The Algorithm and Data Structure for Cluster Expansion Calculator

Author: Yesun Huang Last update: 2020/2/12

This Algorithm is used to implement the Cluster-Expansion Calculator.

Chapter 1: Normalize Operator

Single Mode Operators

Problem:

> Given a arbitrary operator consists of total length of **n** character representing by
>
> $$a\ and\ a^\dagger$$
>
> in arbitrary order as follows:
>
> $$\{aa...a^\dagger a^\dagger...aaa....a^\dagger a^\dagger....aaa\}(length:m)$$
>
> Following the rule of
>
> $$[a,a^\dagger]=aa^\dagger-a^\dagger a=1$$
>
> Decompose the input operator into several normalized operators which all the $a^{\dagger}$ are in front of a s . For example:
>
> $$aa^\dagger a=a^\dagger aa+a$$
>
> Output all the normalized operators and their coefficients due to the decomposition process.

Solution:

> Regardless of the data structure to be described in the next chapter, we simply assume that we are able to keep all the normalized operator' s coefficients in an array which can be access in constant time.(This is possible if we are going to use the hash table)
>
> **Basic algorithm**:
>
> Consider a normalized operator multiplied by a on the left. Assuming this operator have **i** a and **j** $a^{\dagger}$. Obviously the new operator can be decomposed into one normalized operator consisting of **i+1** a and **j** $a^{\dagger}$, and **j** normalized operators consisting of **i** a and **j-1** $a^{\dagger}$. For example:
>
> $$a[a^\dagger a^\dagger aa]=a^\dagger a^\dagger aaa+2a^\dagger aa$$
>
> **Solution**:
>
> Basing on the simple algorithm described above, we simply scanning the operator from right to left. Every time we come across a, we do the composition process above and keep the result in a queue.
>
> - **Step one:**
>
>   Starting from the right, record the current position with variable `i`. we ignore all the a and when we come across the last a , we put the scanned character together as an operator into the queue. And let the a variable `j` record the last a 's position.
>
> - **Step two:**

> Then, if we come across $a^{\dagger}$, we do nothing. However, if we come across a, we out all the **current** operators in the queue. And for every operator, we do the decomposition process described above and add the new operator to the queue if it is not in the queue. (Notice that `i-j-1` tells us the additional $a^{\dagger}$ came across since last decomposition.) Besides, update the coefficient. Reset the value of `j` to `i`.

- **Final step:**

  When the scanning process is finished, use the value of `i` and `j` to out all the operators in the queue and renew the coefficient.

Obviously, the time complexity is

$$O(N^2)$$

conjecture: There is no linear time algorithm. (Find a wise computer scientist to prove it :(

Multimode Operators

Problem:

Given a arbitrary operator(m modes) consists of total length of **n** character representing by

$$a, a^\dagger, b, b^\dagger, c, c^\dagger \dots (2m\ kinds\ of\ character)$$

in arbitrary order as follows:

$$\{aa\dots a^\dagger a^\dagger \dots baab\dots . b^\dagger a^\dagger \dots . cac^\dagger\}(length : m)$$

Following the rule of

$$[O_i, O_i^\dagger] = O_i O_i^\dagger - O_i^\dagger O_i = 1$$

Decompose the input operator into several normalized operators which all the $O_i^{\dagger}$'s are in front of O_i's and all different modes' operators are in alphabetical order.

Output all the normalized operators and their coefficients due to the decomposition process.

Solution:

- **Step one:**

  Using stable sort method to gather different modes' operators separately.**(They are in order!)**

- **Step two:**

  For different modes, we do the single mode operators decomposition process and save them in the coefficient array separately.

- **Final step:**

  Connect all the operators together based on the distributive law and we get all the normalized operators and their coefficients.

