Optimal Construction of Matrix Product Operators and Tree Tensor Network Operators

Student: Hazar Çakır
Supervisor: Prof. Dr. Christian B. Mendl
Advisor: M.Sc. Richard Milbradt
TUM School of Computation, Information and Technology
Technical University of Munich
Munich Germany

hazar.cakir@tum.de
christian.mendl@tum.de
r.milbradt@tum.de

April 12, 2024

Abstract

In the dynamic field of quantum computing, efficient construction of Matrix Product Operators (MPOs) and Tree Tensor Network Operators (TTNOs) plays a pivotal role in understanding complex quantum systems. This project embarks on an approach by applying bipartite graph theory to optimize the construction of MPOs and extending this methodology to TTNOs, a domain where such an application is unprecedented. Initially, the study meticulously reevaluates an existing algorithm for constructing MPOs through bipartite graphs, confirming its validity and efficiency. The transition to TTNOs unveiled significant challenges, necessitating considerable modifications to the algorithm. Despite these obstacles, an optimal construction for TTNOs was achieved, marking a significant advancement in the field. The report details the methodological evolution, from theoretical underpinnings to practical implementation, and underscores a new algorithm to create optimal TTNOs.

1 Introduction

The study of Quantum Computing brings together principles from quantum physics, computer science, and mathematics to improve our understanding of quantum systems. This field is instrumental for developing technologies in quantum computing, cryptography, and the simulation of quantum phenomena. Key to these advancements are Matrix Product Operators (MPOs) and Tree Tensor Network Operators (TTNOs), which are tools for efficiently representing quantum states and operations. While MPOs are well-studied for their application in one-dimensional quantum systems, extending similar methodologies to TTNOs, which are applicable to higher-dimensional systems, remains challenging due to their increased complexity.

This project focuses on the application of bipartite graph theory for constructing MPOs and TTNOs. Bipartite graph theory offers a framework for efficiently determining quantum system interactions i.e. virtual bonds, which is promising for developing algorithms that construct tensor network operators with optimal efficiency. Initially, the project involved a detailed analysis and implementation of a known bipartite graph algorithm for MPO construction, to validate its effectiveness. This is also involving a study of fermionic algebra for the chemical analysis of Hamiltonians to understand real life implementations of the MPO construction.

The next phase is the adaptation of this algorithm for TTNOs. TTNOs are a more general case compared to MPOs, which lacks the linear relation. For that reason, faced substantial challenges that required significant modifications to the original algorithm migrating linear approaches to the tree like structure. Surprisingly enough, it is realized that after all the modifications, the final algorithm doesn't include the bipartite theory.

The objectives of this project were to assess the efficiency of the bipartite graph algorithm in constructing MPOs and to explore the feasibility of extending this approach to TTNOs. Despite encountering theoretical and practical challenges, the project successfully modified and updated the algorithm for optimal TTNO construction. This introduction provides an overview of the project's background, motivation, challenges, and objectives. It sets the stage for the detailed account that follows, covering the literature review, methodology, implementation, and results, culminating in a discussion of the project's contributions to the literature.

The research conducted was part of a guided research project under the supervision of Prof. Mendl and M.Sc. Milbradt . It represents a critical component of my curriculum, providing a unique opportunity to apply theoretical knowledge to

a complex, real-world problem. This project not only allowed me to deepen my understanding of quantum information science but also to contribute to the field by extending existing methodologies to new applications. The process and outcomes of this research are detailed in this report, which serves as the final report for the project.

2 Related Work

In the initial phase of this guided research project, a detailed analysis and implementation were undertaken based on the methodologies outlined in "A General Automatic Method for Optimal Construction of Matrix Product Operators Using Bipartite Graph Theory" by Ren et al. (2020) [1]. This seminal paper provides both a comprehensive description of the algorithm and a theoretical justification for its ability to determine optimal bond dimensions across a broad spectrum of scenarios. Contrary to the assertions made by Ren et al., our investigations revealed that there are certain edge cases where the algorithm does not perform as expected, indicating limitations in its universal applicability.

Furthermore, the paper by Ren et al. is notable not only for its theoretical contributions but also for its practical applications, showcasing three distinct real-world implementations related to the construction of Hamiltonians from various systems. Within the scope of this project, the focus was narrowed to the ab initio electronic Hamiltonian system. The complexities and theoretical underpinnings of this specific system are explained in the subsequent section of this report. A foundational understanding of fermionic algebra is imperative for engaging with this domain, a requirement met through the insights provided by "Matrix Product Operators, Matrix Product States, and ab initio Density Matrix Renormalization Group algorithms" by Chan et al . (2016) [2]. This work significantly contributed to our comprehension of electronic systems and their interplay with tensor networks, laying the groundwork for the applied aspects of our research.

The exploration then expanded to include the construction of Tree Tensor Network Operators (TTNOs), marking the second focus of the project. In this endeavor, "State Diagrams to Determine Tree Tensor Network Operators" by Milbradt et al. (2023) [3] from our chair served as a pivotal reference. Despite its methodological departure from the bipartite graph theory and its suboptimal approach to forming TTNOs, the paper provided a crucial starting point. It offered a novel blueprint for representing the tree structure of TTNOs through state diagrams. This conceptual framework, albeit distinct from the bipartite algorithm's domain, enabled the

development of a new algorithmic approach grounded in bipartite graph theory. Leveraging the state diagram structure, the project embarked on constructing an innovative method for TTNOs, utilizing existing implementations as a foundation.

These foundational texts have guided the initial stages of our exploration, setting a theoretical and practical framework within which our project was positioned. Through the critical examination and application of these sources, our research endeavors to build upon the established knowledge base, addressing gaps and extending the methodology to new contexts.

3 Physical Model

Within the domain of quantum chemistry, the ab initio electronic Hamiltonian system is a pivotal construct for the quantum mechanical description of electron interactions in molecules. This Hamiltonian, devoid of empirical parameters, adheres to the principles derived from first principles or fundamental quantum mechanics. The mathematical form of the electronic Hamiltonian, $\hat{H}$, is presented as follows:

$$\hat{H} = \sum_{pq} t_{pq} a_p^\dagger a_q + \frac{1}{2} \sum_{pqrs} v_{pqrs} a_p^\dagger a_q^\dagger a_r a_s \tag{1}$$

In order to understand the equation, we should first be familiar with the fermionic algebra:

$$a = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \quad a^\dagger = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$$

- $a_p^\dagger$: Creation operator that add an electron to the quantum states p.
- a_p: Annihilation operator that remove an electron from the quantum state p.

