Algorithm using GTN

By Tianyue Li

Table of Content

Review

Last week, I put forward the GTN, the whole thing please refer to the doc below.

https://drive.google.com/file/d/1Nbt8Wq5EbYdYCAhJLnTfrWtYHIVA9gFM/view?usp=sharing

Algorithm: Weighted Enumerative Top-down search

User Specification

User input should includes:

1) A sequence of input tensors
2) An output tensor
3) The GTN relationship between each input tensor and the output tensor.

Algorithm 1: Weighted Enumerative Top-down search

- Algorithm

The algorithm’s picture is attached next slide.

Algorithm 1: Weighted Enumerative Top-down Search Algorithm

Input: User define spec R, Ins, out
Output: p

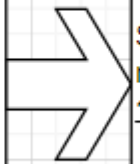
ϕ = LeastGraphRequirement(R)
$\tilde{P} = \{\}$ // data type: dict
$\tilde{Q}$ = GenerateInitialization(ϕ) // generate all possible skeletons
$\tilde{Q}'$ = Evaluate($\tilde{Q}$) // evaluate their weights
$\tilde{P}$ = Assign($\tilde{P}$,$\tilde{Q}$) // Put $\tilde{Q}$ into correct place in $\tilde{P}$
weight = 1 // search start at weight 1
$\tilde{C} = \{\}$ // cache dict
while *True* **do**
  while $\tilde{P}$*[weight]*$\neq \emptyset$ **do**
    p = RemoveFirst($\tilde{P}$[weight]) // get the first skeleton in $\tilde{P}$[weight]
    $\tilde{F}$ = Fillin(P) // The set of full operation based on skeleton p
    for f *in* $\tilde{F}$ **do**
      if $f(ins) = out$ **then**
        return f
    if *compressing(p) not* $\in \tilde{C}$ **then**
      $\tilde{E}$ = Expand(p,ϕ) // Expansion process, find more skeleton
      $\tilde{C}[compressing(p)] = \tilde{E}$
    else
      $\tilde{E} = \tilde{C}[compressing(p)]$
    $\tilde{E}'$ = Evaluate($\tilde{E}$)
    $\tilde{P}$ = Assign($\tilde{P}$,$\tilde{E}'$)
  weight += 1;

Algorithm 1: Weighted Enumerative Top-down search

-Example from TF-coder paper

search space:
tf.add() (2T),
tf.gather() (1G 1N),
tf.argsort() (1N),
tf.transpose() (1G)

Input tensors:
in1: [10, 20, 30, 40, 50, 13, 17, 19, 21, 22, 23]
in2: [1, 1, 1, 1, 1, 0, 0, 0, 2, 2, 2]
Output tensor: [13, 17, 19, 10, 20, 30, 40, 50, 21, 22, 23]
Description: Reorder segments
Specification: in1->G->out, in2->N->out.



Small graph requirement:
1 G 1 N at least.

