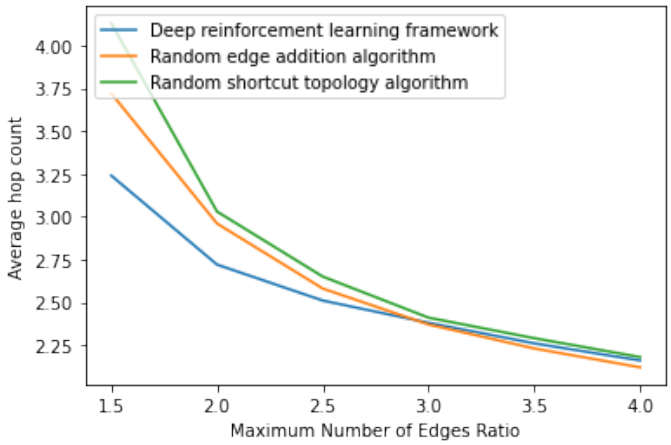


Figure 6.5: Avg hop count vs. maximum number of edges ratio (ring, n=64, d=10)

## 6.4 Selection of the initial graph

In this part, we will fix the number of points in the graph to 64, the maximum degree to 10, and the maximum number of sides to be 128 (two times the number of points) to test the effect of different initial graphs on the results. Because HPC system topologies generally have symmetry, rings and full binary trees are selected for testing here.

The optimization effect of the reinforcement learning algorithm on the ring is obvious, but the optimization effect on the binary tree is relatively small. These two graphs have certain extremes: when the same number of edges are added, the average shortest distance of the ring is the largest (linearly related to n), and the binary tree has several identical substructures, but the average shortest distance is shorter (with logn Linear correlation). Therefore, different initial graphs will not affect the betterness of the reinforcement learning model.

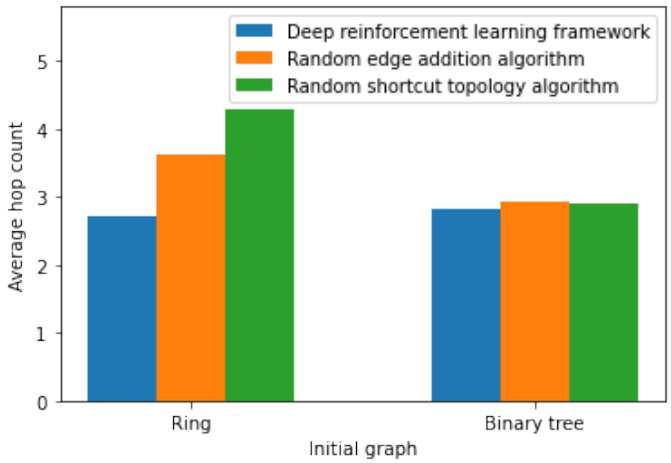


Figure 6.6: Avg hop count vs. different initial graph (ring, n=64, d=10, m=128)

## 6.5 Generated Graph Example

In this section some generated graphs will be shown. We choose the network size $n = 16$, the maximum degree of each node $d = 10$, and the maximum number of edges $m = 32$.

In addition, for the above restrictions, we found that 4*4 2D Torus and 4-Hypercube also meet the requirements.

Since topology generated by our framework is similar to the random shortcut topology, the edge distribution is not regular, the topology is not simple. Therefore, it is necessary to use source routing or distributed routing of the routing table.

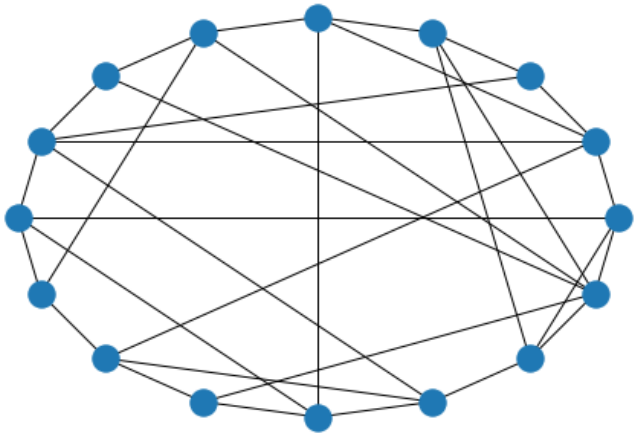

Figure 6.7: Generated Graph Example (DRL Framework)