Mastering all the details of the field of the fermionic algebra exceeds the scope of this guided research, however some level of understanding is required to understand the given Hamiltonian. In this regard we can simplify the Hamiltonian with explaining the two types of interactions in it:

$$\sum_{pq} t_{pq} a_p^\dagger a_q$$

It represents the one-body interactions. This term sums over all pairs of quantum states p and q. The t_{pq} coefficients are known as one-electron integrals and they typically represent the kinetic energy of electrons moving in the field of the nuclei, as well as the potential energy from the attraction between the electrons and the fixed nuclei. The operator $a_p^\dagger$ is the creation operator which adds an electron to the quantum state p, and a_q is the annihilation operator that removes an electron from the quantum state q. This term captures the behavior of single electrons within the molecular system.

$$\frac{1}{2}\sum_{pqrs} v_{pqrs} a_p^\dagger a_q^\dagger a_r a_s$$

It represents the two-body interactions, specifically the electron-electron repulsion. Here, the summation is extended over all quadruples of quantum states p,q,r, and s. The v_{pqrs} coefficients are two-electron integrals and they describe the repulsion energy between pairs of electrons located in states p and q. The factor of $\frac{1}{2}$ is used to compensate for the double counting of electron pair interactions since the repulsion between two specific electrons is counted once when the electrons are in states p and q and again in states r and s. The sequence of creation and annihilation operators $a_p^\dagger a_q^\dagger a_r a_s$ follows the fermionic anticommutation rules and ensures that the Pauli exclusion principle is obeyed, allowing the correct description of the antisymmetry of the wave function under the exchange of any two electrons.

As it can be seen, with higher number of sites, the number of terms are increasing vastly. The computational analyses conducted by by Chan et al . (2016) [2] and Nakatani and Chan (2013) [4] show that the complexity of determining the optimal bond dimensions for the Hamiltonian's representation via the Density Matrix Renormalization Group (DMRG) method scales as $O(M^3K^3) + O(M^2K^4)$ after extensive mathematical derivations and mathematical manipulations which is excluded in this report. The main motivation of the automated algorithms using bipartite graph theory is to calculate optimal bond dimensions with a more efficient way and guaranteeing to reach to the optimal bond dimensions.

4 Methodology

The research is divided into two main sections as Bipartite Graph Algorithm for MPOs and adaptation of this algorithm to TTNOs. In this section, they will be explained:

4.1 Bipartite Graph Algorithm for MPOs

The algorithm from Ren et al. (2020) [1] has already been well-documented, complete with pseudocode. Therefore, this report will not reiterate those details. Instead, our work focused on replicating the existing algorithm to test its reliability. As anticipated, the algorithm was successful in determining the optimal bond dimensions for most cases. However, our findings indicate that the bipartite graph algorithm does not account for certain exceptional cases, suggesting that it is not entirely comprehensive.

In this section, we will provide a brief outline of the algorithm and discuss a specific case where the algorithm fails to perform as expected.

4.1.1 Bipartite Algorithm

To understand how the algorithm works, first we need to understand how we structure the terms. Each term represented as a chain of operators where each local operator is applied on a site. Let's say we have k terms and n sites. One important feature is edges between two sites. The number of edges determines the virtual bond dimension between MPO sites and the goal of the algorithm is to minimize these edges for each site-to-site connection as much as possible. In the beginning, we will have k parallel chains and have k edges between each $n-1$ connections.

Algorithm does a sweep from one end to the other and through that, optimizes bond connections between sites locally. In our implementation we iterate from left to right and will follow this order throughout this section. We will call left chains as $\tilde{U}$ chains and right chains as $\tilde{V}$ chains. For each edge, we do following specific steps:

1. Create non-redundant operator sets for $\tilde{U}$ and $\tilde{V}$ values called as U and V. These sets are formed as removing the duplicated operators in left and right chain. This is practically combining those chains into one chain.

2. Preserve connectivity between U and V sets. Connect nodes from U and V sets with regard to previous connections before combination done in step 1.

3. Apply Bipartite Algorithm to find required vertices with minimum vertex cover and extract edges to form a maximum matching.

4. Form new U values using determined bonds in step 3

For more detailed explanation, please check Ren et al. (2020) [1]. For the local sites, Figure 1 indicates the state after step 3.

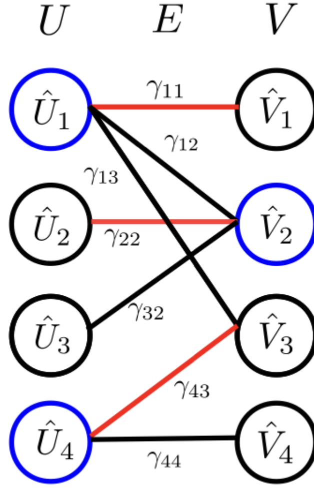


Figure 1: The vertices represent the non-redundant operators in the L- and R- block. The edges represent the interactions with a nonzero pre-factor. The vertices in blue form a minimum vertex cover. The edges in red form a maximum matching

4.1.2 Problematic Fully-Connected Case

Even though there exist a scientific proof in Ren et al. (2020) [1] that the algorithm is complete, we observed a case where the algorithm fails to cover which includes a fully connected edge. Let's first understand the problematic approach of the algorithm.

The bipartite algorithm combines edges from only one side of the bond, picking U and V vertices. So it is either connected on the left or on the right. To understand it better, let's investigate Figure 1. In this diagram, algorithm picks $\hat{U}_1, \hat{V}_2, \hat{U}_4$. Picking $\hat{U}_1$ means combining vertices γ_{11}, γ_{12} ,and γ_{13} on the left side. For $\hat{V}_2$ we combine γ_{22} and γ_{32} on the right hand. However, this approach fails when we need to combine both sides, i.e. fully connected sites.

Let's analyze the Hamiltonian

$$H = \sum_{j=1}^{4} h_j = X_1 Y_1 + X_1 Y_2 + X_2 Y_1 + X_2 Y_2$$

Here in the Figure 2, we can see what the algorithm proposes as a solution and what is actually optimal bond configuration. For MPOs, some kind of post-

processing is required to further optimise the bond connections. We've dealt it in the TTNO case in our own Combine-and-Match algorithm.

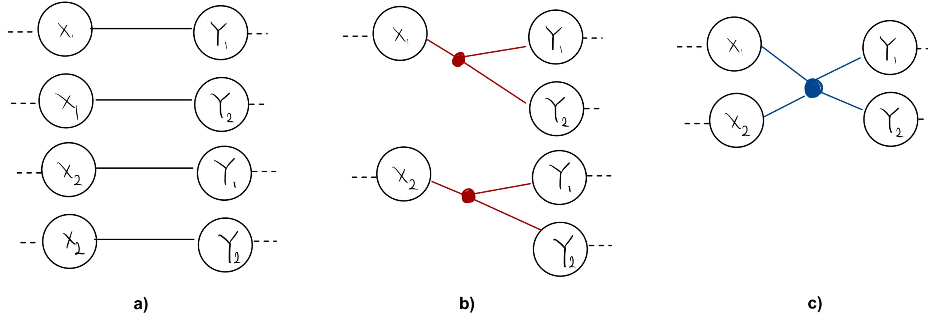


Figure 2: a) The initial configuration of the chains. b) The result of the bipartite-graph algorithm. c) Actual optimal solution.

4.2 Combine-and-Match Algorithm for TTNOs

In this section, we propose a new algorithm to construct the most optimal TTNOs, called "Combine-and-Match Algorithm". This algorithm is constructed upon the State Diagram representation by Milbradt et al. (2023) [3]. The same state diagram implementation is followed for the state diagrams during this paper. In the figure 3, we can see the representation for the hamiltonian, single terms and compound state diagram formed in the beginning of the algorithm.