Multiplied Operators

Problem:

> The high order terms which are produced by the cluster expansion process as **single operator**, and we always keep different multiplied parts of the operator in length increasing order (Or, we can say, integer increasing order).
>
> Map all the basic operator to integer and translate the operator into array(or string for less space), use '0' to represent multiplication. For example:
>
> $$< a^\dagger >< c^\dagger cba^\dagger a > -> [1, 0, 5, 6, 4, 1, 2]$$
>
> Given two normalized operators, output the normalized operator created by the multiplied process.
>
> For example:
>
> $< abc >< a^\dagger b^\dagger bc > \times < a >< bbc >< abbc >=< a >< abc >< bbc >< a^\dagger b^\dagger bc >< abbc >$
>
> or in array form:
>
> $[2, 4, 6, 0, 1, 3, 4, 6] \times [2, 0, 4, 4, 6, 0, 2, 4, 4, 6] = [2, 0, 2, 4, 6, 0, 4, 4, 6, 0, 1, 3, 4, 6, 0, 2, 4, 4, 6]$

Solution:

> The solution is quite similar to order array merging.
>
> - **Step One**
>
> Create variables `p1` and `p2` to indicate the current position of the first array and the second. Also, a boolean variable `Selection` to mark the selection and a variable `n` to records the scanning steps. Also, a new array `Result` to store the result whose length equal to sum of the length of input.
>
> - **Step Two**
>
> Scanning from left to right and compare the number from two array simultaneously, use `n` to record the steps. If the number scanned hits no '0', use the variable `Seletion` to mark the possible selection between two arrays basing on the comparison. If we come across '0' simultaneously, we use `selection` to tell us of the numbers we should put in the new array. Remember that variable `n` indicates the position we starting scanning. If we come across '0' in one array but none '0' in the other, we just simply select the former. If we reach the end of any array, just treat it like we come across '0'.
>
> - **Step Three**
>
> Reset the value of `n` to zeros, and the pointer of the unselected array to `pi-n`.
>
> - **Final Step**
>
> Repeat step two to step three until we reach the end of any array. Then we just simply put the remaining part of the other array in the new array.
>
> It is not difficult to prove that the time complexity is
>
> $$O(N)$$
>
> where N is the sum of the length of input array.

Chapter 2: Operator Tree

> It describes the data structure we used to implement the calculator. The inspiration came from *trie*s which is widely used in string searching.

Introduction

> It should be emphasized that the operators to be stored in the operator tree actually refer to the ensemble average of certain operator, that is:

$$< A >= tr(\rho A)$$

> Thus and **have different meanings in the tree. And we put a '.' in the operator input to distinguish them. For example: represented in the input is {bc.abc}.**

> **We use a order tree structure to describe a set of operators and their coefficient. And this structure support several operation which enable us to implement cluster expansion process in a efficient way.**

Operator Representation

> **All the operators to be put in the operator tree should be normalized operators.**

