Using Reinforcement Learning to Improve Heuristic Sparse Matrix Algorithms

Thesis submitted in partial fulfillment

of the requirements for the degree of

Master of Science

in

Computer Science and Engineering by Research

by

Arpan Dasgupta

2018111010

`arpan.dasgupta@research.iiit.ac.in`

International Institute of Information Technology, Hyderabad

(Deemed to be University)

Hyderabad - 500 032, INDIA

December 2023

International Institute of Information Technology

Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled " **Using Reinforcement Learning to Improve Heuristic Sparse Matrix Algorithms** " by **Arpan Dasgupta**, has been carried out under my supervision and is not submitted elsewhere for a degree.

Date

Adviser: Dr. Pawan Kumar

To *harmony*

Acknowledgments

I would like to express my heartfelt gratitude and appreciation to my family, especially my parents, for their unwavering support, love, and encouragement throughout my academic journey. Their constant belief in my abilities and their sacrifices have been the driving force behind my accomplishments. I am forever grateful for their guidance, understanding, and the countless hours they devoted to nurturing my dreams.

I would like to extend my deepest appreciation to my advisor, Prof. Pawan Kumar, for his invaluable guidance, patience, and expertise. His insightful feedback, unwavering support, and dedication to my success have been instrumental in shaping the direction of my research. I am truly fortunate to have had the opportunity to work under his mentorship, and I am grateful for the wisdom and knowledge he has shared with me throughout this thesis journey.

I would also like to acknowledge and thank my peers at my university. Their intellectual camaraderie, stimulating discussions, and shared experiences have significantly enriched my research and personal growth. Their diverse perspectives and collaborative spirit have fostered a vibrant academic environment that has been crucial to my development as a scholar. I would especially like to thank Animesh, Jai and Joseph for discussions regarding my project which provided essential ideas for my work. I am grateful for the friendships forged and the unwavering support I have received from my fellow students.

This thesis represents the culmination of years of hard work, and I am deeply appreciative of everyone who has played a role in my academic and personal development. Thank you all for your unwavering belief in me.

Abstract

A large number of important mathematical problems do not have perfect solutions which can be computed in a polynomial time. The solution to these problems is often calculated by using polynomial time heuristic algorithms which produce solve the problem with some approximation. If the matrix algorithm can be effectively formulated as a game, reinforcement learning can be used instead of the usual heuristics to improve the quality of the approximation. One such problem is to reduce the fill-in produced during matrix decomposition. A large number of computational and scientific methods commonly require decomposing a sparse matrix into triangular factors as LU decomposition. A common problem that is faced during this decomposition is that even though the given matrix may be very sparse, the decomposition may lead to a denser triangular factors due to fill-in. A significant fill-in may lead to prohibitively larger computational costs and memory requirement during decomposition as well as during the solve phase. To this end, several heuristic sparse matrix reordering methods have been proposed to reduce fill-in before the decomposition. However, finding an optimal reordering algorithm that leads to minimal fill-in during such decomposition is known to be a NP-hard problem. A reinforcement learning based approach is proposed for this problem. The sparse matrix reordering problem is formulated as a single player game. More specifically, Monte-Carlo tree search in combination with a neural network is used as a decision making algorithm to search for the best move in our game. The proposed method, Alpha Elimination was found to produce significantly lesser non-zeros in the LU decomposition as compared to existing state-of-the-art heuristic algorithms with little to no increase in overall running time of the algorithm. In addition to the main problem of the thesis, work was also conducted in the field of extreme classification. In this area, a fast algorithm called LightDXML was developed to solve the problem of extreme classification at scale.

Contents

List of Figures

List of Tables

Chapter 1

Sparse Matrix Decomposition

1.1 Motivation

1.1.1 Matrices

Matrix is an array of numbers or symbols arranged into rows and columns. Matrices are an important way to store tabular data and parameters. Matrix computations form a large amount of modern computing due to the boom in machine learning and data science. This makes it a very important field for research considering that every minor improvement makes a great difference due to the scale at which the operations are performed. As a result matrix operations are optimized to the fullest and algorithms on better matrix computations are essential.

1.1.2 Sparse Matrices

Sparse matrices are an essential extension to matrices as they require much lesser memory to store. Roughly, sparse matrices are defined as matrices which have a relatively larger number of zeros compared to non-zeros. Sparse matrices can appear very frequently in several applications such as computational fluid dynamics, chemical process simulations, and structured graph problems. The presence of a smaller number of non-zeros implies that compute intensive operations which are intractable on matrices in general can become tractable by using algorithms which utilize the sparsity [35]. The sparsity level of the matrix can also be quantified as the total number of zero elements divided by the total number of elements in the matrix.

1.1.3 Matrix Decomposition

A large number of applications of matrices require decomposition. Matrix decomposition in linear algebra refers to the factorization of a matrix into a product of multiple matrices; examples include LU, QR, SVD, etc. [14]. Matrix decomposition algorithms like LU mainly rely on algorithms like Gaussian Elimination. In Gaussian Elimination, the goal is to perform elementary row operations on the provided matrix to obtain an upper triangular matrix U. All the elements in the lower triangle are eliminated and the corresponding lower triangular matrix L can be obtained from the row operations. Thus the final decomposition is the LU factorization, $A = LU$.

1.1.4 Challenges with LU of Sparse Matrices

The LU decomposition is useful in solving systems of linear equations stemming from numerous and wide variety of applications ranging from 3D reconstruction in vision, fluid dynamics, electromagnetic simulations, material science models, regression in machine learning and in a variety of optimization solvers that model numerous applications. Thus it is of essence that the L and U matrices after the decomposition of the sparse matrix are also sparse. For sparse matrices, LU decomposition also presents a significant challenge, because the resulting matrices from the decomposition might not maintain their sparsity. This happens due to fill-in that occurs when using a row with a large number of non-zeros as a pivot, leading to creation of non-zeros in the subsequent rows. [10] shows this with an example in 1.1.

1.1.5 Methods to reduce fill-in

The above described process of elimination causes fill-in due to the wrong row being chosen for elimination. One way to solve this problem is to permute the matrix before elimination. Formally, we choose a row permutation matrix P_{row}, which is pre-multiplied to the original matrix before performing Gaussian Elimination. The inverse of this matrix can be multiplied back after decomposition. This procedure can lead to reduced sparsity if the P_{row} is chosen correctly. However, it can be shown that choosing the ideal permutation matrix can be an NP-hard problem by converting it to a hypergraph partitioning problem [7]. We aim to solve this problem to the best approximation using Reinforcement Learning.

Previously, this problem has been solved by several of the heuristic methods which are used to solve the problem of fill-in reduction. A large number of these methods solve the problem by imagining

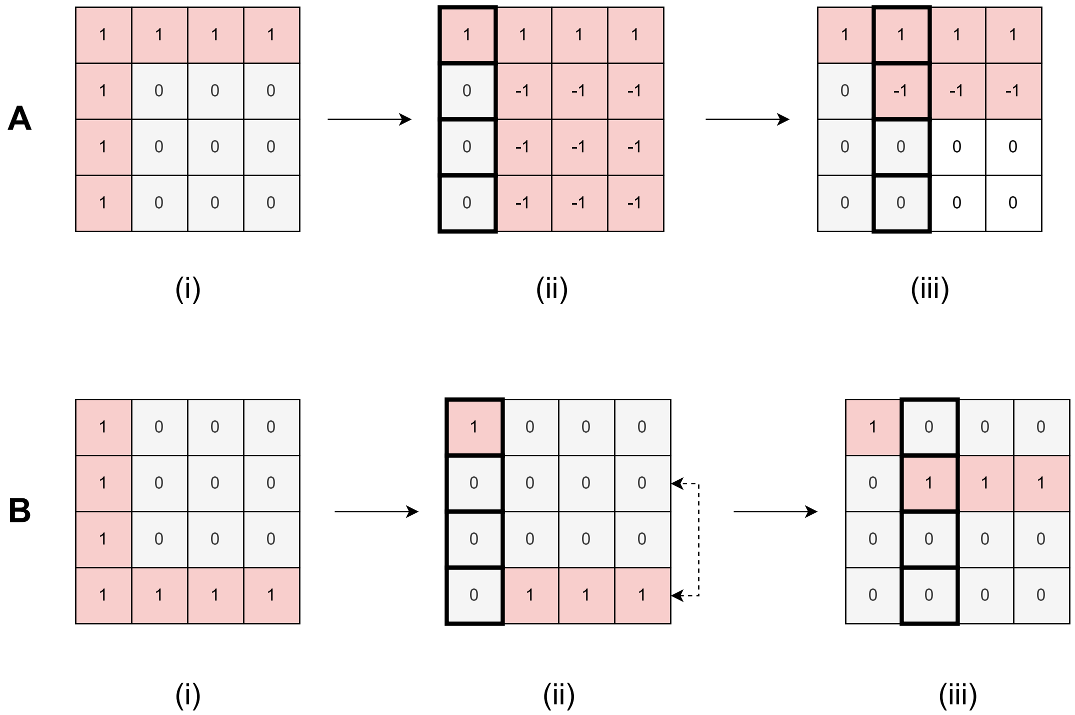


Figure 1.1: Toy example demonstrating the effect of fill-in during LU decomposition on different permutations of the same matrix. The non-zero entries are denoted by red color and the column being eliminated is highlighted at each step. (A) The first column being eliminated causes all the other columns to become non-zeros, only some of which are eliminated in the subsequent steps. We end up with 7 non-zero entries in the U matrix. (B) Using a different row permutation for the matrix, we obtain no fill-in, and end up with 4 non-zeros in the U matrix. While both of the obtained matrices are valid U matrices, rearranging in B, produces a sparser matrix.

the sparse matrix as a graph. The algorithm Approximate Minimum Degree (AMD) [1] permutes row and columns of the matrix by choosing the node at each step which has the minimum degree in the remaining graph. Column Approximate Minimum Degree (ColAMD) [11] performs approximate column permutations and uses a better heuristic than AMD. SymAMD is a derivative of ColAMD which is used on symmetric matrices. Sparse Reverse Cuthill-McKee (SymRCM) [25] is another method that is commonly used for ordering in symmetric matrices which uses the reverse of the ordering produced by the Cuthill-McKee algorithm [25] which is a also graph based algorithm for permuting the graph based on breadth-first search.

1.2 Contribution

In summary, our key contributions are :

1. We formulate the problem of finding the row permutation matrix for reduced fill-in as a single player game complete with state, action and rewards. We then use MCTS to find a solution to the game.

2. We develop an efficient method to find row permutations for a matrix to reduce the fill-in during LU decomposition.

3. We perform extensive experiments on matrices from several real-world domains to show that our method performs better than the naive LU without reordering as well as existing heuristic reordering methods.

4. We discuss how to provide sparse matrices as input to neural networks. As far as we know, there has been not enough work in this domain for sparse matrices, where the matrix structure is changing in each input. While a graph neural network can take sparse matrix as input, it does not allow for the graph structure to change with each instance, which is what we require here.

5. We discuss how to to extend our algorithm to matrices of larger dimensions seamlessly. A method is then suggested to extend our work to be applicable on sparse matrices of sizes up to $10^6 \times 10^6$.

6. Our work also opens the door to discussions about replacing heuristic based methods in other mathematical applications with reinforcement learning.

1.3 Thesis Organization

The remaining thesis is organized as follows :

- In **Chapter** 2 **: Reinforcement Learning for Algorithms**, we have briefly reviewed previous work on applications of Reinforcement Learning for algorithm discovery and meta-learning.

- **Chapter** 3 **: Alpha Elimination** illustrates the method developed to solve the above described problem of reducing fill-in. We talk about the different methods considered and the final method which was obtained.

- **Chapter** 4 **: Experimentation and Results** describes in what setting the experimentation was conducted. The types of experiments performed to show the working of the method and the results which were obtained, along with ablation study is then discussed.
- **Chapter** 5 **: Other Work** discusses in brief another direction of work which was done involving extreme classification. The section goes into the detail of a proposed method called LightDXML which was developed to allow fast and better deep learning models for extreme classification.
- **Chapter** 6 **: Conclusion and Future Work** talks about how the RL method can be extended to solving other mathematical problems and what improvements can be made to it.

Chapter 2

Reinforcement Learning for Algorithms

2.1 Deep Reinforcement Learning

Reinforcement Learning (RL) is a sub-field of machine learning which deals with agents that interact with the environment and learn to perform specific tasks by maximizing a cumulative reward. Generally, a Markov Decision Process (MDP) is used to model an RL problem. An MDP is denoted by a set of values $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$ where $\mathcal{S}$ represents states of the environment, $\mathcal{A}$ represents the set of possible actions that can be taken, $\mathcal{P}$ represents the probability of transition from one state to another given action, and $\mathcal{R}$ represents the reward function given a state and action pair.

RL tries to learn a policy $\pi : \mathcal{S} \to \mathcal{A}$ which maximizes the cumulative reward in the environment. This policy decides on what action is to be taken given a certain state [42].

When the action space is and the state space is general (discrete or continuous), reinforcement Learning can be performed using different methods such as actor-critic [21] [41] algorithms or policy-based methods [24].