4.2.1 Compound State Diagram Representation

Let's investigate further the representation with figure 3. Figure a shows the tree structure of the Hamiltonian. Numbers represent the sites and edges are the bonds between sites i.e. virtual bonds. Here one can see all terms somewhat combined over the sites with colors. The goal of the optimization is to minimize virtual bond dimensions between sites.

In the figure b we can see how the single terms constructed as state diagrams. This structure is not used in the current algorithm but it is good to show how terms in the Hamiltonian could represented individually. The important thing is the notation on the state diagrams. We call **Hyperedge** the square nodes in the state diagram which represents the sites and represented as nodes in the tree structure. On the other hand, we call **Vertex** the black dots in the state diagram corresponds

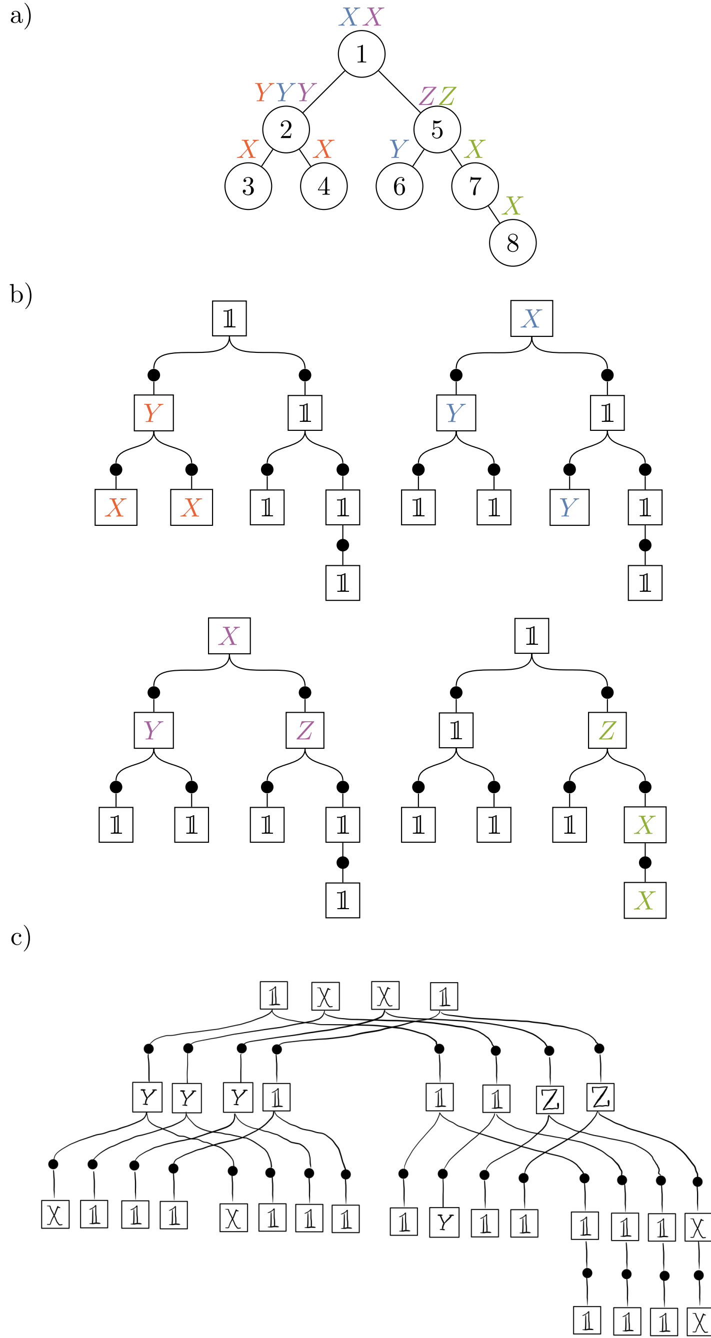


Figure 3: a) Tree structure for a given TTNO, where the Hamiltonian is $H_{\text{toy}} = \sum_{j=1}^{4} h_j = Y_2X_3X_4 + X_1Y_2Y_6 + X_1Y_2Z_5 + Z_5X_7X_8$. Each single site operator in the same term is given the same color and identity operators are not shown. We choose node 1 as the root of the tree structure. b) Single paths state diagrams for each term. c) Compound state diagram formed before the Combine-and-Match algorithm.

to virtual bonds and represented as edges in the tree structure. After this point, we call vertices and hyperedges correspondingly.

We apply Combine-and-Match algorithm to the compound state diagram that can be seen in figure c. This is the most straightforward composition of the Hamiltonian. Here we have maximum number of virtual bonds possible as you can see the number of vertices in the diagram. The algorithm traverses through this compound state diagram and optimises virtual bond dimensions one-by-one via minimizing the number of vertices between hyperedges. While optimising each vertex site, compound state diagram is also updated. In the end, we will have one of the state diagrams with optimal virtual bond dimensions.

4.2.2 Algorithm

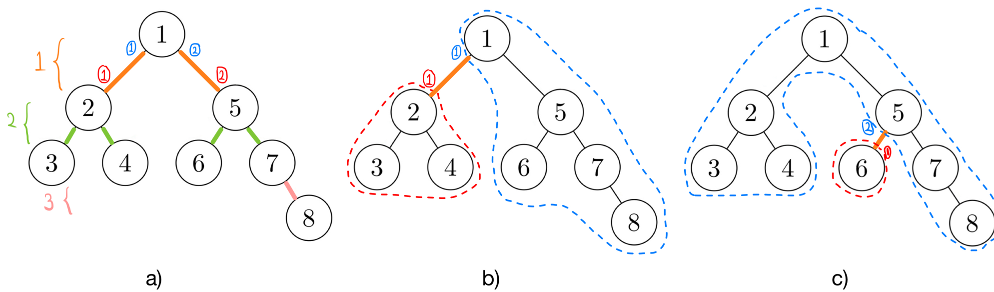


Figure 4: **a)** Tree is traversed in BFS manner but double passing each level. The order of the traverse is first the 1. level (orange), than the second level (green) and finally the third level (pink). For each layer, we process each vertex site twice. For example in the first layer, firstly vertices are processed in the red numbering order, then blue. **b)** For each vertex site, we split tree in two parts. Red part is *U-subtree* and blue part is *V-subtree*. **c)** Another example cut for the bond between site 5 and 6.

As previously mentioned, the algorithm operates by navigating through the compound state diagram, with the objective of optimizing the virtual bond dimensions. This is achieved by minimizing the number of vertices intersected by hyperedges. The navigation follows a Breadth-first search (BFS) strategy, initiating from the root and proceeding through the legs, which are the vertex sites, of the tree. We call processing of a vertex site as *Cut site*, as it is like we are cutting tree through the vertex site. A distinctive aspect of our approach is the dual traversal over each level: rather than processing each vertex site consecutively, the algorithm completes a

full pass through all vertex sites at a given level before embarking on a second pass. The traversal order is visually depicted in Figure 4 a), providing a clear illustration of the process.

