The TARANG Code Generator – SURP Project Report

Prof. Achintya Kumar Dutta (Guide)
Arpon Basu (Mentee)

Email
arpon.basu@gmail.com

Contents

Abstract

This report presents and summarises the work done by the author (mentee) in making a code generator, **TARANG**[1], for translating LateX expressions of tensors into python code (consisting of optimized intermediates, built-in timer features, memory deallocation, appropriate renaming of tensors and many other features), as a smaller sub-project in the larger software project, **BAGH**. This report will elaborate upon the methods used to achieve the above, the sources referred to and the future scope of work.

1 Introduction

The code generator **TARANG**[1] that will be discussed in this report is a multifaceted software with many features and nuances, and thus to fully appreciate every detail of what we have tried to achieve, we must first try to understand what the objectives of this project were.

Basically, we start off with a text file containing uniformly spaced LateX expressions (examples of which are given in the “Test Text Files" in the repository here [1]), which themselves are generated from another set of python files and text file generators in the project **BAGH**, which will encompass this code generator (and multiple others) in it’s ambit. These expressions are each basically up-to 5 tensors contracted with each other, and then these contractions are added and/or subtracted to each other after possibly dividing them with some coefficient (say 2 or 4).

The first preliminary part of our code generator[1] then becomes to render those tensor contractions in the form of a python code, using the **einsum function from the numpy module**[2], which is a special function designed especially for carrying out tensor operations of the kind that usually arise, say in CCSD methods in quantum chemistry (as in our case)[9], or general relativity in physics, and in many more places.

Now, a basic overview of the einsum function[2] quickly reveals that one needs to supply to it the list of indices and tensors, and those too in some specific order for it to do it’s job.

Thus, two python parsing codes are written which use the versatile list splicing facilities within python’s standard functionalities itself to convert the text file into a python code, both representing the same tensor expression.

We then come to the next part of our project, which is perhaps the most important, that of **factorizing** a tensor expression : What does this mean? As the total number of distinct indices (including duplicate ones which don’t appear in the final contraction) increases in a tensor expression, the **computational cost** of evaluating it becomes very high, and this drags the overall performance down. Thus, what is usually done is that tensor contractions, instead of directly summing them up in one go, are broken down into smaller tensor expressions called **intermediates**, which are much less expensive to compute, and the resulting intermediates are reassembled back together, and it usually happens that a significant cost overhead can be avoided through this route, and thus generation of intermediates becomes of paramount importance. How exactly we factorized is described in the sections below[3].

Conceptually speaking, this roughly sums up the content of the project (though many more smaller but no less important details are to be described below), and also describes a massive progress in terms of the **BAGH** framework : From having just a Hamiltonian of a molecule (from which we obtain our LateX text files) from the “second quantisation", to actually having a semi-optimized python code to calculate within reasonable time the actual expressions, we have certainly come a long way. Why do I say semi-optimized though? Because it so happens that many tensor expressions have many **common intermediates**, and thus the obvious way to optimize the code even further is to find out the common intermediates, store their values and then use them repeatedly (saving a lot of time in repeated calculations of the same intermediates).

This would then perfectly wrap up the entire scheme of things we sought to do : From having just a Hamiltonian from the fundamental principles of quantum chemistry to an optimized code to actually execute the expensive calculations to help model and simulate large atoms and molecules. The wavefunctions, the investigation of whose mysteries is the underlying theme of all these code generators, also prompted the author to give the name **TARANG** to the code generator, which in *Sanskrit* means “waves".

The exact details of how these steps were taken, and what remains to be done, come further down in the report.

2 Brief Overview

The main literature review for this project had to be done for the factorization of the expressions, for which we reviewed these papers [3][6][7][8], which provided algorithms for factorization from various points of views and utilities, which made us aware of the different constraints that had to be negotiated. For example, throughout this project and it’s implementation, we have mostly worried about the time-efficiency of our programs, but not so much about the memory aspect of it, which is also dealt with in these papers[3][6][7][8]. There were also the issues of what platform to base our code in (as per my guide Prof. Achintya Kumar Dutta’s advice, it was in python and C++ only), with multiple options such as MATLAB and python being available, and the slightly different types of expressions that arise in quantum physics and quantum chemistry[8], and thus a difference in approaches in dealing with them.