When the action and state space is discrete, the problem becomes easier to formulate as we do not necessarily have to map the action and state space explicitly using a function. To this end, the problem can be solved by methods such as Monte Carlo Tree Seach (MCTS) [5], [15] or Deep Q-Learning [46]. These methods have previously proven to be successful in a variety of domains such as playing games like Go [38], Chess and Shogi [39]. Applications have also been found in domains like qubit routing [40] and planning [32] where the problems can be formulated as a single-player game. With the advent of deep learning, several of these methods are combined with neural networks as function approximators which highly improves their performance.

2.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a tree search mechanism which aims to traverse a tree by using an estimated value based on random trials.

2.2.1 Tree Search Methods

The biggest advantage of using a tree search method to navigate complex spaces is to make the search more systematic. There are several ways to perform a tree search and several of them are useful in separate contexts. The general pattern of a tree search algorithm can be described in a simple algorithm with a heap.

The algorithm involves maintaining a heap which consists a set of candidate nodes to choose from. At each time step, the heap is used to select the node with the least value. The nodes are added to the heaps when an unvisited node is visited by its neighbor and the value function is calculated too. What differentiates the different search algorithms is how the value function of the nodes is computed.

1. **DFS -** The value function of the node is chosen as the negative of the time step it was first visited at.

2. **BFS -** The value function of the node chosen as the time step it was first visited at.

3. **A* Search -** The value function is decided by the estimated value of the node usng a random heiristic.

4. **Monte Carlo Tree Search (MCTS) -** The value function is chosen to be an approximate value which is obtained by the average cost obtained from several trial runs.

2.2.2 Basic Idea of MCTS

MCTS works by the idea that given enough number of trial runs from a point, the approximate value of the expected reward from the node can be calculated. This can prove to be useful due to the fact that given a large state space, it becomes impossible to calculate the exact value of expected reward from a state. Hence, MCTS provides a way to effectively speed up search on large state spaces.

There are four steps to the MCTS algorithm :

1. **Selection :** The selection step involves starting at the root of the tree and then selecting nodes consecutively until a node is reached which has a child that has not yet been explored or the game ends. The selection procedure can be done arbitrarily but it is better to do it in a way that more interesting paths are explored. This is where Upper Confidence Bound (UCB) is used to improve the search mechanism.

2. **Expansion :** Once a node is selected in the tree, one of the children is chosen and if this child was previously unexplored, then the node is initialized.

3. **Rollout/Simulation :** From the chosen child node, rollouts are initiated. The game is played out from the current state until the end of the game is reached. The reward accumulated from this play is calculated, and noted down. The simulation of the game can be done using a random move selector, or by using any other estimate that tells us about which games are interesting. For example, neural networks can be used to guide the rollout step.

4. **Backpropagation :** The final reward obtained for the rollouts are used to update the values for all of the nodes from the child which was chosen, to the root. This step involves updating all the parameters such as total reward, number of trials, neural network estimates, etc.

The figure 2.1 shows the discussed stages of MCTS.

2.2.3 AlphaGo

In the paper [38], the authors at DeepMind use the technique of MCTS to develop a method called AlphaGo which can play the game of Go with good results.

Go is a game that was first invented in China several thousand years ago. The game involves placing stones on a 19×19 board (there are other possible sizes) with an intention to have the largest number of stones at the end of the game. The stones change color if they are completely surrounded by stones of the opposite color. The game ends when neither player is able to place stones any longer (there are other alternate game ending scenarios).

Go is a very complicated game due to the fact that the number of possible states in the game exceeds that of the number of atoms in the known universe. This level of complexity had made it a significant challenge to develop an computer algorithm for Go. It was clear that it is impossible to brute force the

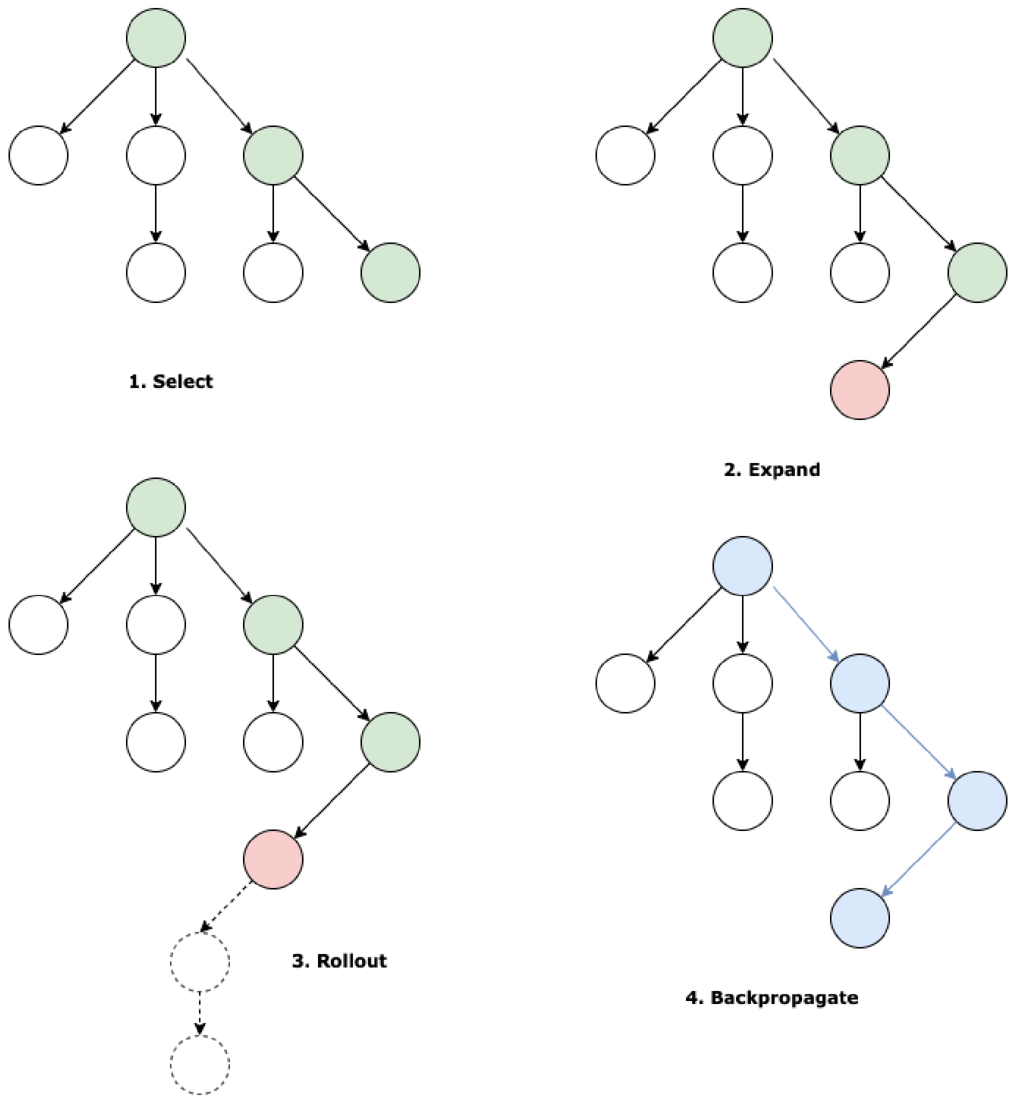


Figure 2.1: Diagram showing the various steps in Monte Carlo Tree Search.

search for the best moves and an method had to be developed which was able to inherently figure out the better moves to play in a certain situation like a human.

MCTS had been used in combination with neural networks before, but there were several issues yet to be solved that caused the method to be less effective than expected. AlphaGo was able to show how to effectively utilize Deep Neural Networks with MCTS on two-player games.

AlphaGo used Convolutional Neural Networks (CNNs) to analyze the board state. In technical terms, the neural network acts as a value and policy estimator. The value estimator acts as a estimate of the expected reward from the current state. The value estimate is instrumental is deciding if a certain path is worth playing out further. The neural network also provides a policy estimate, which is used to figure out which action potentially leads to the highest reward.

These networks are trained using the data collected from the rollout in MCTS, as well as data from existing games. The process of feeding existing data from human games to train RL algorithms is called bootstrapping. AlphaGo was initialized with data from recorded games of expert human players with over 30 million moves. This allowed it to be able to mimic human play first and then improve on these moves using MCTS.

Training of these networks is also not a trivial task, as the neural networks assume data independence while training, which cannot be achieved trivially as the runs are not random. Thus, a data structure called a **replay buffer** is used which randomly feeds moves to train from to the neural network instead of sequentially using moves from the games. AlphaGo also suggests an improvement to replay buffer called **Prioritized Experience Replay (PER)** which returns training data according to their importance.

AlphaGo defeated the reigning world champion Lee Sedol in a 5-match series in 2016. It was an important moment from the perspective of both AI and Go communities. It was considered to be a significant step in the development of AI algorithms. The work done was considered revolutionary not only due to the fact that it performed better than humans, but was also able to discover new strategies that were previously not thought of. The Go community also considered the algorithms as revolutionary due to the new styles of play which it came up with. Most notoriously, a move dubbed as ’move 37’ in a game by AlphaGo against Lee looked like a mistake to onlookers but worked counter-intuitively leading to the victory of AlphaGo.

Post the 2016 victory, DeepMind has developed better algorithms for playing Go. [39] develops AlphaZero, which is able to outperform AlphaGo by a margin. AlphaZero is very impressive due to the fact that it is trained with zero bootstrapping. Thus, the agent has to learn to play the game completely by playing against itself with the only initial data provided being the rules of the game.

2.2.4 Extension of AlphaFold using MCTS

The success of AlphaZero in playing games inspired a series of attempts at using the same method at solving different problems. An unexpected application of this came in biology in the form of protein structure prediction.

Proteins are important building blocks in living organisms which lead to growth, repair, and development of different body parts among other things. Proteins are made up of a long chain of building blocks called amino acids. However, the protein is defined by more than just the composition of amino acids. The properties of the protein depends on the folding of the chain which is only visible in the 3D structure. Scientists have only been able to study a small fraction of proteins due to this constraint.

DeepMind came up with the method AlphaFold [17] which can predict protein structure given just the sequence of amino acids which constitute it. The method uses a set of transformer encoders to do this, which are trained on a set of 190k known structures. It is able to correctly predict the structure of smaller chain proteins. However, the accuracy decreases significantly with larger chains.

The work done in [6] extends the capabilities of AlphaFold by using Monte Carlo Tree Search. The idea is to construct the structure of larger proteins by combining dimers (2-sized chains). The paper uses the standard MCTS approach of finding which dimer structure to add to the previously created chain. The reward function is decided by using a scoring mechanism which can predict if a certain structure is complete or not. This method was shown to significantly improve prediction on large protein structures compared to existing methods.

2.3 Application in Algorithm Finding and Mathematics

While RL has become a go-to standard for creating agents that can play games [38], it is clear that the scope of Reinforcement Learning is much wider than just playing simple games. Whenever a problem can be framed as a single or multi-player game, RL can help find a suitable policy given that the problem can be formulated properly.

2.3.1 AlphaTensor

The first big breakthrough which made it clear that reinforcement learning could contribute very significantly to algorithm discovery was when the method AlphaTensor [13] was proposed by DeepMind. The paper proposed an RL agent which could discover methods to perform faster matrix multiplications on larger matrix dimensions.

The method works by converting the original problem of finding a faster matrix multiplication into a problem of finding a tensor decomposition with the least rank. Any matrix multiplication can be represented as a summation of three-dimensional tensors with unit rank. Each rank-1 tensor represents a single multiplication and the end goal is to find the smallest set of rank-1 decompositions which when summed up can be used to represent the complete matrix multiplication.

Suppose we want to multiply the matrices A and B to obtain matrix C where the matrices are of size n. The original matrix tensor is defined to be a tensor T_n of dimension 3 (for 2-dimensional matrix multiplication) which is composed of 0's or 1's and has $T_n(i, j, k) = 1$ when the simplest form of the matrix multiplication has $a_i \times b_j$ as a term added to c_k. where a, b, and c are the unrolled form of matrices A, B , and C. The figure 2.2 shows the original matrix tensor for size 2, ie. T_2.

A rank-1 tensor is a tensor that can be represented by a multiplication of three single-rank tensors. Here, a rank 1 tensor is a multiplication of three tensors u, v, and w with integer values. Each of the above tensors has the same size as a, b, and c and gives a particular coefficient to the element they represent. In the case of 2×2 matrix multiplication, the single-rank tensors look like this :

$$\mathbf{t} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ -1 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \\ 0 \\ -1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

The original matrix multiplication tensor must then be a sum of several such rank one tensors.

$$\mathbf{T_n} = \Sigma \mathbf{t}_r = \Sigma(\mathbf{u}_r \otimes \mathbf{v}_r \otimes \mathbf{w}_r)$$

The problem is formulated as a game where an initial state is defined by $S_0 = T_n$ and at each step the method chooses to subtract a value $\mathbf{u}_t \otimes \mathbf{v}_t \otimes \mathbf{w}_t$ from the state with a goal to achieve zero tensors at some time t, i. e. $S_T = 0$. The reward for the agent is given at the end when the agent reaches a final state by either reaching zero tensors or by exceeding the number of steps. The reward function can be adjusted to allow for different optimization criteria such as time taken for matrix multiplication.