Algorithm 1 Combine-and-Match Algorithm

Require: $hamiltonian, ref_tree$
Ensure: $compound_state_diagram$
1: $state_diagrams$ ← GET_STATE_DIAGRAMS($hamiltonian, ref_tree$)
2: $compound_state_diagram$ ← GET_STATE_DIAGRAM_COMPOUND($state_diagrams$)
3: Initialize $queue$
4: **for** each child of the root in ref_tree **do**
5:     Enqueue ($root, child$) pair into $queue$
6: **end for**
7: **while** $queue$ is not empty **do**
8:     $level_size$ ← size of $queue$
9:     **for** each ($parent, current_node$) pair in $queue$ **do**
10:         $local_hyperedges$ ← COPY of hyperedges of $current_node$ from $compound_state_diagram$
11:         COMBINE_U($local_hyperedges, parent$)
12:     **end for**
13:     **for** each item in $level_size$ **do**
14:         Dequeue ($parent, current_node$) pair from $queue$
15:         $local_vs$ ← COPY of hyperedges of $parent$ from $compound_state_diagram$
16:         COMBINE_V($local_vs, current_node, parent$)
17:         **for** each child of $current_node$ in ref_tree **do**
18:             Enqueue ($current_node, child$) pair into $queue$
19:         **end for**
20:     **end for**
21: **end while**
22: **return** $compound_state_diagram$

Through the algorithm, we split the tree in two parts: *U-subtrees* and *V-subtrees*. The naming U and V follows the bipartite algorithm discussed in the previous section. *U-subtree* is the subtree of a graph, including all children nodes and sub tree structure for a given node or lower node of a given Cut site. *V-subtree* is the remaining part of the tree after extracting the *U-subtree*. In the figure 4 parts b and c, a partition to *U-subtree* and *V-subtree* for the vertex site 1-2 and site 5-6 can be seen respectively.

The algorithm consist of two parts which corresponds: *Combining U-subtrees* and *Combining V-subtrees*. In the first pass of a level, we check and combine *U-subtrees* and for the second iteration, we focus on the *V-subtrees*. While combining U or V subtrees, the connections in the Cut site are handled, i.e. unnecessary vertices can be removed, new vertices can be introduced or connections of vertices can be altered. After passing a cut site twice, it is guaranteed that corresponding site is optimised and has optimal (minimal) bond dimensions.

The details of the algorithm can be seen in pseudocode of Algorithm 1. It can be summarised as iterating the tree BFS manner and apply Combine U and Combine V functions for cuts.

4.2.3 Combining U-subtrees

During the first iteration of a level in the tree, we go through each U-subtree and combine exact same subtrees. After the first iteration, we will end up with the unique U-subtrees. In the figure 5, the order of traverse and the resulting U-subtrees are shown.

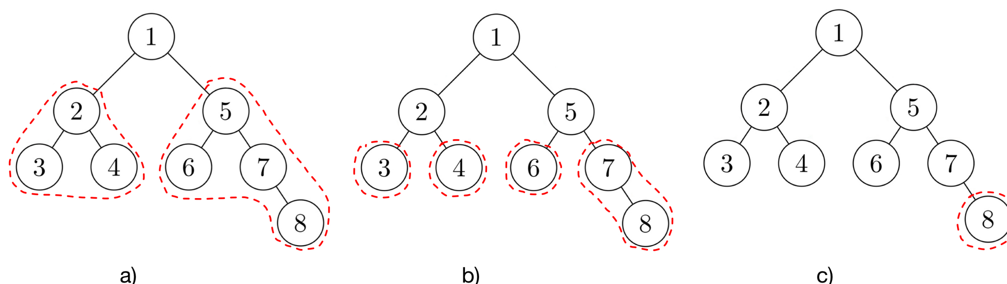


Figure 5: The traverse of U-subtrees for the first, second and third level. The resulting red dotted subtrees are unique set of subtrees

Let's further investigate how the *Combination of U-subtrees* is actually done. First we will go through the psuedocode Algorithm 2 and give some applied examples with a simple case. For this purpose we will use example compound state diagram given in Figure 3 and we will cut through site 1-2 and form U-subtrees. The application of this simple case is shown in the figure 6

Goal of the *U-subtree combination* is to detect exact same subtrees and combine them. The condition of equivalence is to have same local operations (e.g X, Y, Z or I) for all of the hyperedges corresponds to the same level. To detect equal subtrees, one have to go through each site and compare local operators of two subtrees. If all

of the local operations are the same, we can mark those two subtrees equal.

The Combine-and-Match algorithm employs a sophisticated and efficient technique to identify identical subtrees within the compound state diagram. Each hyperedge within the diagram is assigned a hash value, which acts as its identifier for the subtree below it. This hash is computed by integrating the hashes of the child nodes with the associated local operator. Consequently, if two hyperedges share identical hash values for their children and the same local operator, they will also have matching hash values. For leaf nodes, the hash is derived solely from the local operator. As the compound state diagram is constructed, these hash values are systematically calculated and linked to their respective hyperedges. Therefore, to determine the equivalence of two subtrees, one can simply compare the hash values of their hyperedges.

After finding two equivalent subtrees, we combine those. Combination process is simple, we remove the second subtree from the compound state diagram and combine vertices of the subtrees in the cut site. Let's follow the psuedocode 2 and call the first hyperedges of the subtrees as *element1* and *element2*, then we call vertices connected to those hyperedges in the cut site as *keep_vertex* and *del_vertex*, respectively. Also let us call the site below the cut site as *child_site* and above the cut site as *parent_site*. In the cut site, we are going to remove *del_vertex*, however we need to connect the hyperedges in the *parent_site* that are connected with the *del_vertex*, to the *keep_vertex*. This is practically combining those two vertices together. In the end, *keep_vertex* will be connected to all of the hyperedges previously *keep_vertex* and *del_vertex* connected.

Let's check Figure 6 for application of *Combine_U* function. Here cut site is site 1-2 shown with red dashed line. Each U-subtrees is colored differently in the part a) as red, green, orange and purple. When we check subtrees, we can detect that green and orange subtrees are equal, so we are going to combine those two. The resulting subtree is colored as blue in the part b). In the *child_site*, we just basically remove orange subtree from the diagram. In the *parent_site*, *keep_vertex* is connected to the first X, where *del_vertex* is connected to the second X hyperedge. So we remove the *del_vertex* from the cut site and connect *keep_vertex* with both of the X hyperedges.

It can be observed that the number of vertices in the cut site is decreased as 1 after the *Combine_U* operation. As it is stated before, this corresponds to the virtual bond dimension between sites 1 and 2. So we found a more optimal configuration for the given Hamiltonian. After applying *Combine_V* to the same site, we would reach optimal bond dimensions.