Even when the same goal is to be achieved, there are different ways to go about it, as a brief glance through these papers reveals. In the end, the algorithm we actually chose to implement factorization of algorithms[3] was in this paper. The algorithm itself will be discussed below.

3 Elaborate Implementation Details

We will now explain in detail what the codes in our projects do. Note that how to actually use these programs, and their dependency related issues have been addressed at length in the README of our repository[1], and thus will not be discussed again here.

3.1 Raw Code Generator

Coming to the first part of our project, which just involves the translation of the LateX code into raw python code, it was achieved in 2 files, **singleexpressionconversion.py** and **fullexpressionparsing.py**, which can be found here[1] (the repository README provides suitable instructions on how to use it). How exactly do these codes work? Firstly, in the fullexpressionparsing.py file, the incoming text file is broken down into it's individual tensor contractions by using a regex split along the '+/-' signs. The signs are then obviously assigned to the corresponding contractions when the python code is written later (there is a slight nuance in this : Note that if the first term is positive, then one doesn't have it's sign in the file : because obviously (x+y) is written as “x + y" instead of “+ x + y". Thus it has to be prepended to the list of signs). Then comes the actual job of generating an einsum command out of a given expression. This job is done by a function implemented in singleexpressionconversion.py which is then imported in the fullexpressionparsing.py file.

Coming to the job of generating the einsum command from an expression, it involves many sub-tasks to be done together to achieve it properly : Coming to the most complicated task among them, is generating the **permutation terms** (How does a permutation term work? Given a contraction of tensors with indices i_1, i_2, ..., i_n forming an expression say $f(i_1, i_2, ..., i_n)$, then, $f(i_1, i_2, ..., i_n)\mathrm{P}(i_1, i_2) = f(i_1, i_2, ..., i_n) - f(i_2, i_1, ..., i_n)$) from a term like say “t{ac}_{ik} v{bk}_{jc} P(ab) P(ij)" which actually represents (mathematically) $t^{ac}_{ik}v^{bk}_{jc} - t^{bc}_{ik}v^{ak}_{jc} - t^{ac}_{jk}v^{bk}_{ic} + t^{bc}_{jk}v^{ak}_{ic}$, ie:- the term above is a summation (signs included) of 4 terms. Keeping this requirement in mind, we choose a list of lists to be our data structure for the program : Every “indivisible term" (that is, terms which have no permutations in them) has a coefficient associated with it, which includes the sign of the term (+ if addition, - if subtraction) as well as any denominator it might have from fractional terms (as mentioned in the Introduction), and these two terms are stored in a list (it really is just a pair : this list always has exactly 2 terms). Thus, $\frac{t^{ac}_{ij}v^{kl}_{cd}}{2}$ will be stored as “[t{ac}_{ij} v{kl}_{cd}, 2.0]" if it's *to be added*, and as “[t{ac}_{ij} v{kl}_{cd}, -2.0]" if it's *to be subtracted*. Finally, for terms like “t{ac}_{ik} v{bk}_{jc} P(ab), we split them up into multiple parts (2 if there is a single permutation, $2^2 = 4$ if there are 2 permutations, using the function **biPerm(exp)**, aided by **uniPerm(exp)**), and each part generates an expression-coefficient list of it's own, and finally all those lists are stored in another list, making our data structure a list of lists. For the sake of consistency, even the lists of indivisible terms are converted into lists of lists. Thus, for example, $\frac{t^{ac}_{ij} v^{kl}_{cd}}{2}$ would actually be "[[t{ac}_{ij} v{kl}_{cd}, 2.0]]", while $-\frac{t^{ac}_{ij} t^{bd}_{kl} v^{kl}_{cd} P(ab)}{2}$ would be represented as "[[t{ac}_{ij} t{bd}_{kl} v{kl}_{cd}, -2.0], [t{bc}_{ij} t{ad}_{kl} v{kl}_{cd}, 2.0]]", and so on.