The algorithm was shown to find a state-of-the-art algorithm for matrix multiplication by beating the previous matrix multiplication algorithms for several sizes in terms of requiring a lesser number of matrix multiplications. Further, it was shown to be able to find matrix multiplication algorithms that better suited specific hardware by incorporating actual time values into the reward.

2.3.2 AlphaDev

Sorting algorithms are one of the most basic areas where humans have worked for decades to improve performance. However, except for minor progress in small specific problems, the performance of the best algorithms has hit a ceiling. Recently, a method to find faster sorting algorithms has been developed in [27] which starts by using something which most humans do not work with when developing new algorithms, assembly.

AlphaDev works on the idea is that even though there might be a lot of things that humans got correct in terms of coding the usual algorithms, there are still a large number of optimizations at the low level which are not easily visible to humans. AlphaDev works directly with assembly which provides greater flexibility compared to several higher level programming languages. Sorting is then converted into a single player assembly game.

The game is constructed by asking the agent to build an algorithm by sequentially choosing an assembly instruction from a set of possible instructions. The algorithm must then be able to run on a set of input sequences and produce a correct output sequence (the sorted array of numbers). The agent is rewarded according to both the correctness of the sorting algorithm and the time taken to perform this sorting. The final algorithm is output by using MCTS guided by Deep Neural Networks.

At a certain point in the game, the current state $S_t = \langle P_t, Z_t \rangle$ is depicted by using two parts, P_t which is the currently generated code and Z_t which is the current state of the registers. The state of the game was input to a neural network by using transformer encoders.

AlphaDev was shown to produce excellent results by producing code which can sort up to 70% faster for shorter sequences. Since smaller sorting algorithm sequences are often a part of larger sorting algorithms, these sorting algorithms can be called as subprocesses leading to significant reduction in time taken to sort larger arrays. The corresponding sorting algorithms were also been incorporated into the standard sort of the open-source library *llvm*.

AlphaDev can be similarly also applied to try and find better hashing algorithm that minimize hashing time as well as collisions. It can also be used to find smaller low-level optimizations in code for existing algorithms.

2.3.3 Meta Learning using RL

Meta-learning is also a very common way to utilize the ability of RL to explore large search spaces. Meta-learning [36] refers to the usage of learning and optimization methods to find optimal hyperparameters and structures in other learning algorithms. In meta-learning, the focus is not solely on learning from a specific dataset but on acquiring knowledge or learning strategies that can be applied to new, unseen tasks or domains. Meta-learning can involve training a meta-learner or meta-model on the distribution of learning tasks. The meta-learner learns to extract and generalize useful information or patterns from the training tasks, such as identifying relevant features, adapting model architectures, selecting appropriate learning algorithms, or optimizing hyperparameters. This learned knowledge can then be used to quickly adapt or generalize to new, unseen tasks with limited data.

There has been previous work in meta-learning such as learning update rules in neural networks [4], RNNs which can modify themselves [37], and few-shot learning. There has been progress on trying to formulate learning RL agents as an RL problem by unrolling LSTMs across an agent's trajectory [12], [45].

Recently, [33] uses Reinforcement Learning for finding better update rules and value functions for RL algorithms. The current update rules such as TD-Learning [43] and the concept of value functions have been a result of research in the past few decades. However, there are potentially better alternatives to these existing solutions which the work aims to discover using RL. The work done shows that the method is able to learn useful functions and use these functions to improve performance by executing updates. The function seems to correspond to some notion of the value function and hence provides confirmation to existing intuitions in the RL community.

2.3.4 Algorithm Selection

Some problems have several algorithms which can be used to solve them efficiently in different scenarios. The responsibility to select the correct algorithm for the current problem is a problem that was usually left to humans. In a very early work [23] proposed to use RL to find the correct algorithm for a given problem with a goal to minimize the total execution time. They formalize this problem as a MDP and then solve it using Q-Learning which can be combined with any of the techniques such as Monte Carlo or Temporal Difference.

Since then several other machine learning techniques have been able to perform algorithm selection effectively [22] and several of them are RL methods. The field has been a constant success due to its use in methods like boosting and even compiler optimizations.

2.3.5 RL in Mathematics

Reinforcement learning has been of particular help for two different purposes in mathematics: proving known theorems and refuting conjectures by finding counterexamples.

2.3.5.1 Theorem Proving

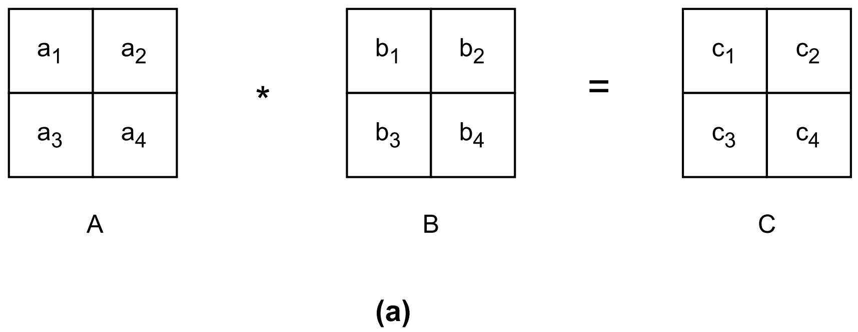
[18] uses Monte Carlo simulations which are steered by RL algorithms to find valid proofs. The typical problem which is considered in this method is that of *connection-based theorem proving*. This kind of proof style starts with a conjecture and a set of axioms in first-order logic (FOL). The algorithm searched for proof that shows that the negated conjecture along with the axiom can not be satisfied. This is done via constructing a closed connection tableau. The method proposed in the paper is to allow MCTS to guide the exploration and extension of the connection tableau. The method was trained on a set of existing proofs and is able to come up with proofs on unseen problems.

[34] and [3] work on the application of RL to Automated Theorem Proving (ATP). [3] comes up with a benchmark for deep-learning-driven theorem provers with coverage of mathematical theorems on calculus. It then comes up with an RL-based ATP which works by using the *proof search graph* as a state. A two-tower neural architecture is used to find proofs in the graph structure.

A complete toolkit for automated theorem proving in connection calculus using RL has been created in Prolog [47]. The toolkit aims at simplifying the work required to create new RL methods for ATP and provides an open-source implementation of the method *rlCoP*.

2.3.5.2 Refuting Conjectures

There are several open conjectures in combinatorics that are incredibly difficult to prove or refute. However, with the ability of RL to efficiently navigate large search spaces, it is possible to try and find counter-examples to the conjectures. This is especially true in combinatorics and graph theory, where it is possible to easily formulate the search space as a game. [44] uses an RL algorithm called the deep cross-entropy method to come up with several constructions which refute conjectures. For example, [2] gave a conjecture about the sum of the largest eigenvalue and matching number on graphs which was provided by an automated conjecture finder. However, this conjecture was later refuted by using the proposed method to find a specific set of graphs where it failed.



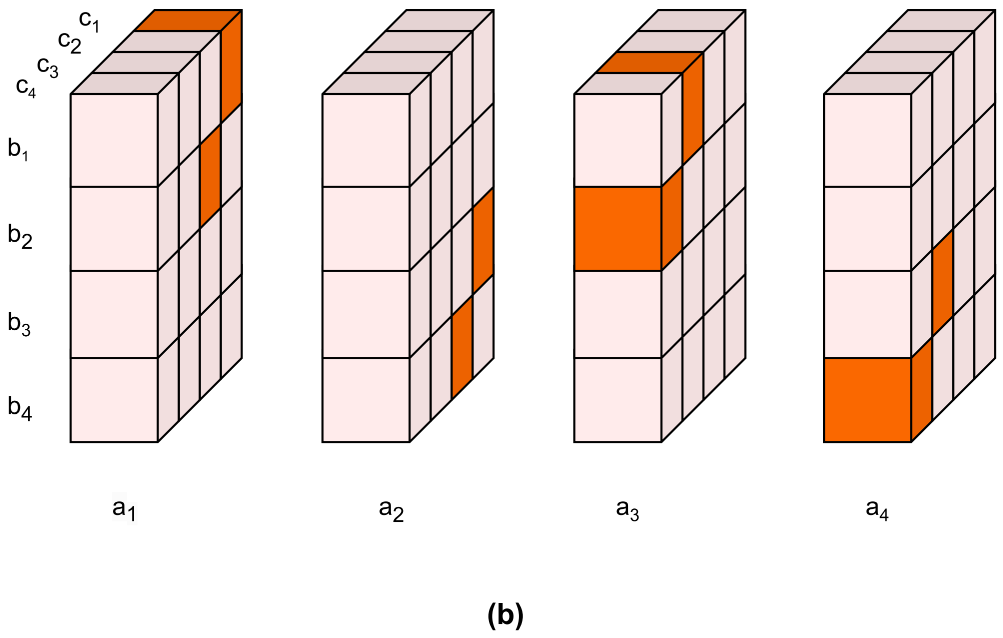


Figure 2.2: AlphaTensor method (a) The original 2×2 multiplication to be performed (b) The matrix multiplication in tensor representation, the darker colored blocks represent value 1 and the rest represent the value 0.

Chapter 3

Alpha Elimination

3.1 Formulating the Game

To apply Reinforcement Learning to the problem of finding optimal permutation sequences for sparse matrices. We need to be able to propose the problem such that an agent can conceive each step like a decision to be made in a game. Fig. 3.1 shows the state of the game at a certain point in the elimination.

Let us assume that the matrix is denoted by A and we are at step i of the elimination, i.e., we are trying to eliminate entries below diagonal of the i_{th} column. Elimination of column i involves (i) Picking a row j as a pivot where the value at column i is non-zero (ii) Swapping row j for the current row i (iii) Using the value $A[i][i]$, we make all values $A[k][i] = 0$ for $k > i$ by performing elementary row operations. Fig. 3.1 also shows the elimination procedure.

The important components to define an RL environment from the existing problems are :

1. **State** The state includes the matrix A as well as the index i which represents the current column to be eliminated. The final state can in fact be represented as a $(N + 1) \times N$ matrix where the extra row represents the one-hot encoding of the column which is being eliminated.

2. **Action** The agent has to choose one of several legal actions at the step. In our game, the action to be taken by the agent must be a row index j where $j \geq i$ and $A[j][i] \neq 0$ or if no such value exists, $j = i$. The agent must thus choose which row to swap in for the current row as a pivot.

3. **Reward** The reward R can be provided to the agent in two different ways. We can provide the reward at the end of each step as negative of the number of non-zeros created at the current step based on the action. Alternatively, we can provide the total number of non-zeros created as a

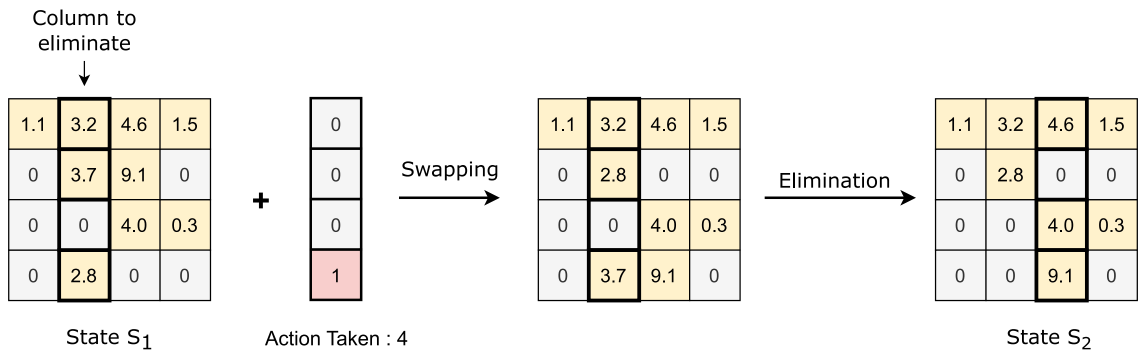


Figure 3.1: Representation of a single step of the game, starting from a state. The action chosen determines the row pivot at each point. After the swapping of the row, the elimination is performed leading to the next step. The column in bold represents the next column to be eliminated.

fraction of initial zeros in the matrix as a negative reward. While both these reward mechanisms are similar, the latter worked better in practice for us due to the ease of tuning the exploration factor c for MCTS.

4. **Transition** The transition from state s_1 to s_2 on taking action a happens as follows. The row to be swapped to the current position is first chosen using the action, and the swapping is performed. Next, the elimination is performed using the row as the pivot using the standard Gaussian elimination technique. This gets us to the next state.

3.2 Applying Deep Monte Carlo Tree Search

Since we deal with discrete action and state spaces, we can apply methods like MCTS to find good policies for the game. Monte Carlo Tree Search progresses by executing four phases in repetition: select, expand, evaluate and backup. The entire algorithm for our problem is shown in form of pseudo-code in Algorithm 1. Although we have discussed MCTS in general in previous sections of the thesis, we show how it applies to our problem in the following section. Further, we go into details about how DL fits into the picture which we had avoided on purpose before.