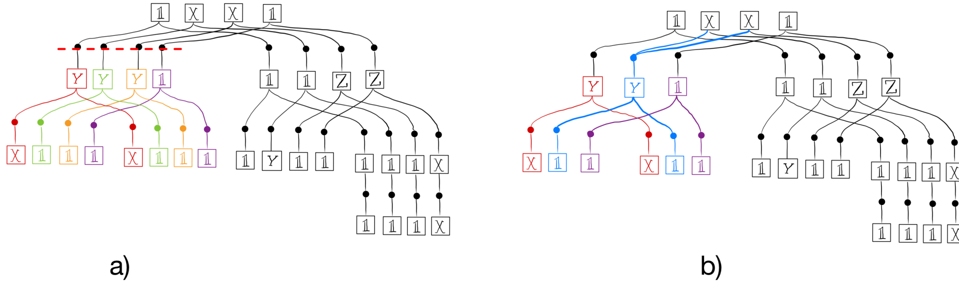


Figure 6: **a)** The compound state diagram before the *Combine_U* is applied. Each subtree is colored differently and green and orange U-subtrees should be combined. **b)** The resulting diagram after combination of U-subtrees.

Algorithm 2 Combine_U

Require: *local_hyperedges*
1: Initialize *combined* as an empty set
2: **for** each pair of hyperedges (*element*1, *element*2) in *local_hyperedges* **do**
3:     **if** elements already combined **then**
4:         Continue to next iteration
5:     **end if**
6:     **if** element1 and element2 combinable **then**
7:         Add j to *combined*
8:         *del_vertex* ← *element*2.find_vertex(*parent*)
9:         *keep_vertex* ← *element*1.find_vertex(*parent*)
10:         Erase subtree of element2 from compound state diagram
11:         *fathers* ← *del_vertex*.get_hyperedges_for_one_node_id(*parent*)
12:         Remove *del_vertex* from state diagram
13:         **for** each *father* in *fathers* **do**
14:             Remove *del_vertex* from *father*
15:             Add *father* to *keep_vertex*
16:         **end for**
17:     **end if**
18: **end for**

4.2.4 Combining V-subtrees

The algorithm's second phase focuses on identifying and merging V-subtrees. While this phase shares a similar objective and methodology with the U-subtrees segment, it encounters distinct scenarios necessitating meticulous attention. This section will go over the standard integration of V-subtrees, then examine unique

cases, providing a detailed exploration of the strategies employed to manage each scenario effectively.

Goal of the *V-subtree combination* is again to detect exact same subtrees and combine them. The condition of equivalence is same as before, however this time, we can utilize the resulting updates of the $Combine_U$ function on the diagram, as $combine_V$ function is applied after finishing the $Combine_U$ optimisations on the given level. In this regard, it can be realized that, to detect equivalence of V-subtrees, comparing first hyperedges of the V-subtrees (which are in the $parent_site$) and their connected vertices. If two of the hyperedges have same local operator and connected to the same subtree, then we can mark those V-subtrees as identical.

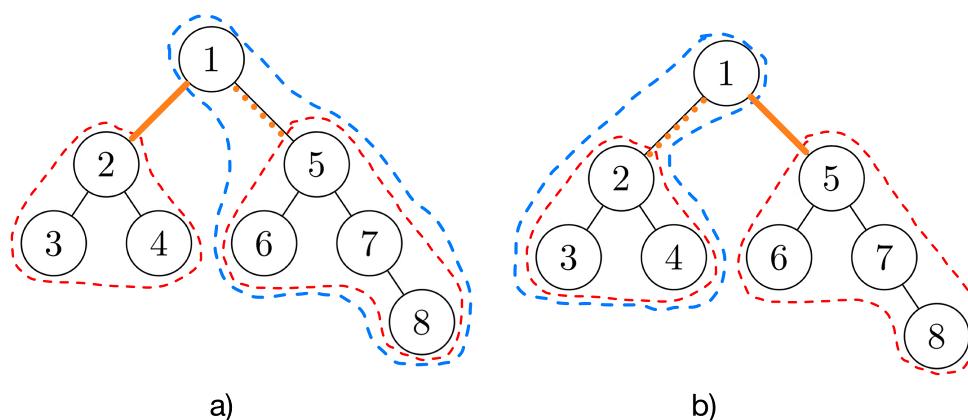


Figure 7: Red dashed lines are U-subtrees that are already optimised. Here they are already combined and the current compound state diagram only includes unique U-subtrees. Blue dashed lines represent V-subtrees. To detect identical V-subtrees, it is enough to check site 1 and orange dashed connection with U-subtrees. **a)** Cut through site 1-2. **b)** Cut through site 1-5.

Let's further investigate the proposal with Figure 7 and Figure 8. In the Figure 7, we are looking to the case for the root as the $parent_site$ of the cut site. As it can seen here, the V-subtrees are consist of site 1 hyperedges connected with U-subtrees. The red dashed U-subtrees have optimised and includes only unique subtrees. As it is shown in Figure 6, same U-subtrees are combined into same vertex in the cut site. So for the V-subtrees, only *not-unique* part is site 1 hyperedges and it is enough to check local operators on site 1. If two hyperedges have same local operator and connected to the same vertices, than they can be marked as equal and we can combine those.

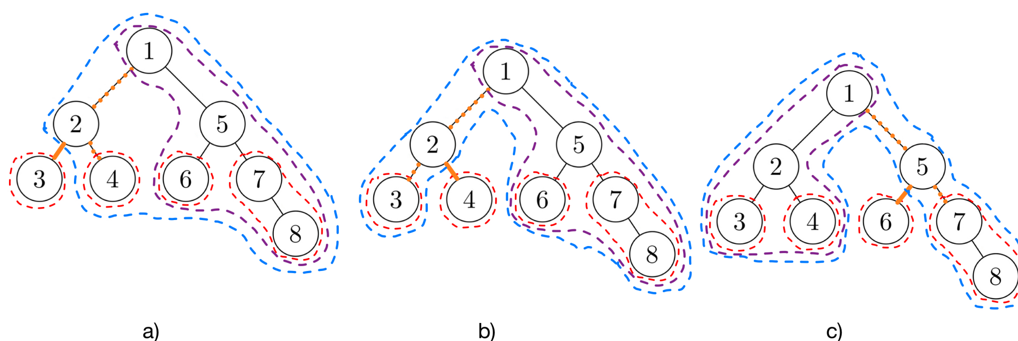


Figure 8: Red dashed lines are U-subtrees that are already optimised. Purple dashed lines are previously optimised V-subtrees. Here they are already combined and the current compound state diagram only includes unique U-subtrees and V-subtrees for the upper level. Blue dashed lines represent V-subtrees that currently being optimised. To detect identical V-subtrees, it is enough to check *parent_site* and orange dashed connections with U-subtrees and optimised V-subtrees. **a)** Cut through site 2-3. **b)** Cut through site 2-4. **c)** Cut through site 5-6.

In the Figure 8, we have more general case with *parent_site* as an arbitrary site. Here the V-subtrees are consist of *parent_site* hyperedges connected with U-subtrees and already optimised V-subtrees. The red dashed U-subtrees and purple dashed V-subtrees have optimised and includes only unique subtrees. So the same situation is valid, only *not-unique* part is *parent_site* hyperedges and it is enough to check local operators on site 1 and their connections with unique parts. If two hyperedges have same local operator and connected to the same vertices, than they can be marked as equal and we can combine those.