Thus, as is clear from above examples, a major portion of singleexpressionconversion.py is spent in the generation of 2/4 terms from single/double permutation containing terms. **Note that we have only implemented provisions for permuting up to 2 different pairs of indices, thus things like P(ab) P(ij) P(cd), for example, won't be rendered properly : the need for such terms however, hasn't arisen in any of our testings so far**.

After the breakdown of every term into a pair (list) of expressions and numbers is achieved, one must generate einsum commands of the sort "np.einsum(' ci, dj, abcd → abij',t1,t1,vvvv)" from those expressions. This involves two steps, extraction of indices from tensors and their contraction to obtain the indices of the final result, as well as the **re**naming of tensors from t to t1, or v to oovv, etc. The former is achieved by efficiently using python string splicing facilities to scoop out the indices from between the curly braces (note that first the upper and lower indices are collated together, ie;- {ab}_{ij} becomes {abij}), and then the actual string in single quotes (the format in which einsum requires it to be) is generated in the function called generateEinsumString (indlist), which just adds all the strings together, removes duplicate indices (as is the Einstein summation convention) and then finally alphabetically sorts the result string (this is more of a convention to make the answer strings more legible to human chemistry literature).

The renaming of tensors is follows some set of pre-fixed rules for each type of tensor (implemented in the **nameTensors** function) : For example t tensors of 2 dimensions are t1, while t tensors of 4 dimensions are t2, similarly r1 and r2 for 1 and 3 dimensional r tensors. For f and v the naming procedure is more complicated, and is implemented in the **transcribe** function (for example, a f tensor associated with the indices "kl" will be renamed fii, or a v tensor associated with "klcd" will be renamed oovv, and so on). In case the user introduces any new tensors, she must write it's naming convention in the nameTensors function, or the tensor will be represented in a similar manner everywhere.

Finally, with all the ingredients for writing a numpy command from a Latex instruction in place, the functions numpyStringBasic (for indivisible terms) and numpyString (for permutation containing terms) render those expressions into

numpy commands : Note that the expressions are always *assumed to be added*, and thus if something is to be actually subtracted, *it's negative is added, by prepending a (-1)* to it's numpy string*. This helps maintain the *locality of expressions*, wherein we don't have to worry about the sign of the neighbouring expressions while writing the string for a certain expression.
Coming back to fullexpressionparsing.py, all these strings (generated by the imported function) are collated together suitably for each term, and then the final string is **written** in a file that's **created from within the code itself**, after some due final processing (like addition of the line "import numpy" at the top, addition of the expressions being rendered in comments (#s in case of python) above their actual numpy commands themselves, and so on, to finally yield a python code created within the same directory in which fullexpressionparsing.py and singleexpressionconversion.py are located.