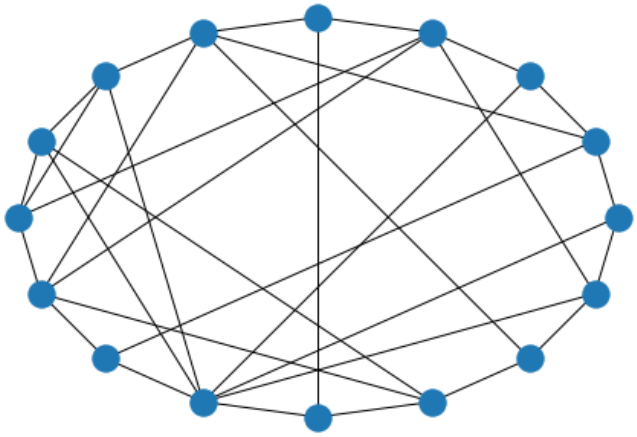

Figure 6.8: Generated Graph Example (Random Edge Addition Algorithm)

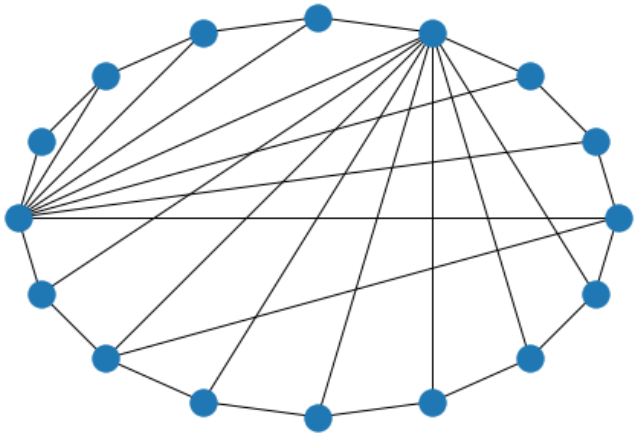

Figure 6.9: Generated Graph Example (Random Shortcut Topology Algorithm)

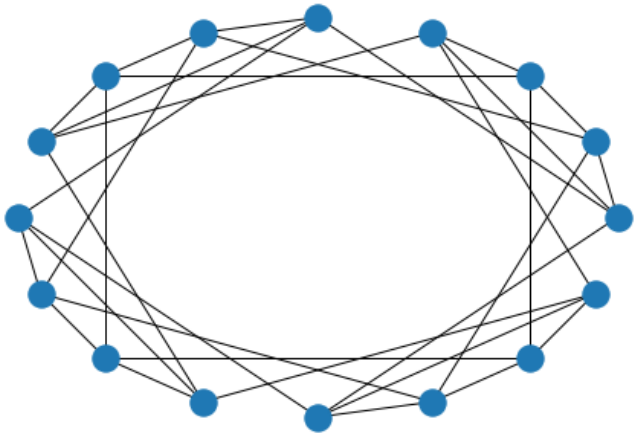

Figure 6.10: Generated Graph Example (4*4 2D Torus)

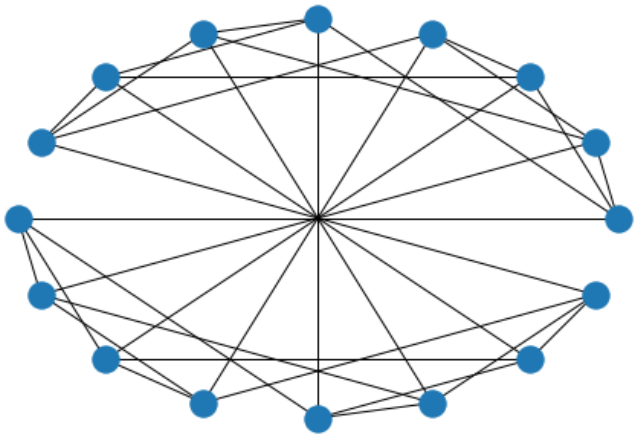

Figure 6.11: Generated Graph Example (4-Hypercube)

# Chapter 7: Future Work

## 7.1 Universality

Through the discussion of the results, the results produced by the reinforcement learning model are better than the results of the random algorithm in all dimensions. Therefore, the reinforcement learning model can play a good role in the graph optimization problem. This model can replace the initial graph at will. The random shortcut topology algorithm test uses a relatively common and regular initial graph, most of which are symmetrical. This model can be replaced at will, and it can be replaced with an irregular and asymmetrical initial graph. Even if it is replaced, reinforcement learning will learn from it and help design space exploration. In this way, the problem that this paper hopes to solve can be upgraded from HPC system topology optimization to a general graph optimization problem.

## 7.2 Modification of restrictions

This model can also make certain modifications to the restriction conditions. The restriction conditions set during the test in this paper are: the upper limit of the degree of each point, and the maximum number of edges to be added.