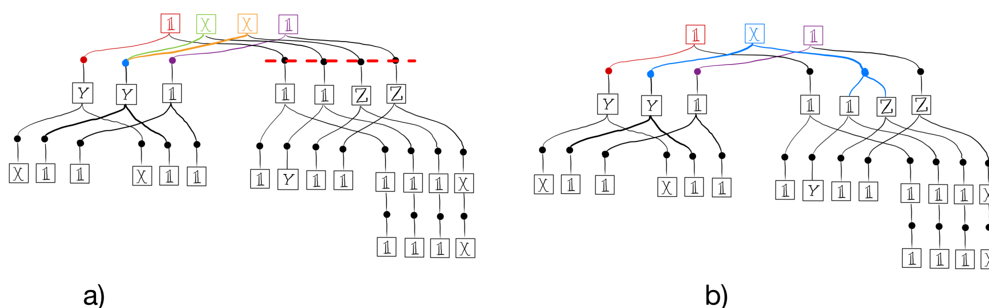


Figure 9: **a)** The compound state diagram before the *Combine_V* is applied. Each V-subtree is colored differently and green and orange V-subtrees should be combined. **b)** The resulting diagram after combination of V-subtrees shown in blue.

In the Figure 9, we can see how this comparison is done. In the part a), each hyperedge is colored differently. And after comparing each hyperedge with each other, it can be seen that green and orange X hyperedges have same local operation on the site 1 and connected to the same vertex to their U-subtrees. Then we can mark those two as identical and should combine those as shown in part b). Lets investigate how the combination is actually performed.

Before going into details, first investigate what is done in this function, while going over the psuedocode Algorithm 3. Here again we are going over a set of hyperedges (from *parent_site* of the cut) and try to detect combinable pairs of V-subtrees. After finding such a pair, we get important hyperedges and vertices for the combine operation and get the numbers of hyperedges in both V-subtrees in the *parent_site*. Using those numbers, we handle combination of V-subtrees in **three** different ways.

1. **Normal combination**: When both V-subtrees have only one hyperedge in the *parent_site* i.e. *del_vertex* and *keep_vertex* have only one hyperedge as parent.

2. **Generative combination**: When just one of the V-subtrees has only one hyperedge in the *parent_site* and other one has multiple i.e. one of the *del_vertex* and *keep_vertex* have only one hyperedge as parent and the other one has multiple. We called this one *Generative combination* as it requires creation of new hyperedge(s) in the *child_site* and new vertex in the *cut_site*.

3. **Fully-connected combination**: When both V-subtrees have multiple hyperedges in the *parent_site* i.e. *del_vertex* and *keep_vertex* have more than one hyperedges as parent. It is called *Fully-connected combination* as if the conditions met, it is required to create a fully-connected node in the *cut_site*.

After the combinations, we call the *Combine_V* function with updated set of vertices again if there happened any Generative combination; as generation of new vertices and hyperedges may cause new possible combinations in V-subtrees. Also after the creation of a fully-connected vertex, we call the *Combine_V* function directly finishing the initial iteration and start over with updated hyperedges. These recursive calls are required to reach global optimum.

Let's set a terminology before explaining combination. We follow the same naming with U-subtrees as we call the first hyperedges of the V-subtrees as *element1* and *element2*, then we call vertices connected to those hyperedges in the cut site as *keep_vertex* and *del_vertex*, respectively; the site below the cut site as *child_site* and

Algorithm 3 Combine_V

Require: $local_hyperedges$

1: Initialize $combined$ as an empty set
2: **for** each pair of elements ($element1, element2$) in $local_vs$ **do**
3:     **if** elements already combined **then**
4:         Continue to next pair
5:     **end if**
6:     **if** $element1, element2$ is combinable **then**
7:         keep_vertex, del_vertex ← vertices of $element1, element2$ in cut site
8:         $d1 \leftarrow$ # hyperedges $keep_vertex$ has in parent site > 1
9:         $d2 \leftarrow$ # hyperedges del_vertex has in parent site > 1
10:         **if** $d1$ and $d2$ **then** ▷ Fully-connected combination
11:             **if** $del_vertex, keep_vertex$ can't create fully connected node **then**
12:                 Continue to next pair
13:             **else**
14:                 Combine vertices, delete duplicates, create a fully connected node and call $Combine_V$ recursively with updated local_vs
15:                 **return**
16:             **end if**
17:         **end if**
18:         Add $element2$ to $combined$
19:         **if** not ($d1$ or $d2$) **then** ▷ Normal combination
20:             Combine vertices normally, remove $element2$ hyperedge
21:         **else** ▷ Generative combination
22:             **if** $d1$ **then**
23:                 Switch $element1$ and $element2$
24:             **end if**
25:             **for** each vertex in $element2$ **do**
26:                 **if** vertex is in cut site **then**
27:                     Delete vertex from collection, create new vertex and hyperedges. Form new connections corresponding to combination of hyperedges.
28:                 **else**
29:                     Remove hyperedge from vertex
30:                 **end if**
31:             **end for**
32:             Remove $element2$ hyperedge from diagram
33:         **end if**
34:     **end if**
35: **end for**
36: **if** $combined$ is not empty **then**
37:     Recursively call $Combine_V$ with updated $local_vs$
38: **end if**

above the cut site as *parent_site*. Additionally we will call hyperedges connected to the *del_vertex* and *keep_vertex* in the *child_site* as *del_sons* and *keep_sons* respectively.

Normal Combination

This is the most common case for the V-subtree combination. Precondition for this is that both of the *del_vertex* and *keep_vertex* has just one parent hyperedge in the *parent_site*.

Combining element1 and element2 is simply removing *element2* and combine *del_vertex* with *keep_vertex*. To remove *element2* from the compound state diagram, after deleting it, we also should iterate through vertices connected to *element2* (except the one in the cut-site) and remove their connection with it. For the cut site, we are going to remove *del_vertex*, however we need to connect the hyperedges in the *child_site* that are connected with the *del_vertex* (*del_sons*), to the *keep_vertex*). This is practically combining those two vertices together. In the end, *keep_vertex* will be connected to all of the hyperedges previously *keep_vertex* and *del_vertex* connected.

Let's check Figure 9 again for the application of *Combine_V* function. Here cut site is site 1-5 shown with red dashed line. Each V-subtrees is colored differently in the part a) as red, green, orange and purple. When we check subtrees, we can detect that green and orange subtrees are equal (as they have the same local operator and connected to the same vertex), so we are going to combine those two. The resulting subtree is colored as blue in the part b). In the *parent_site*, we just basically remove orange subtree from the diagram. In the *child_site*, *keep_vertex* is connected to the second I, where *del_vertex* is connected to the first Z hyperedge. So we remove the *del_vertex* from the cut site and connect *keep_vertex* with both of the I and Z hyperedges.

It can be observed that the number of vertices in the cut site is decreased as 1 after the *Combine_V* operation. As it is stated before, this corresponds to the virtual bond dimension between sites 1 and 5. So we found the most optimal bond dimension for the cut as we optimised U-subtrees and V-subtrees.