3.2 Factorization of Terms

We now come to the main algorithmic part of the program, namely the factorization of terms to produce intermediates, as was mentioned in the Introduction. The algorithm we have referred to for this [3] (Pg6) makes some very important assumptions which must be mentioned : Firstly, in the process of forming intermediates by clubbing together some terms from our expressions, one is free, in general to club as many terms as one wishes to make intermediates : For example, given a term $t^a_k t^b_l t^c_i t^d_j v^{kl}_{cd}$, one could factorize it as I1 = $t^a_k t^b_l = g^{ab}_{kl}$, I2 = $t^c_i t^d_j v^{kl}_{cd} = h^{kl}_{ij}$ and I3 = $g^{ab}_{kl} h^{kl}_{ij} = p^{ab}_{ij}$, or one could also factorize it as I1 = $t^a_k t^b_l = g^{ab}_{kl}$, I2 = $t^c_i t^d_j = m^{cd}_{ij}$, I3 = $m^{cd}_{ij} v^{kl}_{cd} = h^{kl}_{ij}$ and I4 = $h^{kl}_{ij} g^{ab}_{kl} = p^{ab}_{ij}$, ie:- while factorizing, any number of terms could be clubbed together, **but in our algorithm we will always club exactly 2 terms at a time** : Why? Firstly, because that's what our source from literature recommends, but also because any intermediate formed by clubbing $n > 2$ terms can itself further be broken down into $n - 1$ intermediates of two terms each, and thus this assumption doesn't lead to any loss of generality.
The paper also suggests another step for optimization, namely that any two tensors with the exact same index lists should be contracted together immediately : What this means is that those two tensors contract to **form a scalar**. For our testing purposes, we never encountered such tensors anywhere, and consequently, with the guide's suggestion too, we **excluded that possibility from our program**, ie:- if any tensor contraction contains any two terms which can contract to form a scalar, our program gives a segmentation fault (that error has been handled gracefully in the Factorizing program within the Intermediates folder. In the Factorizing program in the main repository (not within any folder), however, it has been left as it is, not least because that program, although visible, is not intended to be used

directly by a user, and consequently thus, if a user does use it , it's assumed that she will avoid such expressions any sub-part of which can contract to a scalar.
Now coming to the algorithm itself, what it suggests is that one takes the vector of index-lists (say {ak,bl,ci,dj,klcd}), starts clubbing any two index-lists at a time (say ci and klcd), replacing them in the original list with their condensate (ie:- the concatenation of two strings which removes any index common to both of them and then finally sorts the resulting string in an alphabetical order. The condensate of ci and klcd, for example, is dikl. Or, for example, the condensate of two identical strings is the null string. This is implemented in the **condense** function in Factorization.cpp), thus making our new **vector** {ak,bl,dj,dikl}, and this process is continued until a vector of length one is obtained (in this case, {abij}).
Now coming to the optimization part of things, it's apparent that given an initial vector of index-lists, there are many different ways to to condense it down to one index-list. How, then, do we compare between 2 different **traversal routes**, from the initial vector to the final common index-list?
At this point, we introduce the concept of **cost** of a string condensation : The cost of condensing 2 strings is basically the time complexity of contraction of two tensors whose indices are represented by those strings. Note however, that this might lead one to believe that only the duplicate indices that get summed over during a tensor contraction matter in the calculation of cost. However, it is not so, because even the indices which survive to the end of a tensor contraction are *eventually summed up at the end of calculations,* ***at a global level****, when the results from all the tensor contractions in a file are collected together*. Thus, instead of leaving the book-keeping for those indices when their summation actually happens, we account for them **at a local level itself**. The calculations themselves then, are not so difficult : For example, suppose we want to find the cost of contraction of klcd and cdij. Then we see that the list of all the indices in them are {'c', 'd', 'k', 'l', 'i', 'j'}. We then **set up a boundary** : Any index smaller than 'g' in the ASCII code is classified as representing a virtual orbital ('V' type variable), while any index larger than 'g' represents an occupied orbital ('O' type variable). Finally, the cost then, in terms of the **big-O notation**, is O($O^{n_o}V^{n_v}$), where n_o and n_v are the total number of 'O' type and 'V' type indices respectively. Thus, the cost of contracting klcd and cdij would be O(O^4V^2).
Before continuing with the optimization part, I would quickly like to digress to mention an important point : as my guide pointed out, one may need more variables for 'O' type and 'V' type indices than can be provided for by smaller-case alphabets alone. Note however, since the boundary is 'g', **any capital letter can represent a 'V' type index since capital letters are smaller than small-case letters in the ASCII code**. Thus, a shortage of indices can be potentially stemmed in this way (note that we have more number of 'O' type indices within the smaller-case alphabets themselves than 'V' type indices).