Algorithm 1 Monte Carlo Tree Search Algorithm for Alpha Elimination

```
Require: Starting State $S$, Model $M$, Immediate reward function $R$
  for loop $\leftarrow$ 1 to num_mcts_loops do
      root $\leftarrow S$
      cur_node $\leftarrow S$
      while True do                                                    ▷ Runs till expand phase reached
          Select best action $A$ which minimizes UCT    (See (3.4)).      ▷ Select Phase
          Compute UCT using prior values and noise
          if cur_node.children[$A$] $\neq$ null then                    ▷ Move to next state if discovered
              cur_node $\leftarrow$ cur_node.children[$A$]
          else
              new_state $\leftarrow$ cur_state.step($A$)                ▷ Expand Stage
              if new_state = null then                                 ▷ Leaf Node
                  break
              end if
              cur_state.children[$A$] $\leftarrow$ new_state
              store reward[cur_state, $A$]
                          $\leftarrow$ $R$(new_state) - $R$(cur_state)
              break
          end if
      end while
      cur_reward $\leftarrow$ model(cur_state)                          ▷ Evaluate Phase
      while cur_node $\neq$ root do
          p_action $\leftarrow$ action from cur_state to parent of cur_state
          cur_state $\leftarrow$ cur_state.parent                       ▷ Backup Phase
          cur_reward $\leftarrow$ reward[cur_state, p_action] + $\gamma$ cur_reward    (see (3.2))
          Update cur_state.Q_value[$A$, p_action] with cur_reward    (See (3.2), (3.3))
          Update cur_state.N_value[$A$, p_action]    (See (3.1))
      end while
  end for
```

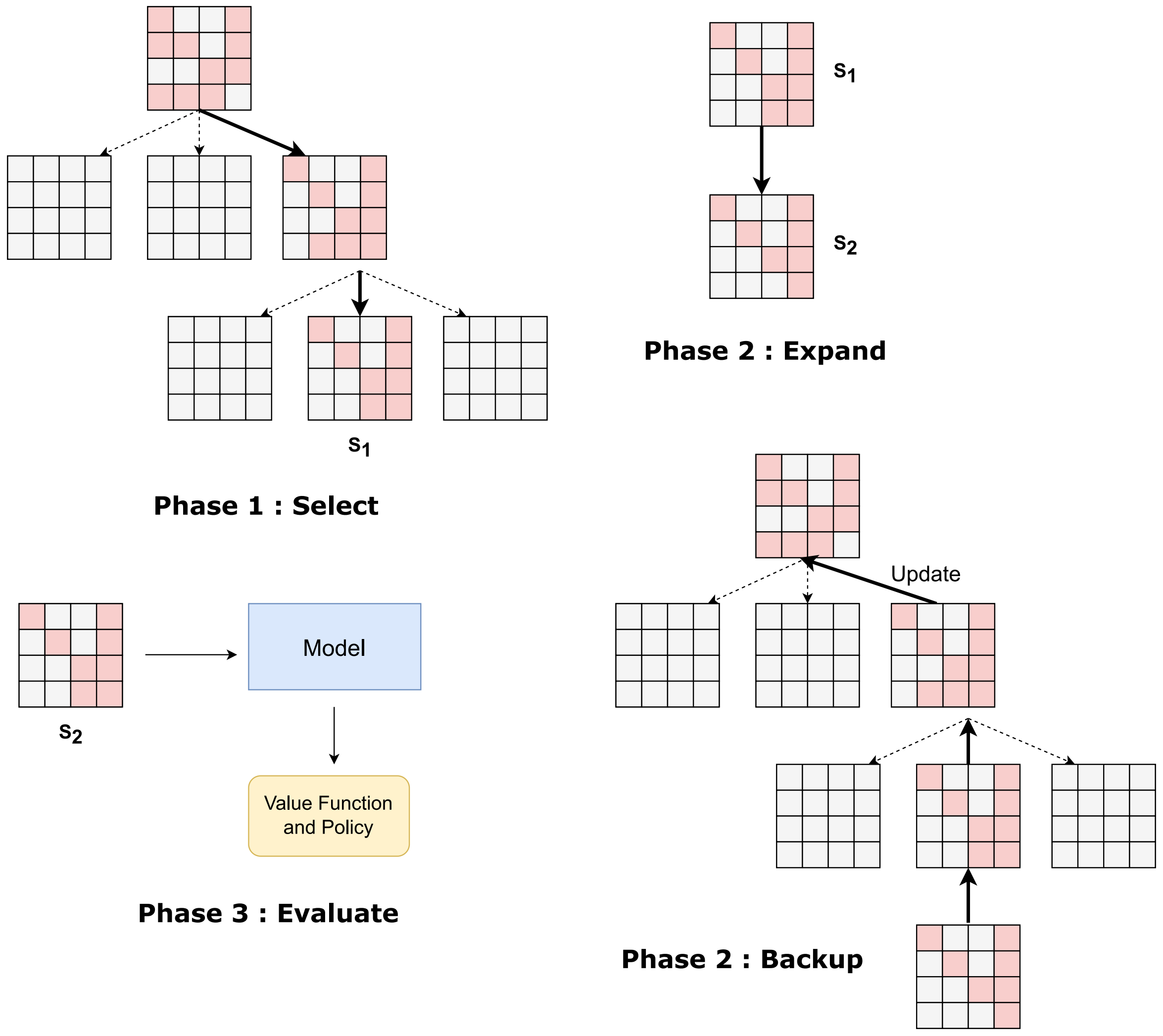


Figure 3.2: Four Phases of the MCTS Algorithm

3.2.1 Select

This step involves starting with a specific state and selecting a node to explore until a state node is reached which has not been explored yet. The selection starts at the root of the subtree to be explored at time t. The agent selects an action from a set of legal actions according to some criteria recursively.

Assuming that $R(S, a)$ represents the immediate reward for taking action a at state S, we keep track of two things during the MCTS procedure (over all passes during exploration):

1. N-values: $N(S, a)$ represents the number of times the state action pair has been taken in total.
2. Q-values: $Q(S, a)$ represents the expected long term reward for the state action pair (S, a). To calculate this, we simply keep track of $W(S, a)$ which is the sum of all rewards received over the

previous iterations. Here N, W and Q are updated as follows:

$$N(S,a) = N(S,a) + 1 \tag{3.1}$$

$$W(S,a) = R(S,a) + \gamma W(S,a) \tag{3.2}$$

$$Q(S,a) = \frac{W(S,a)}{N(S,a)}. \tag{3.3}$$

These updates are actually performed in the backup stage.

We use an asymmetric formulation of Upper Confidence Bound on Trees (UCT) as a criteria for selection of the next action at each step

$$\text{UCT}(S,a) = Q(S,a) + c\frac{\sqrt{N(S,a)}}{N(S,a)} \times P(a|S), \tag{3.4}$$

where c represents the exploration-exploitation constant (Higher c encourages exploration) and $P(a|S)$ represents the prior probability of taking action a given state S. $P(a|S)$ is calculated by adding Dirichlet noise to the function approximator f (neural network in our case) prediction.

$$P(a|S) = (1-\epsilon)f(S) + \epsilon\eta_\alpha. \tag{3.5}$$

Here, $\eta_\alpha \sim Dir(\alpha)$ where $\alpha = 0.03$ and $\epsilon = 0.25$. Here $Dir(\cdot)$ stands for Dirichlet distribution. This ensures that all moves are tried while search still overrules bad moves [38]. The prior estimate $P(a|S)$ improves as the MCTS continues to explore.

3.2.2 Expand

The expand step is invoked when the select reaches a state and takes an action which has not been explored yet. The step involves creating a new node and adding it to the tree structure.

3.2.3 Evaluate

On reaching a node newly created by the expand stage or reaching a leaf node, the evaluation phase is commenced. This involves estimating the long term reward for the current state using a neural network as an estimator. The neural network is provided the current state as the input, which estimates the expected reward and the prior probabilities for each action from the current state. The neural network architecture used for this purpose is described in the following section.

3.2.4 Backup

Once the evaluation phase is completed, the reward value estimated at the last node in the tree is propagated backwards until the root is reached. At each of the ancestor nodes, the values of the expected reward for each action Q and the number of times each action is taken N are updated using the update equations described previously.

As the MCTS probes the search space, it gets better estimates for the prior and the expected reward. The $N(S, a)$ represents the policies to take, and is thus used for training the policy (3.6), while the average Q value is used to train the value estimator (3.7)

$$\pi(a|S) \propto N(S, a) \tag{3.6}$$

$$V(S) = \frac{\Sigma_a W(S, a)}{\Sigma_a N(S, a)}. \tag{3.7}$$

3.3 Neural Network Architecture

The role of a neural network in the Deep MCTS algorithm is to be able to act as a function approximator that can estimate the expected reward ie. the Q-values for a particular state and the expected reward for each state-action pair for that state. Since actually calculating the Q-values is not feasible due to the intractable size of the search space, a neural network is used instead due to its ability to learn rules from the previous exploration data directly.

MCTS provides the neural network with a state S and the network must output two values: (i) π, estimate of the Q values for each action (ii) V, value function of that state. This is achieved by adding two output heads to the same neural network.

The main challenge at this point was on how to provide the sparse matrix as an input to a neural network. For this, we considered several possible architectures.

1. **Convolutional Neural Network (CNN):** Taking into account that a sparse matrix is essentially a image where each pixel is either zero, or some non-zero value, CNN is an intuitive choice for providing sparse matrix an an input. CNNs have been used previously for similar problems such as board games such as Go [38], Chess and Shogi [39]. CNN's are successfully able to capture row and column dependence which is present in the problem.

2. **Linear Sparse Networks with Positional Encodings:** Sparse networks for linear networks are much more efficient than sparse CNN's. A simple method of unrolling the rows matrix and

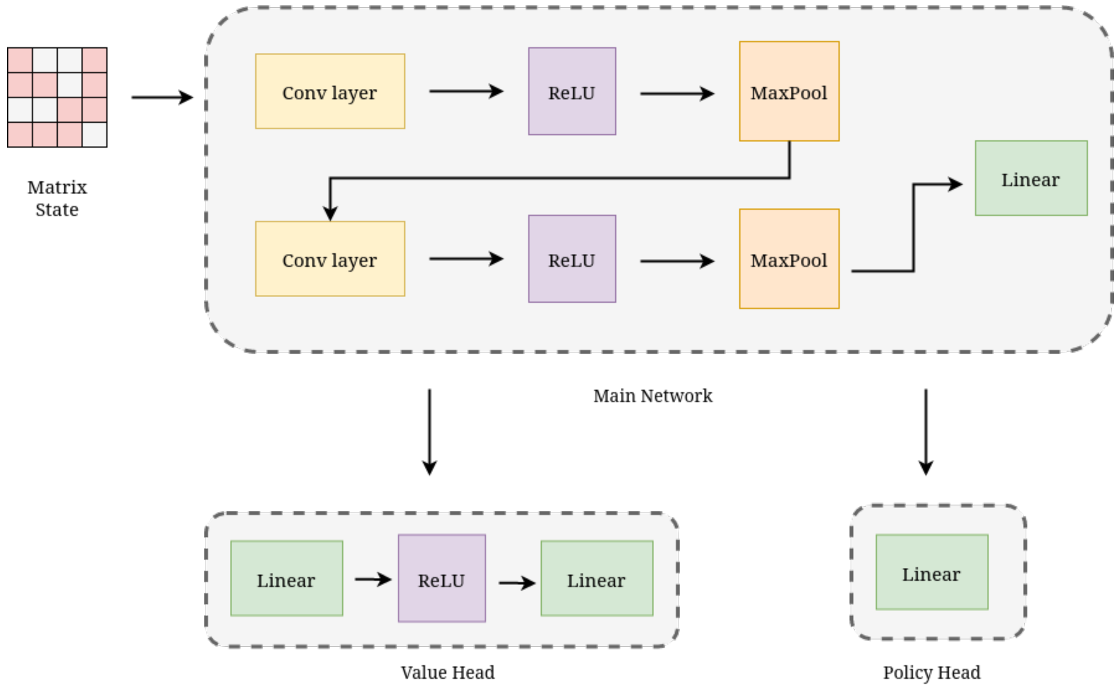


Figure 3.3: Architecture of the neural network used for value and policy prediction. The first and second convolution layers have kernel sizes 3 and 5 respectively with a padding of 2 and stride 1 over 3 channels. The MaxPool layers have a kernel size of 2. Each of the linear layers have a single fully connected layer. The hidden layer dimensions are equal to the matrix size N, except for the last layers, which correspond to 1 for the value head and N for the policy head.

providing it as single dimensional input to the network has the disadvantage that most of the column information is lost. For example, any two elements which are vertically adjacent have no way to preserve that information. Thus, positional embedding would need to be used to provide this information. However, this method still suffers from scaling problems due to the number of parameters being proportional to the size of the matrix N.

3. **Graph Neural Networks:** There is an obvious mapping between a sparse matrix. If the matrix represents the adjacency matrix of the graph, every non-zero in the matrix the matrix $A[i][j] \neq 0$ represents the an edge from i to j. However, providing the structure of a graph as the input to a GNN is a difficult task and there seems to be no obvious way to do it. We leave this idea as a future direction to work on.

Intuitively, a sparse matrix can be treated as an image with mostly zero pixels. CNNs have been successful in similar problems capturing row and column dependencies, for example, in board games such as Go [38], in Chess and Shogi [39]. While sparse layers for artificial neural networks are much more efficient than CNNs, a simple method of unrolling the matrix and providing it as single-dimensional input to the network does not work as most column information is lost. For example, any two vertically adjacent elements cannot preserve that information. Thus, positional embedding would need to be added with column information. However, this approach faces scaling issues, as the parameter count is proportional to matrix size N. Graph neural network could also be used, but it was unclear how to provide graph structure as GNN input. This idea remains a potential future direction.