Tree Structure

Node

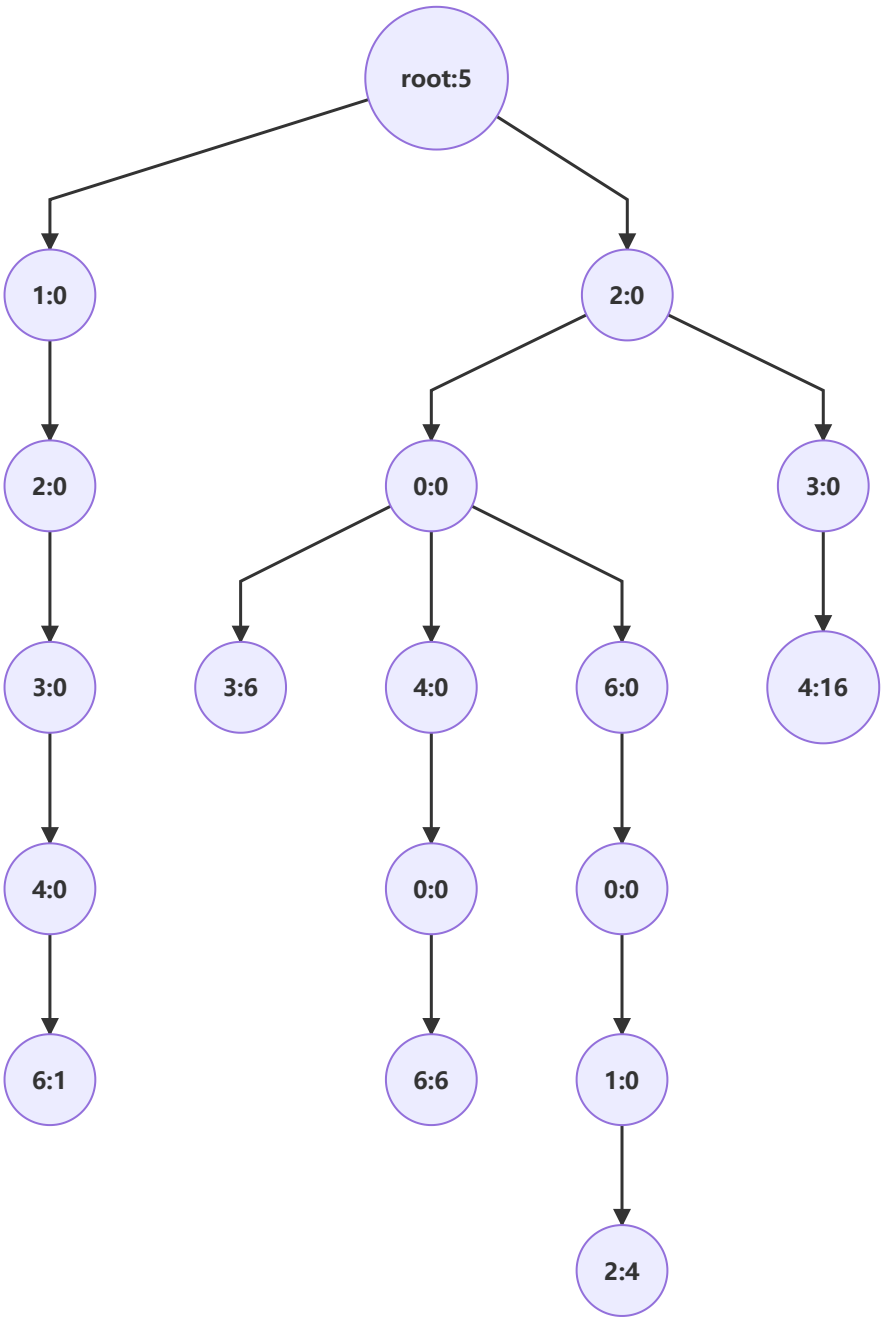
> **Each node contain links that are either null or references to its child. (Maybe there is also a link to its parents .) Obviously, you can use a pointer (or reference) array to store the link, the array length is 2m+1, m is the numbers of modes.(U can improve it and use less space based on the fact that some link is absolutely null due to the property of normalized operator.). We store the number of operators in the root.**
>
> **Besides, each note except for the root is labelled with the integer value which we use to denote basic operators. However, sometimes the labels do not refer to the basic operators but the position of certain array. In this case, we call the tree 'Position tree'.**
>
> **Specifically, just like what we did in *tries*, we store the coefficient associated with each operator in the node corresponding to its last number in the array. *nodes with zero coefficients exist to facilitate search in the operator tree and do not correspond to certain operator.***
>
> **Obviously, the max deep of the tree is n, n is the length of the longest operator.**

> - **Example:**
>
> **consider a set of operators:**

operator	array representation	coefficient

operator	array representation	coefficient
{a^dag a b^dag b c}	[1,2,3,4,6]	1
{a b^dag b }	[2,3,4]	16
{a.b^dag	[2,0,3]	6
{a.c.a^dag a}	[2,0,6,0,1,2]	4
{a.b.c}	[2,0,4,0,6]	6

The tree representation is:



Search

> Given a certain normalized operator in form of array, simply scanning the array from left to right and do DFS using the pointer stored in current node' s pointer array. The pointer obviously can be reached in constant time. If we come across a null pointer or the coefficient of the node standing for the last basic operator is zero during the searching, it means that the operator does not exist in the tree.
>
> The search process can be completed in O(n) time, where n is the length of the given operator.