Continuing on with the optimization part, we now have to figure out how to compare two steps with different time complexities and assign one of them to be a more costlier operation. We are aided in this by the practical observation in this field, which basically says **V > O**, ie:- the number of virtual orbitals is usually much more than the number of occupied orbitals. Aided with this knowledge, an ordering on O($O^o V^v$) becomes obvious :

- O($O^{o_1} V^{v_1}$) > O($O^{o_2} V^{v_2}$) if $(o_1 + v_1) > (o_2 + v_2)$.
- If $(o_1 + v_1) = (o_2 + v_2)$, then O($O^{o_1} V^{v_1}$) > O($O^{o_2} V^{v_2}$) if $v_1 > v_2$.

Note that the above procedure essentially defines a total ordering on $(o, v) \in \mathbb{W}^2$, where $\mathbb{W} := \{0, 1, 2, ...\}$, and thus to make our lives easier, we use the **Cantor Hash Function**[4], an order-preserving bijective map $\mathcal{F}$ from $\mathbb{W}^2 \to \mathbb{W}$, $\mathcal{F} : (o, v) \mapsto (\frac{1}{2}(o + v)(o + v + 1) + v)$, defined in Factorization.cpp as the **CantorHash** function (and the **reverseCantorHash** function as it's inverse map), to compare the time complexities of two operations by mapping them to whole numbers and comparing them instead (because comparing two whole numbers is much easier than overloading the comparison operator for a C++ STL datatype pair<int,int>), and finally mapping those whole numbers back to our pairs when needed. Thus, if somebody wants to change how the ordering works, then possibly the CantorHash functions have to be removed, and operator overloading will remain the only way.
Now, consider the **traversal route (a possible way of factorization, 2 terms at a time)** {ak,bl,ci,dj,klcd} → {ak,bl,ci,cjkl} → {ak,bl,ijkl} → {ak,bijk} → {abij}. How does one compute the cost of this entire route? We evaluate the costs of every step (we already have defined the cost function for two strings earlier), every reduction of the vector leading to a decrease in size of one, **and take the maximum of all those costs, because that's what it'll take to reduce our vectors of indlists** (remember that in the big-O notation, all the terms lesser than the dominant one are insignificant). We can thus, associate every traversal route with a unique cost.
Finally, coming to the main objective of Factorization, which is to generate the **cheapest** (in terms of cost) traversal routes (note that the generation of intermediates naturally leads to the concept of traversal routes, and vice versa) given a vector of index lists. But for that, we need to generate all possible traversal routes that the vector can have[3] : Which is, if you realize, making a choice of two indices between 0 and vec.size() - 1 (these indices refer to 2 strings which are condensed in that step of the traversal) *for all* vecs that will arise in our traversal. But if we start off with, say a vector of size 5, as in this case ({ak,bl,ci,dj,klcd}), then one will have vectors of sizes 5 → 2 in the traversal, and at every stage we'll have $\binom{s}{2}$ possible ways to go forward, where s is the size of the vector we're talking about

$(2 \leq s \leq 5)$. Thus, given an initial vector size of n, **we have a total of** $\prod_{s=2}^{n} \binom{s}{2}$ traversal routes, and each traversal route is represented by an element in the Cartesian product of the sets $\mathcal{P}_s$ for $n \geq s \leq 2$, where $\mathcal{P}_s := \{(i, j)|0 \leq i < j \leq s - 1\}$. Now, with the help of robust libraries in the C++ STL the sets (actually vectors in the C++ implementation) $\mathcal{P}_s$ are easily constructed by the functions **makePairs** and **pairsTilln**, and their Cartesian Product is taken by the function **cartesianProduct** (obtained from [5]). That generates for us the entire “script" that is to be followed in traversing all possible traversal routes, a script that is enacted in the function **traverse**, which takes one by one the elements generated by cartesianProduct, calculates the cost of that traversal route, and stores both the traversal and it's cost in a vector format in the function's return value.