We decide to use CNN as it emphasizes the row-column relationship, and is relatively simple to understand and implement. An issue with standard CNN is that it needs to densify the sparse matrix to be able to operate on it. Thus, we can use a sparse version of CNN [8] to run the CNN algorithm on larger matrices and take advantage of the sparsity. However, the benefit in running time while using sparse convolution only comes above a certain threshold of the matrix size N. However, as we will see at this threshold or above of matrix size, the time taken to train the network increases significantly due to large search space and alternative ways to scale the network have to be explored anyway.

The final network architecture is denoted in Fig. 3.3. The problem of scaling to larger matrices will be discussed in detail. We observed that masking of the input also led to a significant improvement in rate of learning by the model.

3.4 Algorithm Details

3.4.1 Training

We train our model on randomly generated matrices with a certain sparsity level. For each train matrix, the MCTS starts the self-play at time step 0 and generates a data point for every time step. Each data point consists of the state as input and the respective Q-values and updated value function as the output. These data points are added to a buffer which is used to train the neural network after a certain set of intervals. Prioritized experience replay (PER) [38] is used to efficiently train the neural network. The use of a buffer is necessary due to the fact that each data point must be independent from the previous, which is a fundamental assumption for training of neural networks.

3.4.1.1 Size of matrix to train:

The size of the matrix on which we train the neural network must be fixed, since the input to the neural network cannot vary in size. However, the size of the matrix to be tested on is variable. Hence, we use the block property of LU decomposition to always change our problem into the same fixed size. For example, we train our Deep MCTS for a large matrix of size $N \times N$, but in practice we may get a matrix A of size $n \times n$ where $n \leq N$. We can however, convert the smaller matrix to the size it was originally trained by adding an identity block as follows

$$\begin{bmatrix} I_{N-n} & 0 \\ 0 & A_{n\times n} \end{bmatrix}_{N\times N}.$$

The LU decomposition of the above matrix is equivalent to that of A, and the row permutation matrix is also just a padded one. This procedure can also be interpreted as a intermediate time step of the LU decomposition of a larger matrix.

3.4.1.2 Sparsity of matrix:

The structure of the matrix received during testing must be similar to the ones in training, thus we need to make sure that the sparsity levels of the matrices match too. As a general rule, we see that the common matrices in practice have a sparsity level of ≥ 0.85 (ie. more than 85% of values are non-zeros). Thus, the network is trained on various levels of sparsity above this threshold.

3.4.2 Prediction

During prediction, instead of selecting the next action based on UCT, we directly utilize the output of the trained model as a prediction of the expected reward. We therefore choose optimal action $a^* = \max_a \pi(a|S)$ where π is output of policy function of neural network.

At each step, the algorithm outputs the row j to be swapped with current row i. Using this information, we can reconstruct the row permutation matrix P for final comparison with the other methods.

3.4.3 Masking for the Neural Network

Neural networks are powerful learners; however, when presented with an abundance of information, they may struggle to discern the essential aspects of a problem and might establish spurious correlations. In our experiments, we observed that providing the original matrix with floating point entries to the neural network resulted in slow and erratic learning. Consequently, we employed masking of the non-zeros to minimize noise and ensure that the neural network focuses on the role of non-zero entries as pivots.

More specifically, the non-zero entries are converted to a constant value of 1, whereas, the zeros (accounting for floating-point errors) are assigned a value of 0. This masking technique assumes that none of the non-zeros will become zero during elimination. Although this assumption may not be universally true, it is valid in most cases involving real-world matrices with floating-point values. Even in cases where the assumption does not hold, such instances are relatively rare and have minimal impact on the overall policy.

3.4.4 Scaling to Larger Matrices

The method effectively finds row permutation matrices, but the search space for even small matrices is vast. Matrix size N and training data requirements increase with larger matrices. In real-life applications, matrix sizes can reach up to millions or higher. To address this, we employ a graph partitioning algorithm from the METIS library [19] to partition the matrix into parts of size 500, which allows for efficient learning within a reasonable time-frame. We remark here that most LU factorization for large matrices are anyway partitioned into small parts to achieve parallelism on modern day multi-core or multi-CPU architectures [14], and only "local" LU factorization of the smaller sub-matrices are required.

This size is effectively impossible to deal with directly and has to be partitioned into smaller size matrices anyway. Dividing them into smaller matrices is also the best way to parallelize an algorithm over multiple CPU's. Thus, we explore graph partitioning techniques to partition our matrix into smaller sizes. The matrix $A \in \mathbb{R}^{n \times n}$ can be viewed as the adjacency matrix of a graph with n vertices. If we can find a cut in the graph, we can club together all the vertices on either size of the cut to form a smaller matrix. With a k-way cut, it is possible to partition a very large matrix into a set of smaller matrices that are easier to deal with.

The figure 3.4 shows how a graph can be effectively partitioned. Once a suitable cut is found in the graph, the set of nodes in the graph cut (which is supposed to be much smaller than the remaining partitions) is permuted to be at the end of the adjacency matrix and the two partitions are permuted to be together. Since each of the two remaining partitions do not have nodes as neighbors with each other, the anti-diagonal blocks are empty. Thus, we have effectively divided the original problem into smaller subproblems. Our Elimination algorithm can be used to on these smaller matrices.

We use the METIS library [19] to partition very large matrices into ones of size $N = 500$. This seems to be an optimal number to train reordering as our algorithm is able to learn well at this size and matrices of this size are suitable for all applications on the CPU in a reasonable time.

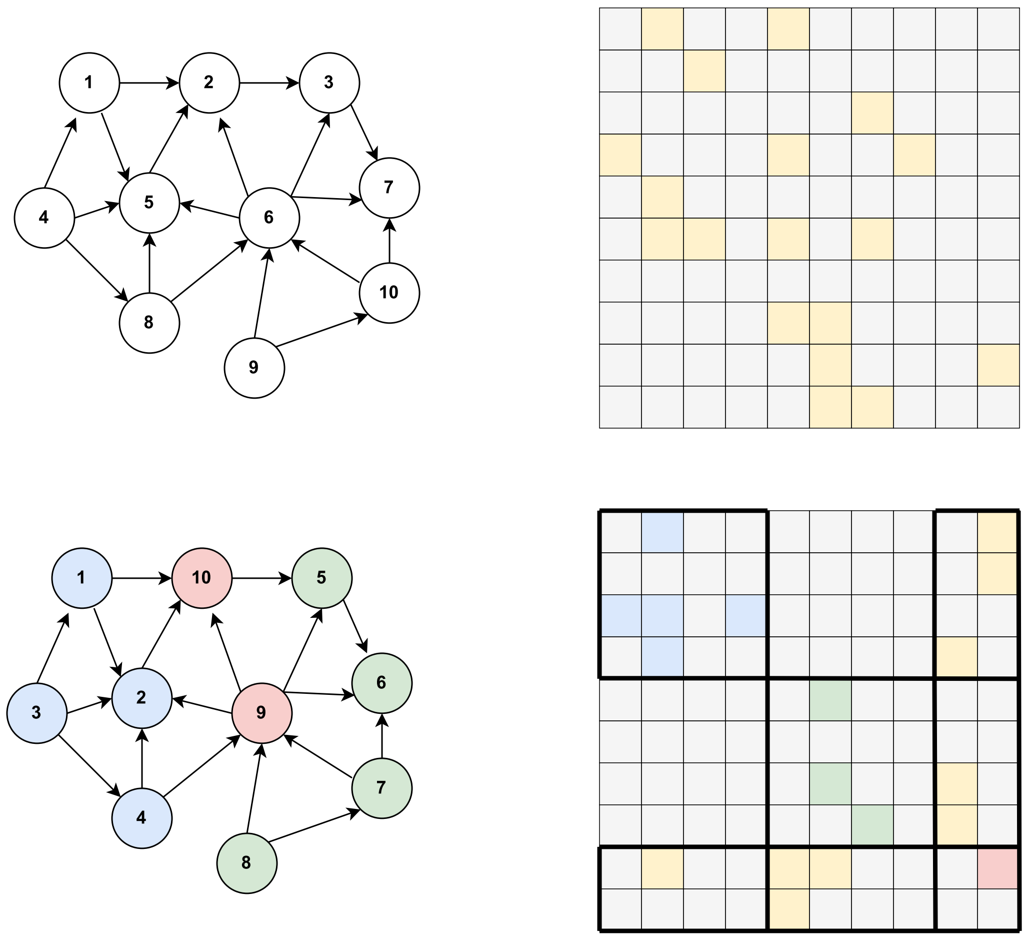


Figure 3.4: Sparse matrix before and after graph partitioning. The unpartitioned graph represents the initial matrix. A non-zero entry at $A[i][j]$ means that there is an edge from node i to node j. The partitioning is done by finding a set of nodes that separate other blocks of nodes. Then the blocks are grouped together with the separator nodes at the end, this creates a new sparse matrix easily separable into blocks.

Chapter 4

Experimentation and Results

4.1 Experimental Setup

Experiments were conducted on a Linux machine equipped with 20 `Intel(R) Xeon(R) CPU E5-2640` v4 cores @ 2.40GHz, 120GB RAM, and 2 `RTX 2080Ti` GPUs. The total number of non-zeros in the LU decomposition is used as the evaluation metric, as our method aims to minimize it. We compared our approach to the naive LU decomposition in sparse matrices and existing heuristic algorithms that minimize fill-in, such as ColAMD [11], SymRCM [25], and SymAMD. After exporting the matrix to MATLAB, where these methods are implemented, LU decomposition was performed.

The final evaluation involved matrices from the SuiteSparse Matrix Collection [20]. Table 4.1 displays the selected matrices, which span various application areas, symmetry patterns, and sizes ranging from 400 to 11 million elements.

4.2 Compared Methods

The comparison between Alpha Elimination and the baseline as well as the other methods is shown in Table 4.2. As it is evident from the results, Alpha Elimination obtains significant reduction in the number of non-zeros as compared to the other methods. This leads to significant reduction in storage space for the factors of the sparse matrices, and leads to reduction in solve time using LU factorization. The reduction in the number of non-zeros provides even more significant memory savings when the size of the matrices increases. Our method produced up to 61.5% less non-zeros on large matrices than the naive method and up to 39.9% less non-zeros than the best heuristic methods. While in some matrices our method gives a significant reduction, some matrices are much simpler in structure, providing much lesser time for improvement over simple algorithms. For example, the matrix *ohm500* has a very simple structure (almost already diagonal) and it is trivial for every row or column reordering algorithm to figure out the optimal ordering. Thus, all the methods end up having the same number of non-zeros. Some of these fill-reducing ordering methods are not applicable for non-symmetric matrices, hence applied on symmetric part $A + A^T$ of a matrix A; whereas, our proposed method is not restricted by structural assumptions on the matrix.

4.3 Time Comparison

Table 4.4 presents the total time taken for finding the permutation matrix and subsequently performing LU decomposition. The time required for LU decomposition decreases when the algorithm processes fewer non-zeros. As shown in Table 4.3, the time consumed for performing LU decomposition after reordering is proportional to the number of non-zeros generated during the decomposition process.

For smaller matrices, the time saved during LU decomposition is overshadowed by the time required for ordering. However, with larger matrices, our method not only achieves a reduction in the number of non-zeros but also results in a noticeable decrease in LU decomposition time.

4.4 Hyperparameter Tuning and Ablation Study

The training is stopped when the average reward no longer improves or the average loss does not decrease. As a standard, either of these conditions were generally met for $N = 500$ size matrices by iteration 300. The time taken for training matrices of size 10, 50, 100, 250, 500 and 1000 is 0.09, 1.2, 6.3, 13.1, 27.7 and 122.5 hours respectively for 100 iterations. The number of iterations also increases with increase in N. The most difficult hyperparameter to train is the exploration factor c. The correct value of c determines how quickly the MCTS finds better solutions and how much it exploits those solutions to find better ones. This value is found experimentally. This is best demonstrated using Fig. 4.1.

The advantage of masking the matrix before providing it as input is demonstrated in Fig. 4.2. The graph shows that masking helps the neural network learn better.

Table 4.1: Matrices used for testing from Suite Sparse Matrix Market

Matrix	**Domain**	**Rows N**	**Structurally Symmetric (Y/N)**
west0479	Chemical Process Simulation Problem	479	Yes
mbeause	Economic Problem	496	No
tomography	Computer Graphics / Vision Problem	500	No
Trefethen_500	Combinatorial Problem	500	Yes
olm500	Computational Fluid Dynamics Problem	500	Yes
Erdos991	Undirected Graph	492	Yes
rbsb480	Robotics Problem	480	No
ex27	Computational Fluid Dynamics Problem	974	No
m_t1	Structural Problem	97,578	Yes
Emilia_923	Structural Problem	923,136	Yes
tx2010	Undirected Weighted Graph	914,231	Yes
boneS10	Model Reduction Problem	914,898	No
PFlow_742	2D/3D Problem	742,793	Yes
Hardesty1	Computer Graphics / Vision Problem	938,905	Yes
vas_stokes_4M	Semiconductor Process Problem	4,382,246	No
stokes	Semiconductor Process Problem	11,449,533	No

Table 4.2: Total Non-Zero Count in Lower and Upper Triangular Factors after re-orderings for matrices from Suite Sparse Matrix Market Dataset.