### 7.2.1 If the edge has weight

When this model is tested, it is assumed that the added cost of each edge is the same, which is 1. In the actual HPC system topology, there may be a certain distance between two machines. When we need to add a data cable between the two machines, the added cost is not necessarily the same (for example, the cost of the short distance is small, and the cost of the long distance is large). In order to solve this problem, we can:

- Modify the cost of each edge from 1 to actual cost
- Set the upper limit of the total cost
- Consider the optimization effect of the cost when adding the edge

and so on.

### 7.2.2 Each machine has a flow limit

When testing this model, it is assumed that the maximum degrees of all points are the same. In an actual HPC system, the processing capabilities and interface connection capabilities of each machine may be different. The number of ports of each machine may also be different. In order to solve this problem, we can set different degree limits for different machines.

### 7.2.3 Model training according to different needs

Consider the problems encountered by the HPC system in the actual process: we hope to optimize not only the HPC system topology, but the overall HPC system operating efficiency. Assuming that we can obtain historical data of the HPC system running for a period of time (such as the amount of communication in each edge, etc.), we can use these historical data as a training set to help choose among candidate edges in each iteration. And assess the quality of the topology. The topology obtained through such training fits the actual use situation. For different HPC computer rooms, we can train to obtain different topological structures, and achieve good use effects for different usage scenarios.

# Chapter 8: Conclusion

Topology exists widely in HPC systems. Since the delay is relatively large compared to the line and microchip injection delay, structural optimization of interconnection topology is very important for HPC systems. There are many ways to generate topologies. One option is to generate new topologies, but this method is relatively mature, almost as a bottleneck; our work is to add shortcuts based on an existing topology, this method is very promising but is also very challenging. We proposed a model combines DRL and MCTS, and can produce very good results, which have lower average hop count and diameter, compared with random shortcut topology algorithm and random edge addition algorithm that I proposed. In addition, our model can also solve more problems, such as general graph optimization problems, or when edges are weighted. We can solve those problems by modifying model.

# Bibliography

[1] Sheldon B Akers. The star graph: An attractive alternative to the n-cube. In *Proc. Int'l Conf. Parallel Processing., 1987*, 1987.

[2] Fawaz Alazemi, Arash Azizimazreah, Bella Bose, and Lizhong Chen. Routerless network-on-chip. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 492–503. IEEE, 2018.

[3] George Almási, Ralph Bellofatto, José Brunheroto, Călin Caşcaval, José G Castanos, Luis Ceze, Paul Crumley, C Christopher Erway, Joseph Gagliano, Derek Lieber, et al. An overview of the blue gene/l system software organization. In *European Conference on Parallel Processing*, pages 543–555. Springer, 2003.

[4] Mehdi Baboli, Nasir Shaikh Husin, and Muhammad Nadzir Marsono. A comprehensive evaluation of direct and indirect network-on-chip topologies. In *Proceedings of the 2014 international conference on industrial engineering and operations management*, pages 2081–2090, 2014.

[5] Eiichi Bannai and Tatsuro Ito. On finite moore graphs. *J. Fac. Sci. Tokyo Univ*, 20(191-208):80, 1973.

[6] Jean-Claude Bermond, Francesc Comellas, and D. Frank Hsu. Distributed loop computer-networks: a survey. *Journal of parallel and distributed computing*, 24(1):2–10, 1995.

[7] Sourav Das, Janardhan Rao Doppa, Dae Hyun Kim, Partha Pratim Pande, and Krishnendu Chakrabarty. Optimizing 3d noc design for energy efficiency: A machine learning approach. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 705–712. IEEE, 2015.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[9] John Kim, William J Dally, and Dennis Abts. Flattened butterfly: a cost-efficient topology for high-radix networks. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 126–137, 2007.

[10] Michihiro Koibuchi, Hiroki Matsutani, Hideharu Amano, D Frank Hsu, and Henri Casanova. A case for random shortcut topologies for hpc interconnects. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 177–188. IEEE, 2012.

[11] Ting-Ru Lin, Drew Penney, Massoud Pedram, and Lizhong Chen. A deep reinforcement learning framework for architectural exploration: A routerless noc case study. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 99–110. IEEE, 2020.

[12] Maheswara R. Samatham and Dhiraj K. Pradhan. The de bruijn multiprocessor network: a versatile parallel processing and sorting network for vlsi. *IEEE Transactions on Computers*, 38(4):567–581, 1989.

[13] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[14] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[15] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442, 1998.

# APPENDICES