Insert

> Given a certain normalized operator and its coefficient, do the search process described above. If the search hit, renew the coefficient. If not, create new node following the branches according to the array form of the operator and store the coefficient in the last node. Renew the value of the root.
>
> The insert process can be completed in O(n) time, where n is the length of the given operator.

T-T-Add

> Given two trees, add them together. Compare the total number of operators of two trees and select the larger one as output tree. Do DPS in the other tree and for every operator in the tree, do the insert process based on the coefficient.

Delete

> To delete a certain operator in the tree need a bit more work. If the last node of the operator is not the leave, we just set its coefficient to zero. If it is the leave, destroy the node and the pointer linked to it, and in the DFS recalling process, we delete all the node way back until we reach the root, the node with none zero coefficient or the node with more than one child.(In lazy version, we can also just set its coefficient to zeros but might result in space wasting). Renew the value of the root.
>
> The delete process can guarantee to be completed in O(n) time and costs O(ln n) on average, where n is the length of the certain operator.

T-O-Multiply

> In most case, we face the problem to multiply one tree with a certain operator of another tree. Use the latter tree as the output tree. Delete the certain operator in the tree but store its coefficient. Do DFS in the tree being multiplied and do the multiplied process between each operator in the tree and the certain operator described in the last chapter, multiplied their coefficient and insert the result to the output tree.

T-T-Multiply

> Multiply a tree with another will cost a great amount of time and have no better idea than brute force yet. Create a new root node, do DFS in both trees, multiply every operator in one and another, and add the result to the new tree.

Build from Position tree

Given an operator represented by an array and the root node of a constructed *position tree*, we need to rebuild an *operator tree* from the input. This time, we do BFS in the *position tree* and use a variable `deep` to store the current depth.

Notice that there might be some same basic operators (in array representation, it means the same number) in a single operator, we need to merge the note which represent the same basic operators in the same depth. While building the tree according to the given array and the position tree, we use array and bags to store the references of certain basic operator in the same depth.

Then we merge these nodes together, add up their coefficients. If they contain different children in representation of position, merge the references to children together and update their children's reference to parents. Besides, we just do the latter if their children are the same. Also, update the value of root if we both of the nodes we merged have none zero coefficients.

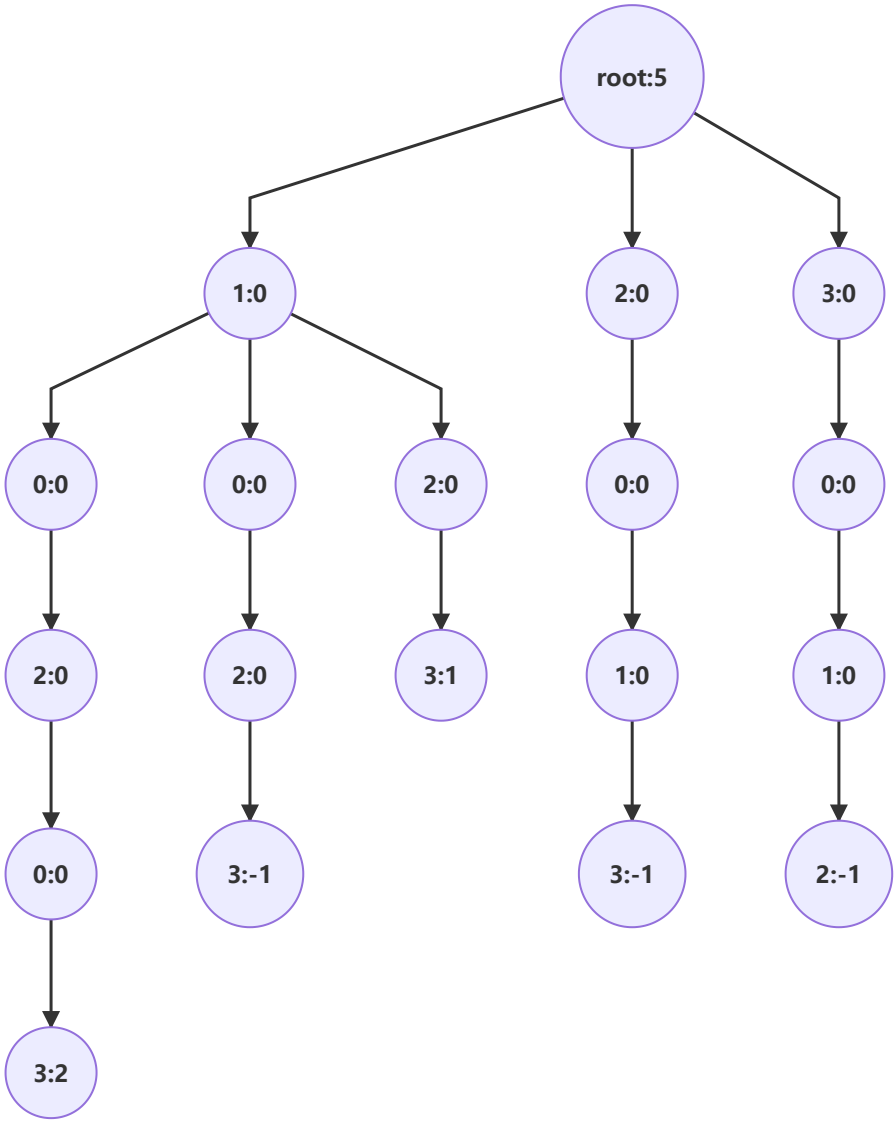
Remember to do this after the searching of current depth or it might lead to serious error.