Generative Combination

Precondition for this is that one of the *del_vertex* and *keep_vertex* has one

parent hyperedge in the *parent_site* and the other one has multiple. This causes problem because we can not combine *del_vertex* and *keep_vertex* directly as it creates an invalid tree structure. If *keep_vertex* has more than one hyperedges in the *parent_site*, we switch *del_vertex* and *keep_vertex*. After this operation, we will always have *keep_vertex* having just one parent and *del_vertex* having multiple parents.

The solution for that is to modify *del_vertex*, such that combination is possible. Process is as follows: We duplicate the *del_sons* hyperedges (hyperedges that are connected to the *del_vertex* in the *child_site*) in the *child_site*. These copy hyperedges are connected to the same vertices except *cut site*. For the *cut site*, a new vertex is created. This new vertex is connected to the all of the hyperedges that *del_vertex* connected except *del_sons* and *element1*. The copy hyperedges will be connected to the new vertex in the *cut site*. Finally, the connections in the *del_vertex* to the *parent_site* will be removed except *element1*. This new vertex is basically extracted from the *del_vertex*. This sub-state allows as to combine *del_vertex* and *keep_vertex* directly as in the **Normal Combination**.

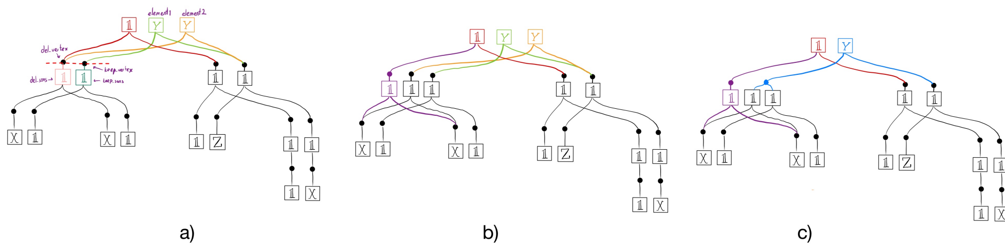


Figure 10: **a)** The compound state diagram before the *Combine_V* is applied. Each V-subtree is colored differently and green and orange V-subtrees should be combined. **b)** The sub-state diagram after the new hyperedge and new vertex is generated in the *cut site*. **c)** The resulting diagram after combination of V-subtrees, shown in blue.

Let's check Figure 10 for the application of *Combine_V* function. Here cut site is site 1-2 shown with red dashed line. Each V-subtrees is colored differently in the part a) as red, green and orange. When we check subtrees, we can detect that green and orange subtrees are equal (as they have the same local operator and connected to the same vertex on the site 5), so we are going to combine those two. However as it can be seen, *del_vertex* has 2 parent hyperedges. So we are going to duplicate pink I hyperedge and *del_vertex* as described in the previous paragraph. Resulting sub-state diagram can be shown in the part b). Purple hyperedge and the vertex is the copied ones. The copied vertex is connected to the purple hyperedge

and parents of the *del_vertex* except element1. In this state, we can combine element1 and element2 normally. The resulting subtree is colored as blue in the part c).

It can be observed that the number of vertices in the cut site stayed same after the *Combine_V* operation. Even though we are decreasing the number of hyperedges in the *parent_site*, we create a new hyperedge in the *child_site*. The operation seems unnecessary but actually experiments showed that these steps are crucial to have optimal tree after processing each cut through the tree.

Fully-connected Combination

Precondition for this case is that both of the *del_vertex* and *keep_vertex* have multiple parent hyperedge in the *parent_site*. This causes problem because we can not combine *del_vertex* and *keep_vertex* except the fully connected case. In this situation, we will investigate the diagram if it is possible to create a fully connected layer in the cut site. When it is not the case, we will skip combination as it is not feasible at all to combine.

Fully-connected vertex refers a vertex with multiple parents and multiple children. In the previous state diagram representation or in the bipartite graph theory, we only create vertices with one-to-many connections, i.e one child-multiple parents or one parent-multiple children. In this case, however, we will create a vertex with multiple children and parents when it is possible.

The process of detection of the fully connected case as follows: After determining *del_vertex* and *keep_vertex*, we iterate through hyperedges of the both vertices in the *parent_site*. For each parent hyperedge of the *del_vertex* in the *parent_site*, there should be a parent hyperedge of *keep_vertex*, that is combinable and vice versa. The creation of fully-connected vertex is possible if only this condition is met. Otherwise, the combination is skipped for these element1 and element2

The creation process is as follows: All hyperedges connected to the *del_vertex* on the *parent_site* will be removed from the compound state diagram with their connections to the other vertices. Also we remove *del_vertex* from the diagram and add *del_sons* hyperedges to the *keep_vertex*. This is basically combining *del_vertex* and *keep_vertex* as before. In the and we will have *keep_vertex* as a fully-connected vertex.

Let's check Figure 11 for the application of *Combine_V* function. Here cut site

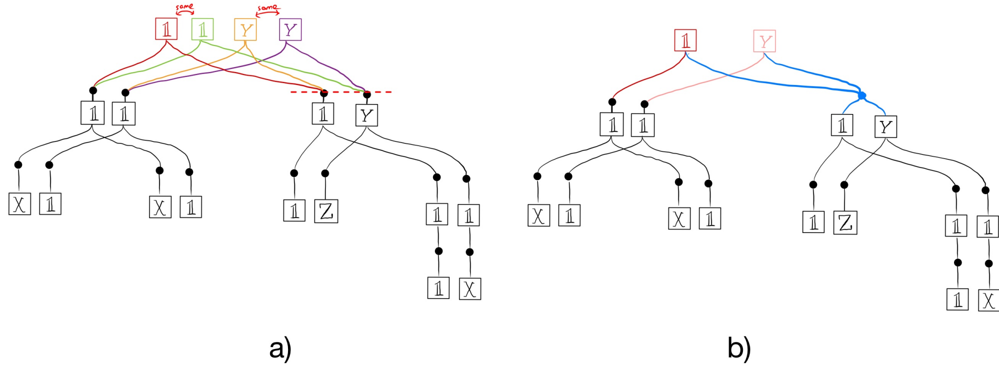


Figure 11: **a)** The compound state diagram before the *Combine_V* is applied. Each V-subtree is colored differently. **b)** The resulting diagram after combination of V-subtrees, and creation of fully-connected vertex, shown in blue.

is site 1-5 shown with red dashed line. Each V-subtrees is colored differently in the part a) as red, green, orange and purple. When we check subtrees, we can first detect that red and green subtrees are equal (as they have the same local operator and connected to the same vertex on the site 1), so we are going to combine those two. However as it can be seen, both *del_vertex* and *keep_vertex* have 2 parent hyperedges. So we are going to iterate through each parent hyperedges of *del_vertex* and *keep_vertex* and check compatibility as described. It can be seen that yellow and purple subtrees are also combinable which are parents of vertices in the cut site. In the part b) , we remove green and purple subtrees as they are connected to the *del_vertex* and combine *del_vertex* and *keep_vertex* together, which can be seen in blue color.

It can be observed that the number of vertices in the cut site is again decreased as 1 after the *Combine_V* operation. This additional check and combination makes it possible to find optimal bond dimensions in any cases.

5 Results and Evaluations

As it is stated in the previous sections, Combine-and-Match algorithm has reached optimum bond dimensions for any case and any tree structure, including having a leaf node as a root.