Finally, another function **keepMinimum**, sorts through the “traverseLog" generated by the traverse function (which is a vector of pairs of traversals and their corresponding costs), and discards all traversals but the ones with the minimum cost, and that's what the Factorization program then yields to us : For any vector of index lists such as {ak,bl,ci,dj,klcd}, it returns to us the cheapest possible factorization(s) that could be done of that vector.

Our repository[1] contains the file for the reader's perusal and usage (with a README to guide for the same). The author, though, would strongly recommend directly using the final “product", the **TARANG** code generator itself.

3.3 The TARANG Code Generator

We finally arrive to discussing what is the main output of this project, the code generator itself. Needless to say, it's usage is described here[1].

What does the Tarang code generator do exactly? It takes a text file as input (just like the raw code generator), but instead of outputting raw code, it outputs code which involves optimized intermediates, which of course, involves the Factorization program. Thus, in a way, massive parts of the code remain same from earlier, but what changes are some crucial extra features that impart to the code generator it's final form.

Talking about the Factorization part of the code generator, the file itself changes very little : Just to include the factorization program in the larger multi-file program, the initial Factorization.cpp is now split into IntermediateFactorization.cpp and IntermediateFactorization.h, with the former containing all the implementations while the latter contains the function definitions and their documentation. The only change, per se, is that as pointed out earlier too, tensors with exact same index lists can give segmentation faults, and now a few extra lines of code have been added to warn the user about the source of the error and exit gracefully instead of crashing.

A far more important point about Factorization is this : As hinted at earlier too, among the final steps of the completion of the entire code generator **BAGH** would be sorting out similar intermediates and optimizing the code even further. Now, partially due to a lack of literature which coherently outlines an algorithm for intermediate identification (most just leave it at a heuristic stage), and partially due to project constraints, the last step of optimization was *not* performed. Some nuances regarding that have been mentioned in the future scope of work.

Thus, among the many possible ways in which intermediates could be generated (What do I mean by this? Suppose I have 3 terms in my Latex text file, each of which can be traversed through, in the cheapest possible way only, in 1, 9 and 3 ways respectively. Then we have $1 \times 9 \times 3 = 27$ different ways of generating an “intermediate file", which, rather misnomerly, is the final product of our code generator), our code generator shows the user **one** among them, for representation purposes.

Besides the aforementioned point of showing the user only of the many possible “intermediate files", the code generator’s factorization part is mostly similar to the standalone version discussed in the previous subsection.

Now, coming to the part of how intermediate containing python code is actually generated, it further consists of 3 files : FullFactorized.py + SingleFactorized.py (just like the raw code generator), and FactorizedFileGeneration.cpp, which is a new file written for the code generator only.

The form and function of FullFactorized.py is almost 80% similar to it’s raw code generator counterpart : which is, taking as input text files, but instead of rendering them into python code as earlier, it’s converted into another text file which acts as an input to FactorizedFileGeneration.cpp, which finally renders the text file into the python code containing intermediates for every term.

One of the only major differences between FullFactorized.py (which imports all of it’s custom functionalities from SingleFactorized.py) and fullexpressionparsing.py is the shuffling feature in SingleFactorized.py : According to my guide’s advice, even though tensor contractions are commutative, it’s often helpful to have the tensors contracted in a certain order : more specifically, $f > v > t > r > l$. Thus, in SingleFactorized.py, with the help of 3 functions (**charHash, reversecharHash and sortShuffle**), the tensors are first mapped to integers (otherwise sorting them is not possible because the order is clearly not lexicological), sorted and then mapped back to their original names, and the sorting is done over the zipped list of tensors and their indices (otherwise there’ll be scrambling). **Thus, any addition of a new tensor (with a custom naming and priority requirements) requires an update of the nameTensors, charHash and reversecharHash functions. Moreover, if that tensor is not ahead of t in priority, it’s advised that the user also change the writeCode function in the FactorizedFileGeneration.cpp file suitably, especially lines #132 and #154. It currently accommodates tensors till l**. Apart from this, FullFactorized.py only has a few implementation related