Here is an example:

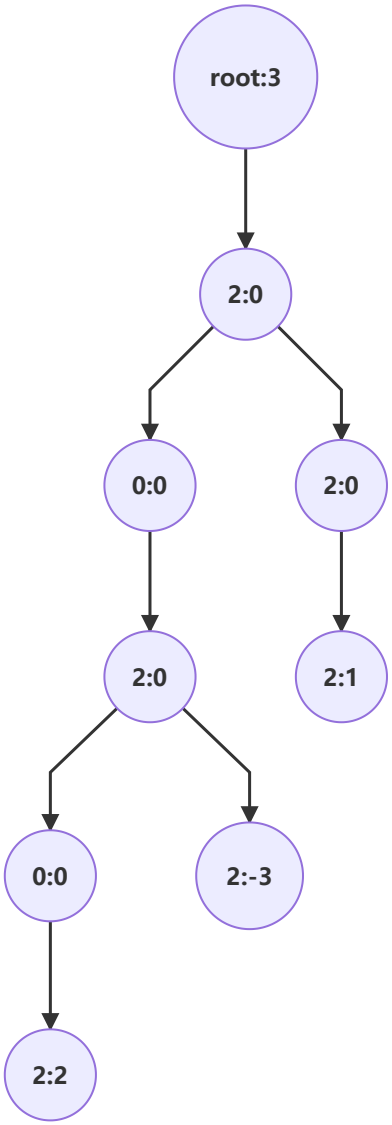
Input:

Operator: Array representation:[2,2,2]

Position Tree:



Output:



API

In pseudocode

```
public class Node
{
    public int label{get;set;}
    public int value{get;set;}
    public Node [] children{get;set;}
    public Node parent{get;set;}
    public Node(int label,int value, int M){};//M is the total number of basic
operator.
}

public class OPTree
{
    public Node root{get;set;}
    public int search(int[] operator){};//output the coefficient.
    public bool delete(int[] operator){};
    public bool insert(int[] operator){};
    public static bool ttadd(OPTree tree1,OPTree tree2){};
```

```
    public static bool tomultiply(OPTree mtree, Node moperator){};
    public static Node ttmultiply(OPTree tree1,OPTree tree2){};
    public OPTree(OPTree PositionTree){};//Build from Position Tree.
}

public static class Normalizer
{
    public static OPTree Normalize(int[] operator){};
    public static int[] Multiply(int[] operator1, int[] operator2[])
}
```

Cluster Expansion

> *Cluster Expansion* process is the key part of the calculator, but with the help of the fascinating data structure, it is not hard to implement.