Matrix	**LU Methods**				
	Naive LU	**ColAMD**	**SymAMD**	**SymRCM**	**Proposed Method**
west0479	16358	4475	4510	4352	**3592**
mbeause	166577	126077	NA	NA	**94859**
tomography	108444	41982	NA	NA	**35690**
Trefethen_500	169618	150344	153170	119672	**94632**
olm500	3984	**3070**	**3070**	**3070**	**3070**
Erdos991	61857	4255	4287	4372	**3584**
rbsb480	192928	63783	NA	NA	**55185**
ex27	122464	104292	NA	NA	**63948**
m_t1	9789931	9318461	8540363	8185236	**7398266**
Emilia_923	5.67E08	4.49E08	4.29E08	4.56E08	**3.9E08**
tx2010	1.48E10	3.83E09	2.34E09	2.44E09	**1.3E09**
boneS10	3.98E08	1.89E08	NA	NA	**1.1E08**
PFlow_742	1.98E08	9.20E07	8.43E07	8.92E07	**8.3E07**
Hardesty1	6.03E08	5.91E08	5.90E08	5.92E08	**4.9E08**
vas_stokes_4M	1.35E09	8.71E+08	NA	NA	**5.9E08**
stokes	9.78E10	6.42E10	NA	NA	**3.9E10**

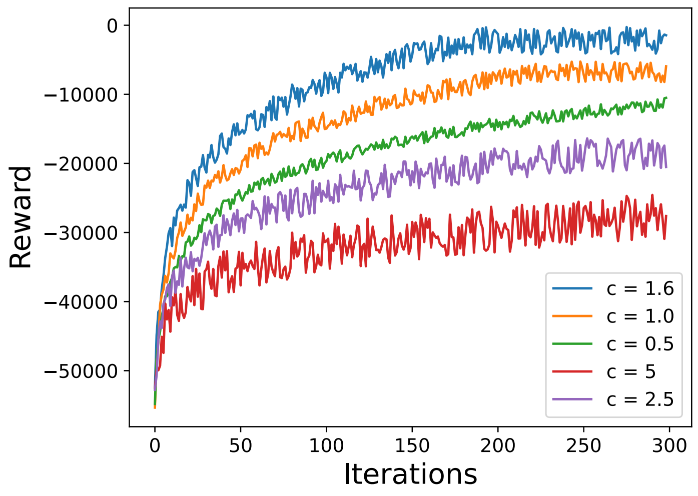


Figure 4.1: Reward plot versus iterations for different exploration factors c.

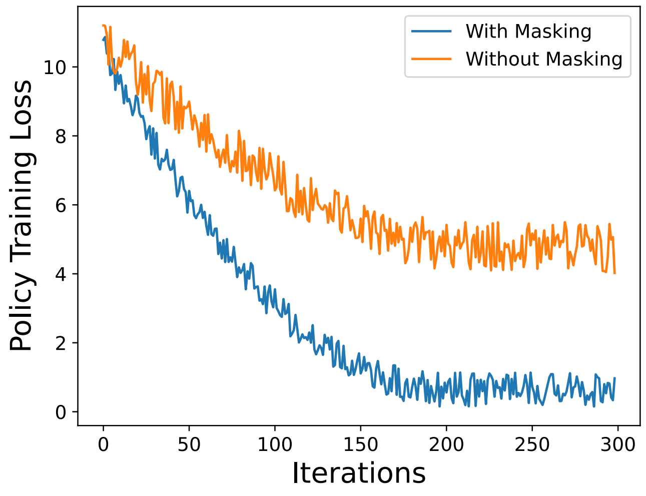


Figure 4.2: Training loss vs Iterations for masked and non-masked input.

Table 4.3: Comparison of time (in seconds) taken for LU factorization after reordering by different methods.

Matrix	**Time taken for LU (s)**				
	Naive LU	**ColAMD**	**SymAMD**	**SymRCM**	**Proposed Method**
mbeause	0.0326	0.0319	NA	NA	**0.0302**
tomography	0.04250	0.0394	NA	NA	**0.0286**
Trefethen_500	0.0498	0.0419	0.0392	0.0347	**0.0302**
m_t1	5.8790	4.7894	4.1321	3.7031	**3.2820**
tx2010	24.3018	15.7040	14.5840	15.6194	**12.9365**

Table 4.4: Comparison of total time (in seconds) taken for LU (including reordering).

Matrix	**Time taken for LU (s)**				
	Naive LU	**ColAMD**	**SymAMD**	**SymRCM**	**Proposed Method**
mbeause	**0.0326**	0.0345	NA	NA	0.0336
tomography	0.0425	0.0475	NA	NA	**0.0391**
Trefethen_500	0.0498	0.0437	0.0404	0.0362	**0.0334**
m_t1	5.8790	5.2174	4.5281	4.1161	**3.9320**
tx2010	24.3018	16.2510	15.077	16.0324	**13.7185**

4.4.1 Inferences from the Spy plots

One way to understand how the re-ordering systems work is to look at what the permutation does to the original matrix. A sparse matrix can be effectively visualized by using spy plots which show non-zero values clearly indicating the structure of the matrix. Figure 4.3 shows the matrix *mbeause* without any permutation applied to it. Figure 4.4 shows the matrix after application of the heuristic algorithm ColAMD. It is visible from the permutation that only the columns have been permuted. Figure 4.6 shows the matrix after the application of the method SymAMD. It is to be noted that this symetric permutation generally ends up with non-zero values at the diagonal. Figure 4.5 shows the matrix after application of the method SymRCM. The SymRCM method works by dividing the matrix into blocks

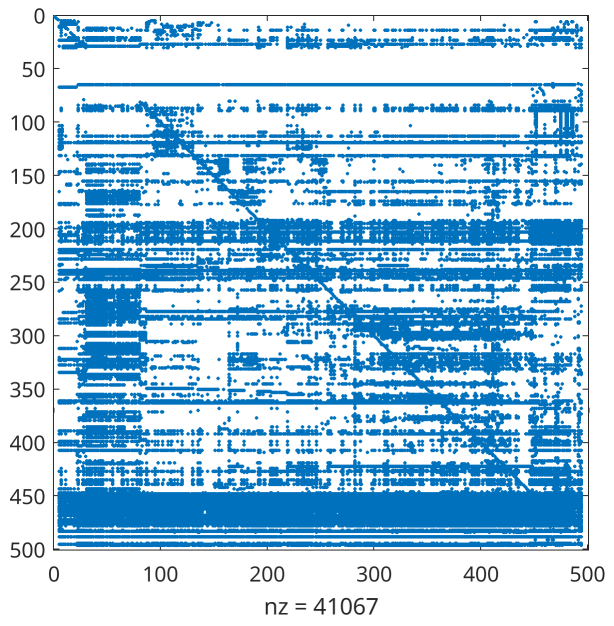


Figure 4.3: Spy plot for matrix *mbeause* before performing LU without matrix permutation

and this is clearly visible from the final matrix. Finally, figure 4.7 shows the matrix after application of the proposed method Alpha Elimination. We can clearly see that the rows are permuted such that the empty rows are at the bottom which makes logical sense.

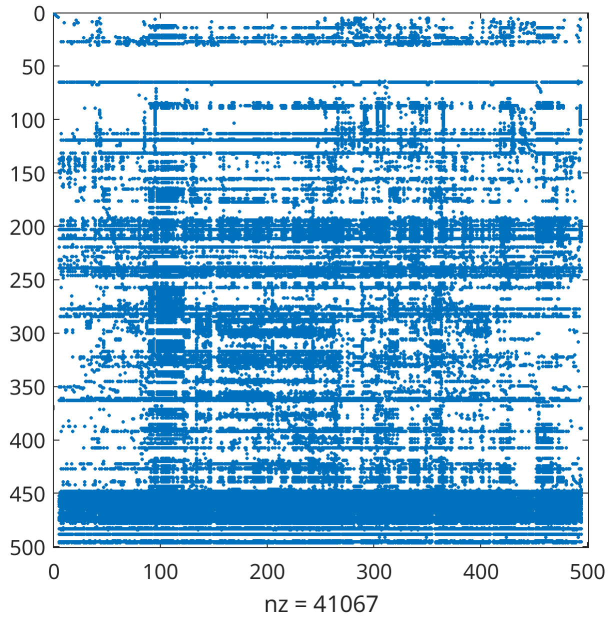


Figure 4.4: Spy plot for matrix *mbeause* before performing LU using the method *colamd*

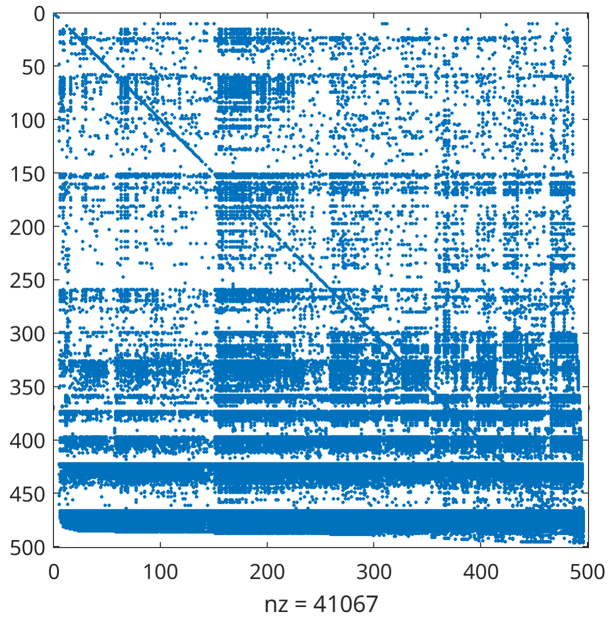


Figure 4.5: Spy plot for matrix *mbeause* before performing LU using the method *symrcm*

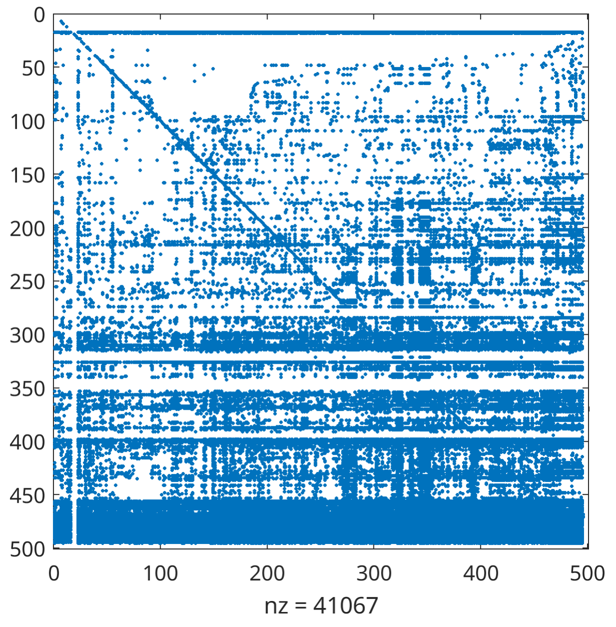


Figure 4.6: Spy plot for matrix *mbeause* before performing LU using the method *symamd*

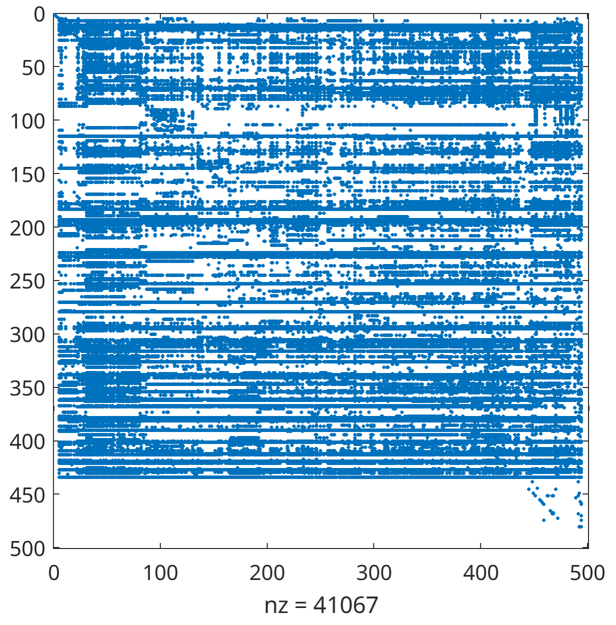


Figure 4.7: Spy plot for matrix *mbeause* before performing LU using the proposed method *Alpha Elimination*

Chapter 5

Extreme Classification

5.1 Introduction to Extreme Classification

Extreme multilabel classification (XML or XMC), is an active area of interest in machine learning. In comparison to traditional multilabel classification, the number of labels is extremely large, hence, the name extreme multilabel classification. Classical methods such as one-versus-all classification, SVM, and neural networks do not scale directly to XML due to a large number of labels. Further, there are several other problems such as the presence of tail labels which render simple models ineffective at solving the problem effectively 5.1.