details regarding the generation of the aforementioned text files, but nothing more of interest.
Coming to FactorizedFileGeneration.cpp, the style of code generation is quite similar to that of the raw code generator. For every term, the sequential traversal of it's intermediates is represented through successive numpy commands. In addition to this, the file also provide for a **timer feature** within the python file it generates : The timer is a switchable feature (you can switch the timer off if you want to) which prints the time taken by all the intermediates **which take more than 5 seconds to run, thus giving the user an insight into which terms are taking up most of the computational resources**. Another very important feature that has been implemented is that of **memory deallocation** in the python code : Note that although automatic garbage collectors do exist within the python language, they would free an intermediate only when the program draws to an end. Till then, as more and more intermediates are produced, they start to consume a lot of unnecessary memory. Thus, any operation that's required to be done with the intermediates is done almost immediately after their creation, and then those intermediates are explicitly assigned to 'None' in python, which frees all the memory stored inside those tensors, thus performing efficient memory management. Apart from this, the file outputs a python code shifted away by a tab, owing to issues of integrating it within the larger python code of the **BAGH** framework. Finally, since all of the file talked about earlier are part of a multi-file system and getting them to produce the desired results requires some coordinated action so that all the files get the right input, the entire process has been written down in a bashscript file, **ifg.sh**, which basically takes text file input from the user, and runs some command line arguments within itself to produce the desired python code in the same directory, thus making the interface much more accessible for the user. In effect thus, the bashscript file **acts as the director of a multi-file project where each of the files is an actor assigned to do a specific job**.

4 Conclusions

Thus, the objective of the code generator, which was to generate a code containing the intermediates for every term of the LateX text file supplied, suitably accompanied by various other features such as timers, memory managers and helpful comments, has been accomplished.

5 Future Scope of Work

Thus, after this extensive elaboration of the project details, we conclude that one of the most important and toughest parts of the project was Factorization, and consequently, any developer looking to build upon this work must pay most of her attention on optimizing the intermediate code. The main difficulty that arises in that process, is that many tensor contractions have > 1 possible cheapest contractions (for example, each term in the text file t2.txt[1] (in the folder Test Text Files) had on an average 1.27 possible traversal routes which were the cheapest), and in moderate sized files which contain approximately 30 - 40 terms, that can lead to up-to 15,000 (= 1.27^{40}) different possibilities for an intermediate generated file, and thus the developer will have to somehow parse through all of these possibilities and choose the case in which maximum number of intermediates were elided away. Moreover, they will have to do so without resorting to storing all of the possibilities, because after a point of time storing it takes a huge memory overhead, as the author himself experienced. Instead, the possibilities would have to be generated “on the fly", and would somehow have to be disposed of with once it is determined that there exist intermediate codes with a **compression factor** better than them.
The author would also like to point out now that the implementation of the traverse function and it’s algorithm has been described, since the final destination of all traversals is same, many sub-paths of the traversal routes will be common to each other, and thus the algorithm presents a scope for future streamlining, as pointed out by the reference paper too[3]. Although the guide assures that the size of an initial vector of index-lists never generally exceeds 5, the brute force algorithm, which checks $\prod_{s=2}^{n} \binom{s}{2} = e^{O(nlogn)}$ cases would soon grow to demand a lot of computational resources.
In addition to that, the code generator requires manual updating every time a new tensor is added. That could possibly be automated in some way.
Finally, even though the need has not arisen so far, it may very well happen that one needs to permute ≥ 3 pairs of indices together in a tensor. Then, a **multiPerm** function would need to be written in TARANG as well as in the raw code generator too.

6 Bibliography

[1] The project repository

[2] An introduction to np.einsum

[3] Intermediate factorization algorithm (Actually used)

[4] Cantor Hash Function

[5] Cartesian Product Reference

[6] Intermediate factorization algorithm (Reviewed-1)

[7] Intermediate factorization algorithm (Reviewed-2)

[8] Intermediate factorization algorithm (Reviewed-3)

[9] Theoretical background to the expression generation