Introduction

> In order to calculate the evolution of the ensemble average of certain operator basing on the *Lindblad master equation*, we need to know the ensemble average of the higher order term, which make it inevitable to solve infinite system of differential equations. However, with the help of *Cluster Expansion* approach, we are able to make some approximation and simplify the problem.
>
> Here is the simple introduction to the recursive definition of cluster expansion process:
>
> $$\begin{gathered} < N >=< N >_s \\ + < N-2 >_s \Delta < 2 > \\ + < N-3 >_s \Delta < 3 > \\ + < N-4 >_s \Delta < 4 > + < N-4 > \Delta < 2 > \Delta < 2 > \\ + \ldots \\ + \Delta < N > \end{gathered}$$
>
> where each product term represents one factorization symbolically and implicitly includes a sum over all factorizations within the class of terms identified. The purely correlated part is denoted by the last term. By omitting the last term, we are able to make good approximation and avoid solving infinite system of differential equations.

Problem

> Given an operator consist of n **basic operators**, use the cluster expansion approach to decompose it into a set of operators consist of no more than n-1 basic operators.
>
> For example:
>
> $$\begin{gathered} < aaa >= 3 < a >< aa > -2 < a >^3 \\ < abc >=< a >< bc > + < b >< ac > + < c >< ab > -2 < a >< b >< c > \end{gathered}$$
>
> Output the decomposed result in the form of operator tree.

Solution

Basic Idea

> Consider a simple condition of the problem where all the basic operators in a single operator are different. In that way, the problem reduce to a well-known model: finding all different ways to put n different balls in n same boxes. By assuming that the boxes with one ball represent single basic operator factorization, boxes with m(>1) balls represent the m-basic-operators correlations terms. It is not difficult to figure out that each distribution way of the balls represent one certain term in the cluster expansion.
>
> By applying the theory of *Bell*, the total number of the terms in n-length-operator cluster expansion is

$$B_n$$

where *B_i* is the *Bell Number*.

And the recursion formula of *Bell Number* is

$$B_{n+1} = \Sigma_{k=0}^{n} C_n^{n-k} B_k$$

Following the instruction of this formula, we can use recursion technique to calculate *cluster expansion*.

Tree representation

We will use two kind of trees in the recursion process. One is the *B-Tree*, corresponding to the right side of the cluster-expansion equation. The other is the *Delta-Tree*, which represent the expression of Delta where:

$$\Delta < N >=< N > - < N >_s - < N-2 >_s \Delta < 2 > - < N-3 >_s \Delta < 3 > -\dots$$

By assuming the operator with no '0' element in array representation refers to Delta in B-Tree but in delta tree, it is obvious that if we set all the coefficients in *B-Tree* to their opposite number except for this operator, we can generate the corresponding *Delta-Tree*. In practice, we only store the latter as a *position tree*. For convenience, we consider that

$$Delta_{tree}(1) =< 1 > (not < 1 > - < 1 >_s)$$

Combination

Notice that there is combinatorial number as factor in the recursion formula, thus we need to generate certain combination while performing the recursion.

The algorithm is simple. To choose m elements in a n-array, we just need m loop structures, each loops from the outer loop's current value to the end of the array. The current value of each loop produce one way of combination.

Recursion

With the help of the reduction, we can use the recursion formula of *Bell Number* to perform cluster expansion in an effective way. Recall that *B-Tree* is also equal to , we simplify the recursion formula of cluster expansion to:

$$B_{tree}(N+1) =< N+1 >=< a_{n+1} >< N > +\Delta < a_{n+1} \times \{1\} >< N-1 > + \Delta < a_{n+1} \times \{2\} >< N-2 > +\dots+\Delta < N+1 >$$

where a_{n+1) refer to the n+1-th element of the given array. And the latter factors of each term do not contain the n+1-th element.

Specific Implement

- Pre-calculation

  Before performing cluster-expansion for specific operator, we follow the following steps to build *Delta-Tree* corresponding to n-length-operator.