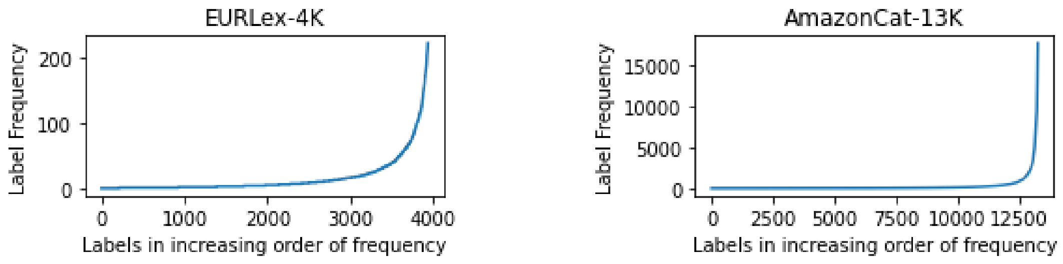


Figure 5.1: The label distribution for two XML datasets. The distribution shows how a large number of (tail) labels have very little support compared to some of the other (head) labels. This distribution gains an even larger tail as the number of labels grows.

5.1.1 Metrics

Given the predicted score vector $\hat{y} \in R^L$ and ground truth label vector $y \in \{0,1\}^L$, theses are the commonly used metrics for comparing performances of XML algorithms.

5.1.1.1 Precision@k:

A very popular metric for scoring how well an XML algorithm has performed is by using P@k. It measures how many of the top k predicted labels are correct. It can formally be represented by

$$P@k := \frac{1}{k} \sum_{l \in \text{rank}_k(\hat{y})} y_l.$$

5.1.1.2 DCG@k and nDCG@k:

XML is also often formulated and scored as a ranking problem. Consequently, Discounted Cumulative Gain (DCG) is a common metrics for measuring the performance of a ranking system. DCG@k measures how much the ground truth scores of each of the top k labels predicted by the algorithm add up to. Normalized Discounted Cumulative Gain (nDCG) is a normalized version of the same metric.

$$\text{DCG@k} := \sum_{l \in \text{rank}_k(\hat{y})} \frac{y_l}{\log(l+1)}, \qquad \text{nDCG@k} := \frac{\text{DCG@}k}{\sum_{l=1}^{\min(k,||y||_0)} \frac{1}{\log(l+1)}}.$$

5.1.1.3 Propensity Based Metrics:

Often, it is possible to achieve high P@k scores by just focusing on the head labels and completely ignoring the tail distribution. This does not serve the purpose of the problems XML was designed for. Thus, [16] proposes a propensity-based version of all the above metrics. The value p_l represents the propensity score of label l. The addition of the propensity score to the denominator ensures that predicting labels which do not occur as frequently correct provide a higher score. These propensity scores are calculated directly from the dataset.

$$\text{PSP@}k := \frac{1}{k} \sum_{l \in \text{rank}_k(\hat{y})} \frac{y_l}{p_l}.$$

$$\text{PSDCG@}k := \sum_{l \in \text{rank}_k(\hat{y})} \frac{y_l}{p_l \log(l+1)}.$$

$$\text{PSnDCG@}k := \frac{\text{PSDCG@}k}{\sum_{l=1}^{\min(k,||y||_0)} \frac{1}{\log(l+1)}}.$$

5.1.2 Datasets

Several kinds of datasets are used to benchmark XML methods. Titles and reviews scraped from Amazon using the internet archive form a set of datasets. Prediction of product tags is performed using these obtained parameters [28], [30] and [29]. A certain set of datasets focus on prediction of Wikipedia article tags using the titles as well as the content [48]. The EurLex [26] dataset formulates large-scale XML problems for legal documents in the European Legislature. There are also some proprietary datasets for XML such as the Bing advertisement bids.

5.2 Review of different Methods of Extreme Classification

The different existing extreme classification methods can be broadly classified into four categories.

1. **Compressed Sensing Based :** The main goal of these set of methods is to embed the original label space into a smaller space by using some compression algorithm. This allows the label space to be handled by simpler methods. Due to the sparsity of the number of label vector, it is possible to compress it into a smaller space without losing much information. This idea is derived from signal encoding structures in signal processing. The original label is later recovered from the embeddings.

2. **Linear Algebra Based :** These are a set of miscellaneous methods which utilize optimizations using linear algebra. Some of these methods try to generalize compressed sensing based methods by formulating them as optimizations.

3. **Tree Based :** These methods try to use the hierarchy in either the labels or the instances to divide the original problem effectively into smaller problems. They divide the original problem repeatedly until the problem is manageable by simpler algorithms. There are two major types of tree-based methods : label partitioning and instance partitioning.

4. **Deep Learning Based :** In the recent past, deep-learning has become a golden hammer of sorts, and is the best solution in almost every field that requires learning. In XML, however, deep learning was not immediately successful unlike in several other domains. It seemed that it was non-trivial to apply deep learning due to a few reasons. Firstly, XML has a very large label space which implies that the output layer of the deep learning model would need to be very big,

implying extremely large models which do not scale. Secondly, due to amount of data available for tail-labels is very small, leading to DL models being unable to learn effectively. However, recent methods have managed to effectively use Deep Learning for significant gains, especially in larger datasets where there is a lot of data. The deep learning methods are not applied directly, but use some structure to make the original problem tractable by a deep learning model.

5.3 Proposed Deep Learning Method to Perform Extreme Classification : LightDXML

In our work [31], we propose a fast and effective way to use Deep Learning in Extreme Classification. This method improves upon the DeepXML framework. DeepXML [9] comes up with a general framework for applying deep learning methods to XMC. Consequently, they propose a new method called ASTEC using the existing method which achieves a new state of the art for some datasets in the domain. It comprises 4 main modules:

- Module 1: Intermediate Feature Representation Learning
- Module 2: Label Shortlisting
- Module 3: Final Feature Representation Learning
- Module 4: Final Model Learning

Additionally, a module called a *re-ranker* is used to improve the predictions.

LightDXML improves upon the deep learning based XML framework DeepXML by using label embeddings instead of feature embedding for negative sampling and iterating cyclically through three major phases:

1. proxy training of label embeddings
2. shortlisting of labels for negative sampling and
3. final classifier training using the negative samples

5.3.1 Benefits of LightDXML

- Our approach alleviates the requirement for a costly re-ranker
  - re-ranker typically used in recent deep learning based extreme classification methods
- Efficacy of the proposed method on real-world extreme classification datasets
- Specifically, our approach

1. improves train and prediction time
2. has less memory requirements than SOTA DeepXML approaches

5.3.2 LightXML: Method

Algorithm 2 Various steps in LightDXML.

Step 1 Every data point $\boldsymbol{x_i}$ is mapped to the metric space $\mathcal{M} = \mathbb{R}^D$ denoted by $\hat{\boldsymbol{x}}_i$, using a weighted TF-IDF combination of FastText embeddings.

Step 2 A label encoder $\boldsymbol{\mathcal{Z}_l}$ maps each label to the same metric space $\mathcal{M}$, i.e., $\boldsymbol{\mathcal{Z}_l} : \boldsymbol{l_j} \rightarrow \hat{\boldsymbol{l_j}} \in \mathcal{M}$, where $\boldsymbol{l_j}$ is the label centroid of the j^{th} label. We use the Euclidean distance metric in the space $\mathcal{M}$.

Step 3 Using the label encoding $\boldsymbol{\mathcal{Z}_l}$, a shortlist $\boldsymbol{\mathcal{S}_k}(\boldsymbol{x_i})$ is created to search for the top k labels using a sub-linear search structure, i.e, $\boldsymbol{\mathcal{S}_k} : \boldsymbol{x_i} \rightarrow [\boldsymbol{l_{i_1}}, ..., \boldsymbol{l_{i_k}}] \in [1, L]^k$.

Step 4 A sparse one-vs-all extreme classifier $\boldsymbol{\mathcal{Z}} : \boldsymbol{x_i} \rightarrow \mathbb{R}^L$ is trained to predict the final labels. This step modifies the feature representation of data points to generate $\hat{\boldsymbol{\mathcal{Z}}}_{\boldsymbol{x}}$.

Step 5 Iterate over steps 2, 3, and 4 until (validation) accuracy is maximized. This completes the training process.

Step 6 The final predictions are made as an ensemble of the shortlist and the extreme classifier.

5.3.3 Experimental Results

We conduct extensive experiments on several datasets of varying sizes and compare with existing methods. Our method has the least prediction time on almost all the datasets. In particular, compared to the very recently proposed Astec, our prediction time is two to four times faster. Our model size remains the smallest on three datasets, and second best on two datasets.

In particular, compared to Astec, our model requires significantly less memory, yet retaining better accuracy than Astec. Although, model sizes and train time of tree-based methods such as Bonsai, Parabel is the least on some datasets, they are known to suffer from poor accuracy as also indicated by lower P and PSP scores in Table. Also, when compared to Astec, we have better PSP scores on most datasets, confirming our belief that our method caters to tail labels better.

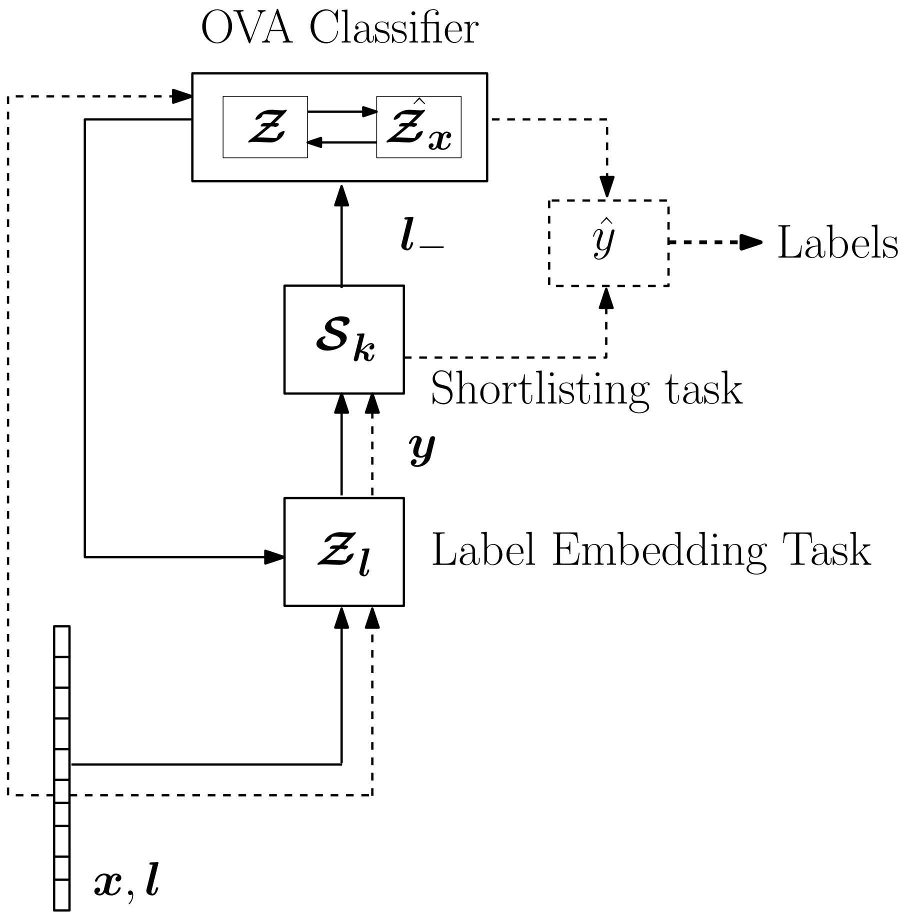
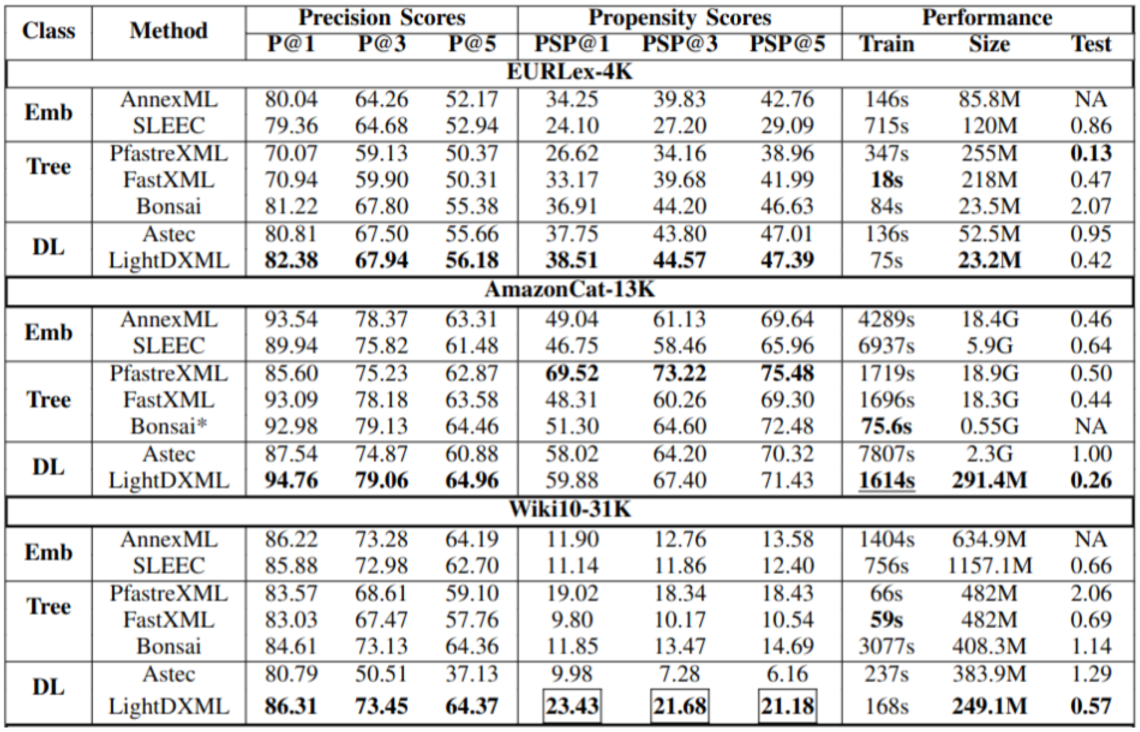


Figure 5.2: LightDXML architecture. The block $\mathcal{Z}_l$ is the label encoding architecture that generates dense label embeddings. The block $\mathcal{S}_k$ additionally generates the k negative labels. Finally, the block $(\mathcal{Z}, \hat{\mathcal{Z}}_x)$ is the extreme classifier that also generates a new set of data feature embeddings. $\hat{\mathcal{Z}}_x$ is then fed into the block $\mathcal{Z}_l$ for alternating optimization over the blocks. Solid lines shows the training phase and the dotted lines show the prediction phase.