In the Figure 13, we are comparing the virtual bond dimensions that we found with the optimal bond dimension that can be achieved using singular value decompositions. To express the analysis more quantitatively, 300.000 different random

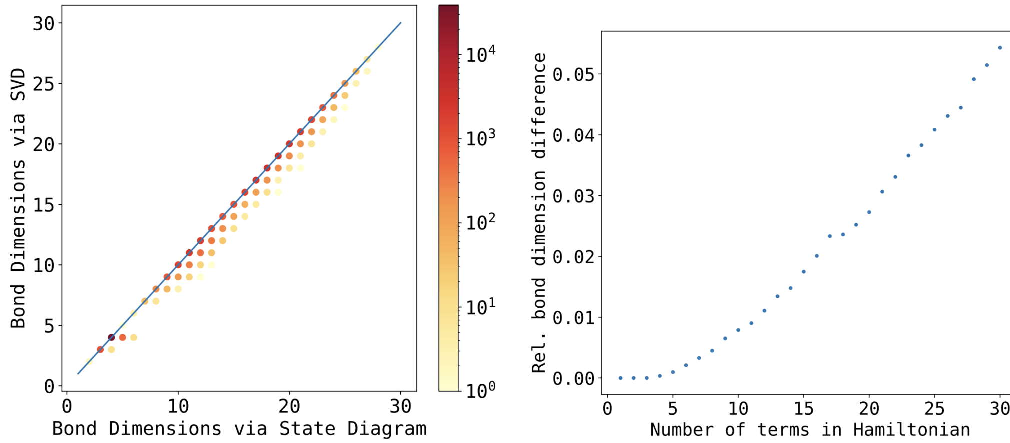


Figure 12: Results for the previous algorithm. As it can be seen, it is far from the optimal most of the cases.

Hamiltonians were generated and the Combine-and-Match algorithm were able find most optimal bond dimensions for all of the cases.

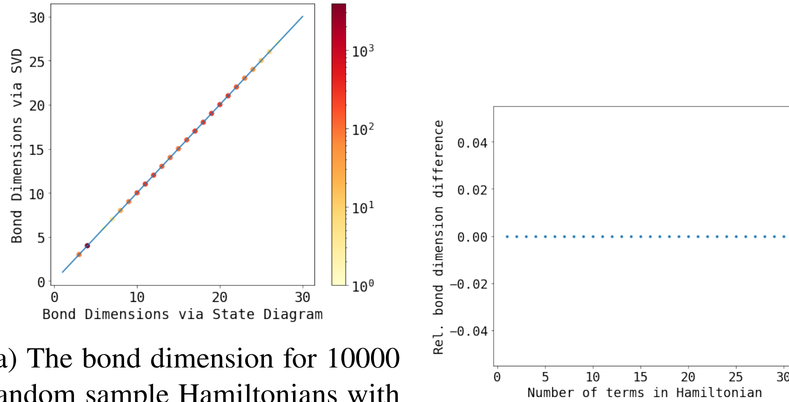
If we analyse the cases where a leaf node is determined as a root, we still have optimal bond dimensions for all of the cases; for 300.000 cases . For this specific setup, previous algorithm was failing and could not find optimal bond dimensions.

6 Discussions and Future Work

Throughout this research, it has been observed that the Combine-and-Match algorithm regularly achieves the theoretical optimum. Yet, it's important to note that these outcomes are derived solely from empirical testing. The algorithm was subjected to a broad array of randomized scenarios and exhibited success across all instances. To definitively assert the optimality of the algorithm, a thorough theoretical analysis and proof are essential. Only through such rigorous validation can we legitimately claim the algorithm's optimality.

An additional area for application involves extending the Combine-and-Match algorithm to the initial fermionic scenario. Given that each linear MPO structure can be conceptualized as a tree-like TTNO structure, this presents an opportunity to test and analyze the algorithm's practical implementation in real-world scenarios.

Enhancements to the algorithm could also include the integration of variable coefficients for individual terms. The current version of the algorithm operates



(a) The bond dimension for 10000 random sample Hamiltonians with 30 terms obtained via our algorithm versus the optimal (minimal) bond dimension based on singular value decompositions. A darker colour represents more of our sample bonds with the given relation of found to optimal bond dimension. The blue line shows y = x.

(b) The number of terms in the Hamiltonian against the average difference in bond dimension as obtained via our algorithm compared to the minimal dimension per bond.

Figure 13: Results for the Combine-and-Match algorithm. As it can be seen, We've reached optimal bond dimensions.

under the assumption that all coefficients are uniformly set to 1. However, as demonstrated in the linear case of the bipartite algorithm, accommodating term coefficients is indeed feasible. Extending this capability to Tree Tensor Network Operators stands as an essential step forward. By successfully implementing this feature, we could substantiate the claim of having a comprehensive algorithm for Tree Tensor Network Operations.

7 Conclusions

This study has presented a comprehensive investigation into the construction of Matrix Product Operators (MPOs) and Tree Tensor Network Operators (TTNOs) utilizing the principles of bipartite graph theory. Through rigorous analysis, we have confirmed and extended the findings of Ren et al. (2020), uncovering the limitations of the previously established algorithm in handling certain edge cases. Our endeavor to adapt these methodologies to the more complex domain of TTNOs required significant algorithmic innovation, facilitated by the application of state diagrams as proposed by Milbradt et al. (2023).

The outcomes of this research demonstrate that the developed Combine-and-Match algorithm is capable of determining the optimal bond dimensions for any tree structure, independently of Bipartite Graph Theory. Considering that linear MPO structures can be envisaged as analogous to tree-like TTNO structures, this achievement also signifies an enhancement over the algorithm proposed by Ren et al. (2020).

This work's implications extend beyond the technical accomplishments, potentially influencing a range of applications within quantum chemistry and physics. The enhanced understanding and optimization of tensor network operators may contribute to the development of more powerful quantum simulation tools, furthering our capacity to probe the subtleties of quantum mechanics and its real-world applications.

In conclusion, this guided research project not only fulfills an academic pursuit but also marks a substantive stride towards the advancement of quantum computational methods. The journey from conceptual understanding to practical application encapsulates the essence of research in computational science, signifying the iterative nature of discovery and the perpetual quest for improvement and innovation.

References

[1] Jiajun Ren, Weitang Li, Tong Jiang, and Zhigang Shuai. A general automatic method for optimal construction of matrix product operators using bipartite graph theory. *The Journal of Chemical Physics*, 153(8), August 2020.

[2] Garnet Kin-Lic Chan, Anna Keselman, Naoki Nakatani, Zhendong Li, and Steven R. White. Matrix product operators, matrix product states, and ab initio density matrix renormalization group algorithms, 2016.

[3] Richard M. Milbradt, Qunsheng Huang, and Christian B. Mendl. State diagrams to determine tree tensor network operators, 2024.

[4] Naoki Nakatani and Garnet Kin-Lic Chan. Efficient tree tensor network states (ttns) for quantum chemistry: Generalizations of the density matrix renormalization group algorithm. *The Journal of Chemical Physics*, 138(13), April 2013.