  - Step One

    Suppose that we already have stored the *Delta-Tree* for the operator whose length is less than n+1. To calculate the sum of the set of terms

    $$< a_{n+1} \times \{i\} >< N-i >,$$

> we use the combination process to produce all the possible combinations of *i* elements from the first *n* elements of the array and put the n+1-th element in front of the combinations to produce the operators:
>
> $$< a_{n+1} \times \{i\} >$$
>
> For each operator describe above, we gather the remaining elements to produce the operators:
>
> $$< N - i >$$
>
> It should be emphasized that they need to keep the original order to maintain the property of normalized operators.
>
> - Step Two
>
>   Insert the operator into the *Delta-Tree(n+1)* with coefficient -1. Then build an operator tree from the *position tree Delta-tree(i+1)* according to the operator
>
> $$< a_{n+1} \times \{i\} >.$$
>
>   Use the T-O-Multiply method to multiply the operator and the operator tree together.
>
> - Step Three
>
>   Perform all the above steps for i from 0 to n-1.
>
> - Final Step
>
>   Insert the operator <N+1> to the *Delta-Tree*(n+1) with the coefficient 1 and we finish the process of building *Delta-Tree* for n+1-length-operator.
>
> - Online Calculation
>
>   For certain input n-length-operator, we simply build an operator tree from the *Position Tree* **Delta-Tree(n)**. Output all the operators in the tree with the opposite number of the coefficient except for the operator and we generate the cluster expansion approximation of the operator.

API

```
public static OPTree ClusterExpansion(int[] operator)
```

Solver

> All the following parts should be implemented through Python.

Initial Value

Problem

> Given a initial-state (folk state) and a certain normalized operator (not in multiplied form), output its expectation value.
>
> For example:
>
> $$< 1_a, 2_b|a^\dagger ab^\dagger b|1_a, 2_b >= 2 =< a^\dagger ab^\dagger b >$$
>
> Recall the basic relation of the basic operators:
>
> $$\beta^\dagger|n_\alpha >= \delta_{\alpha\beta}\sqrt{(n_\alpha + 1)}|n_\alpha + 1 >$$
> $$\beta|n_\alpha >= \delta_{\alpha\beta}\sqrt{n_\alpha}|n_\alpha - 1 >$$
> $$< n_\alpha|n_\beta >= \delta_{\alpha\beta}\delta_{n_\alpha n_\beta}$$

Solution

> All the Input states should be represented in array. The *i*-th number in the array represent the initial-photon numbers of the *i*-th mode.
>
> For example:
>
> $$|1_a, 2_b, 0_c, 5_d >\rightarrow [1, 2, 0, 5]$$
>
> - Step One:
>
> Scanning the operator from left to right, since the operator is normalized, it is obvious that the basic operators in the same mode will be gathered together. For certain mode, record the number of the creation operator and annihilation operator as n and m.
>
> - Step Two:
>
> If n is not equal to m, the expectation value of this mode will be zero.
>
> if n is larger than the photon number (marked as n_alpha) of current mode, the expectation value of this mode will be zero.
>
> Otherwise, the expectation value of this mode will be
>
> $$n_\alpha!/(n_\alpha - n)!$$
>
> - Final Step:
>
> Multiply the expectation values of each modes together.

Evolution

> We derive the expressions for the evolution of the expectation value of certain operator based on the Lindblad master equation, that is
>
> $$\frac{d\rho}{dt} = -\frac{i}{\hbar}[H,\rho] + \Sigma_n \frac{C_{Cn}}{2}[2O_n\rho O_n^\dagger - \rho O_n^\dagger O_n - O_n^\dagger O_n \rho](1)$$
> $$\frac{d < A_i >}{dt} = tr(A_i \frac{d\rho}{dt})(2)$$
> $$H = \hbar\Sigma_i C_{Hi} O_i(3)$$
> $$Collapse\ Operator: \sqrt{C_{Cn}} O_n(4)$$
>
> where *C_i* is coefficient and *O_i* is operator.
>
> Then we can derive the evolution for the expectation value of certain operator:
>
> $$\frac{d < A_i >}{dt} = < \frac{i}{\hbar}[H, A_i] > + \Sigma_n < \frac{C_{Cn}}{2}[2O_n^\dagger A_i O_n - O_n^\dagger O_n A_i - A_i O_n^\dagger O_n] > (5)$$

Problem

> Given an operator, H and the collapse operators, output the expression for the evolution of the expectation value of the given operator in the form of operator tree.