Figure 5.3: Experiment results for LightDXML on three popular datasets.



Class	Method	Precision Scores			Propensity Scores			Performance		
		P@1	P@3	P@5	PSP@1	PSP@3	PSP@5	Train	Size	Test
EURLex-4K										
Emb	AnnexML	80.04	64.26	52.17	34.25	39.83	42.76	146s	85.8M	NA
	SLEEC	79.36	64.68	52.94	24.10	27.20	29.09	715s	120M	0.86
Tree	PfastreXML	70.07	59.13	50.37	26.62	34.16	38.96	347s	255M	**0.13**
	FastXML	70.94	59.90	50.31	33.17	39.68	41.99	**18s**	218M	0.47
	Bonsai	81.22	67.80	55.38	36.91	44.20	46.63	84s	23.5M	2.07
DL	Astec	80.81	67.50	55.66	37.75	43.80	47.01	136s	52.5M	0.95
	LightDXML	**82.38**	**67.94**	**56.18**	**38.51**	**44.57**	**47.39**	75s	**23.2M**	0.42
AmazonCat-13K										
Emb	AnnexML	93.54	78.37	63.31	49.04	61.13	69.64	4289s	18.4G	0.46
	SLEEC	89.94	75.82	61.48	46.75	58.46	65.96	6937s	5.9G	0.64
Tree	PfastreXML	85.60	75.23	62.87	**69.52**	**73.22**	**75.48**	1719s	18.9G	0.50
	FastXML	93.09	78.18	63.58	48.31	60.26	69.30	1696s	18.3G	0.44
	Bonsai*	92.98	79.13	64.46	51.30	64.60	72.48	**75.6s**	0.55G	NA
DL	Astec	87.54	74.87	60.88	58.02	64.20	70.32	7807s	2.3G	1.00
	LightDXML	**94.76**	**79.06**	**64.96**	59.88	67.40	71.43	**1614s**	**291.4M**	**0.26**
Wiki10-31K										
Emb	AnnexML	86.22	73.28	64.19	11.90	12.76	13.58	1404s	634.9M	NA
	SLEEC	85.88	72.98	62.70	11.14	11.86	12.40	756s	1157.1M	0.66
Tree	PfastreXML	83.57	68.61	59.10	19.02	18.34	18.43	66s	482M	2.06
	FastXML	83.03	67.47	57.76	9.80	10.17	10.54	**59s**	482M	0.69
	Bonsai	84.61	73.13	64.36	11.85	13.47	14.69	3077s	408.3M	1.14
DL	Astec	80.79	50.51	37.13	9.98	7.28	6.16	237s	383.9M	1.29
	LightDXML	**86.31**	**73.45**	**64.37**	$\boxed{\textbf{23.43}}$	$\boxed{\textbf{21.68}}$	$\boxed{\textbf{21.18}}$	168s	**249.1M**	**0.57**

Chapter 6

Conclusion and Future Work

Through the work conducted in this thesis, we have established that it is possible to solve existing problems in mathematics which are currently solved by heuristic algorithms by replacing them with reinforcement learning algorithms. This can be done if the problem can be effectively represented as a single or multi player game and the game state can be provided as input to a neural network. The experiments conducted show that the process of row elimination performed by the RL agent leads to significantly lesser non-zeros compared to traditional heuristics algorithms. Another advantage of using the proposed method lies in the fact that it can be adapted to specific applications with different requirements by adjusting the reward function accordingly. For example, further work can be done on developing an algorithm which leads to matrices with higher numerical stability by compromising on the final number of non-zeros.

The neural network employed for the sparse matrix serves as a critical component of the algorithm. Further research focusing on the development of scalable architectures capable of handling large sparse matrix inputs may enhance the quality of the policy output. A combination of the heuristic methods along with MCTS for bootstrapping with additional training data can be explored in the future to get further improvements.

The work done in the thesis is very relevant as the method can be used and incorporated into math libraries to improve the performance of LU on sparse matrices.

Finally, the work done in extreme classification presents a light-weight, simpler, robust, and faster framework LightDXML for extreme classification which was put to test on several large scale real world datasets. Very recent methods which use label information, i.e., label captions and metadata perform better than LightDXML. LightDXML can be modified to learn label embeddings from this information too to improve its performance. The work sets a precedent on how different modules can be adjusted to lead to significant gains in the running and prediction times in a framework.

Chapter 7

Publications

7.1 Relevant Publications

1. **Arpan Dasgupta, Pawan Kumar**. *"Alpha Elimination: Using Deep Reinforcement Learning to Reduce Fill-In during Sparse Matrix Decomposition"* **European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML) PKDD 2023**

2. **Istasis Mishra, Arpan Dasgupta, Pratik Jawanpuria, Bamdev Mishra, Pawan Kumar**, *"Light-weight Deep Extreme Multilabel Classification"*, **International Joint Conference on Neural Networks (IJCNN), 2023**

Bibliography

[1] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, 1996.

[2] M. Aouchiche and P. Hansen. A survey of automated conjectures in spectral graph theory. *Linear algebra and its applications*, 432(9):2293–2322, 2010.

[3] K. Bansal, S. Loos, M. Rabe, C. Szegedy, and S. Wilcox. Holist: An environment for machine learning of higher order logic theorem proving. In *International Conference on Machine Learning*, pages 454–463. PMLR, 2019.

[4] Y. Bengio, S. Bengio, and J. Cloutier. *Learning a synaptic learning rule*. Citeseer, 1990.

[5] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[6] P. Bryant, G. Pozzati, W. Zhu, A. Shenoy, P. Kundrotas, and A. Elofsson. Predicting the structure of large protein complexes using alphafold and monte carlo tree search. *Nature communications*, 13(1):6028, 2022.

[7] Ü. V. Çatalyürek, C. Aykanat, and E. Kayaaslan. Hypergraph partitioning-based fill-reducing ordering for symmetric matrices. *SIAM Journal on Scientific Computing*, 33(4):1996–2023, 2011.

[8] S. Contributors. Spconv: Spatially sparse convolution library. https://github.com/traveller59/spconv, 2022.

[9] K. Dahiya, D. Saini, A. Mittal, A. Shaw, K. Dave, A. Soni, H. Jain, S. Agarwal, and M. Varma. Deepxml: A deep extreme multi-label learning framework applied to short text documents. In *Proceedings of the 14th ACM International Conference on Web Search and Data Mining*, pages 31–39, 2021.

[10] A. Dasgupta and P. Kumar. Alpha elimination: Using deep reinforcement learning to reduce fill-in during sparse matrix decomposition. *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases, 2023*, 2023.

[11] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng. Algorithm 836: Colamd, a column approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software (TOMS)*, 30(3):377–380, 2004.

[12] Y. Duan, J. Schulman, X. Chen, P. L. Bartlett, I. Sutskever, and P. Abbeel. Rl˜2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016.

[13] A. Fawzi, M. Balog, A. Huang, T. Hubert, B. Romera-Paredes, M. Barekatain, A. Novikov, F. J. R Ruiz, J. Schrittwieser, G. Swirszcz, et al. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022.

[14] G. H. Golub and C. F. Van Loan. *Matrix computations*. JHU press, 2013.

[15] A. Guez, D. Silver, and P. Dayan. Scalable and efficient bayes-adaptive reinforcement learning based on monte-carlo tree search. *Journal of Artificial Intelligence Research*, 48:841–883, 2013.

[16] H. Jain, Y. Prabhu, and M. Varma. Extreme multi-label loss functions for recommendation, tagging, ranking & other missing label applications. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 935–944, 2016.

[17] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, et al. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589, 2021.

[18] C. Kaliszyk, J. Urban, H. Michalewski, and M. Olšák. Reinforcement learning of theorem proving. *Advances in Neural Information Processing Systems*, 31, 2018.

[19] G. Karypis and V. Kumar. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. *Technical report*, 1997.

[20] S. P. Kolodziej, M. Aznaveh, M. Bullock, J. David, T. A. Davis, M. Henderson, Y. Hu, and R. Sandstrom. The suitesparse matrix collection website interface. *Journal of Open Source Software*, 4(35):1244, 2019.

[21] V. R. Konda and J. N. Tsitsiklis. Actor-critic algorithms. *Advances in neural information processing systems*, 12:1008–1014, 2000.

[22] L. Kotthoff. Algorithm selection for combinatorial search problems: A survey. *Data mining and constraint programming: Foundations of a cross-disciplinary approach*, pages 149–190, 2016.

[23] M. G. Lagoudakis, M. L. Littman, et al. Algorithm selection using reinforcement learning. In *ICML*, pages 511–518, 2000.

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

[25] W.-H. Liu and A. H. Sherman. Comparative analysis of the cuthill–mckee and the reverse cuthill–mckee ordering algorithms for sparse matrices. *SIAM Journal on Numerical Analysis*, 13(2):198–213, 1976.

[26] E. Loza Mencía and J. Fürnkranz. Efficient pairwise multilabel classification for large-scale problems in the legal domain. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 50–65. Springer, 2008.

[27] D. J. Mankowitz, A. Michi, A. Zhernov, M. Gelmi, M. Selvi, C. Paduraru, E. Leurent, S. Iqbal, J.-B. Lespiau, A. Ahern, et al. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618(7964):257–263, 2023.

[28] J. McAuley and J. Leskovec. Hidden factors and hidden topics: understanding rating dimensions with review text. In *Proceedings of the 7th ACM conference on Recommender systems*, pages 165–172, 2013.

[29] J. McAuley, R. Pandey, and J. Leskovec. Inferring networks of substitutable and complementary products. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*, pages 785–794, 2015.

[30] J. McAuley, C. Targett, Q. Shi, and A. Van Den Hengel. Image-based recommendations on styles and substitutes. In *Proceedings of the 38th international ACM SIGIR conference on research and development in information retrieval*, pages 43–52, 2015.

[31] I. Mishra, A. Dasgupta, P. Jawanpuria, B. Mishra, and P. Kumar. Light-weight deep extreme multilabel classification. *arXiv preprint arXiv:2304.11045*, 2023.

[32] R. Munos et al. From bandits to monte-carlo tree search: The optimistic principle applied to optimization and planning. *Foundations and Trends® in Machine Learning*, 7(1):1–129, 2014.

[33] J. Oh, M. Hessel, W. M. Czarnecki, Z. Xu, H. P. van Hasselt, S. Singh, and D. Silver. Discovering reinforcement learning algorithms. *Advances in Neural Information Processing Systems*, 33:1060–1070, 2020.

[34] B. Piotrowski and J. Urban. Atpboost: Learning premise selection in binary setting with atp feedback. In *Automated Reasoning: 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings 9*, pages 566–574. Springer, 2018.

[35] Y. Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[36] J. Schmidhuber. *Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook*. PhD thesis, Technische Universität München, 1987.

[37] J. Schmidhuber. A 'self-referential'weight matrix. In *ICANN'93: Proceedings of the International Conference on Artificial Neural Networks Amsterdam, The Netherlands 13–16 September 1993 3*, pages 446–450. Springer, 1993.

[38] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

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

[40] A. Sinha, U. Azad, and H. Singh. Qubit routing using graph neural network aided monte carlo tree search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 9935–9943, 2022.

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

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

[43] G. Tesauro et al. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[44] A. Z. Wagner. Constructions in combinatorics via neural networks. *arXiv preprint arXiv:2104.14516*, 2021.

[45] J. X. Wang, Z. Kurth-Nelson, D. Tirumala, H. Soyer, J. Z. Leibo, R. Munos, C. Blundell, D. Kumaran, and M. Botvinick. Learning to reinforcement learn. *arXiv preprint arXiv:1611.05763*, 2016.

[46] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[47] Z. Zombori, J. Urban, and C. E. Brown. Prolog technology reinforcement learning prover: (system description). In *International Joint Conference on Automated Reasoning*, pages 489–507. Springer, 2020.

[48] A. Zubiaga. Enhancing navigation on wikipedia with social tags. *arXiv preprint arXiv:1202.5469*, 2012.