Solution

> We perform two operations to generate the expression for evolution. One to deal with Hamiltonian operator and the other to deal with collapse operators. In order to perform *Cluster-Expansion*, we request the client to input the max-length of single Operator
>
> - Hamiltonian Operator
>
> We request the client to input the Hamiltonian Operator as a list whose elements are the operators represented in the form of array and their coefficients.
>
> > Step One:
> >
> > Following the rule of *Poisson bracket*, we can derive that:
> >
> > $$< \frac{i}{\hbar}[H, A_i] > = i\Sigma_j C_{Hj}(< O_j A_i > - < A_i O_j >)$$
> >
> > Travel through the list, follow this expression to connect the operators and normalize them, insert the result to the final expression in the form of Operator Tree.
>
> > Step Two:
> >
> > Travelling through the Operator Tree, for the single operator whose length larger than the max-length, delete the operator, do *Cluster-Expansion* approximation for the operator and insert the result to the Operator Tree.
>
> - Collapse Operators
>
> We request the client to input the list **in the form of expression (4)** as a list. The operation is similar to the one for Hamiltonian Operator, just following the following expression:
>
> $$\Sigma_n < \frac{C_{Cn}}{2}[2O_n^\dagger A_i O_n - O_n^\dagger O_n A_i - A_i O_n^\dagger O_n] >$$

Main Program

Request

> The user should input the following elements to launch the solver
>
> - The initial-state in the form of array, folk state only.
> - Hamiltonian Operator and Collapse Operators.
> - The max-length of single operator and the tracking operators.
> - Time nodes as a list for tracking evolution.

> Basically, we will use a differential equation solver provided by *scipy* to solve the differential equation we derived based on cluster-expansion approach. We will output the expectation value associate with the input time nodes of the tracking operators as an array.

Derive and Assign

> - Step One:
>
> we create a dynamic array (motivated by classic dichotomy) to store the current value and also the reference of its evolution-expression tree and its node in the tracking tree of each tracking operator. Also, an *Operator Tree* whose node' s value equal to the operator 's position in the dynamic array.
>
> - Step Two:
>
> Add the initial tracking operators provided by the user to the Operator Tree and distribute space for them in the dynamic array. Calculate the initial value for them, store them in the dynamic array. And assign their nodes in tracking tree to the dynamic array.
>
> - Step Three:
>
> Travelling through dynamic array, if the value of the address of its evolution-expression is null, derive the expression for it. Do DFS in the evolution-expression tree. If we come across any single operator which is not exist in the tracking tree, repeat step two for it.
>
> - Final Step:
>
> Repeat Step Three until all the values of the references of operators ' evolution-expressions are not null.

Calculate the Evolution

> Create a new dynamic array whose size is equal to the existed one to store the increment (This is needed in most of the differential equation solver).
>
> For each tracking operator, do DFS in its evolution-expression tree. For single operator, multiply its current value with its coefficient. For multiplied operators, multiply all the current value of single operator which it consist of and its coefficient together.
>
> Add up the sum and store it in the new dynamic array.
>
> Separate its real part and imagine part and store in the new array whose size is twice of the new dynamic array. (This is needed as most of the differential equation solver do not support complex number)

Solve the Problem

> Connect the Solver with the differential equation solver and store the evolution of each tracking operator inputted by user in the array as output.

API

```
class CEBSolver:
    "聚类分解求解器"
    def
_init_(self,InitialState,HamilitonOperator,CollapseOperator,TrackingOperator):
        return IsSuccess
    def InitialValue(self,Operators)
        return InitialValues
    def Derive(self,Operator):
        return ExpressionTree
    def Assign(self,MaxLength):
        return self.TrackingArray
    def EvolutionF(self,CurrentValue,t):
        return dydt
    def Solver(self,t_list):
        return sol
```