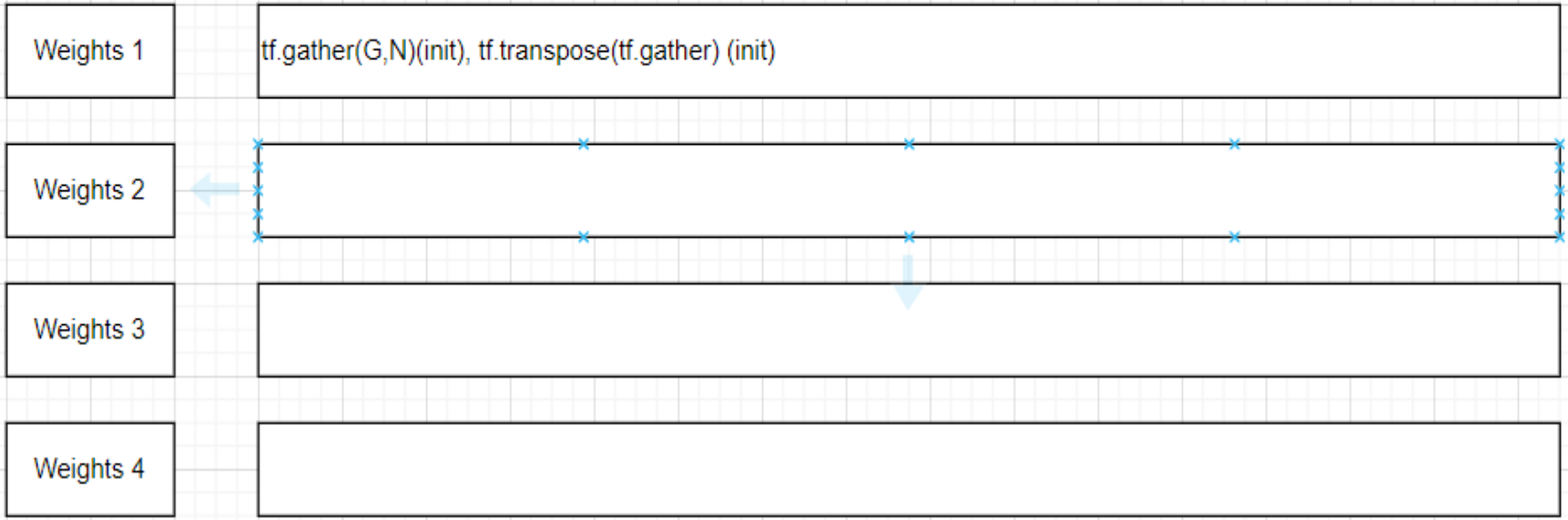
1 init the search set

1 try tf.gather(G,N), no result, expand it

search space:
tf.add() (2T),
tf.gather() (1G 1N),
tf.argsort() (1N),
tf.transpose() (1G)

Input tensors:
in1: [10, 20, 30, 40, 50, 13, 17, 19, 21, 22, 23]
in2: [1, 1, 1, 1, 1, 0, 0, 0, 2, 2, 2]
Output tensor: [13, 17, 19, 10, 20, 30, 40, 50, 21, 22, 23]
Description: Reorder segments
Specification: in1->G->out, in2->N->out.

Small graph requirement:
1 G 1 N at least.



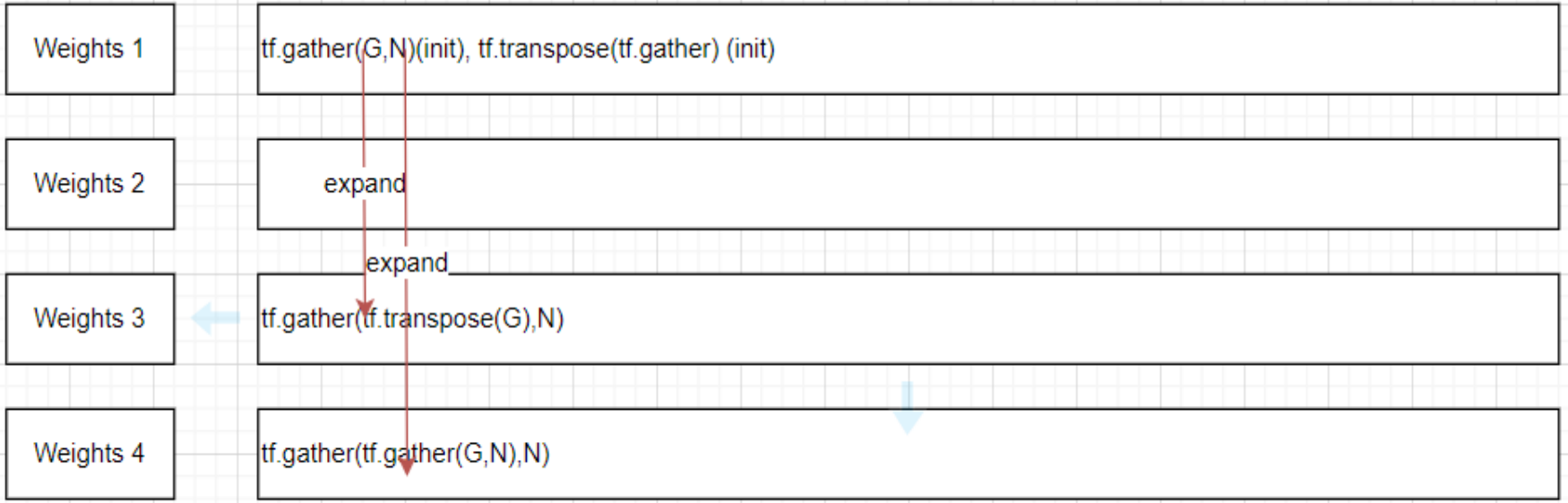
2 search on tf.gather(G,N)

Start from G

search space:
tf.add() (2T),
tf.gather() (1G 1N),
tf.argsort() (1N),
tf.transpose() (1G)

Input tensors:
in1: [10, 20, 30, 40, 50, 13, 17, 19, 21, 22, 23]
in2: [1, 1, 1, 1, 1, 0, 0, 0, 2, 2, 2]
Output tensor: [13, 17, 19, 10, 20, 30, 40, 50, 21, 22, 23]
Description: Reorder segments
Specification: in1->G->out, in2->N->out.

Small graph requirement:
1 G 1 N at least.



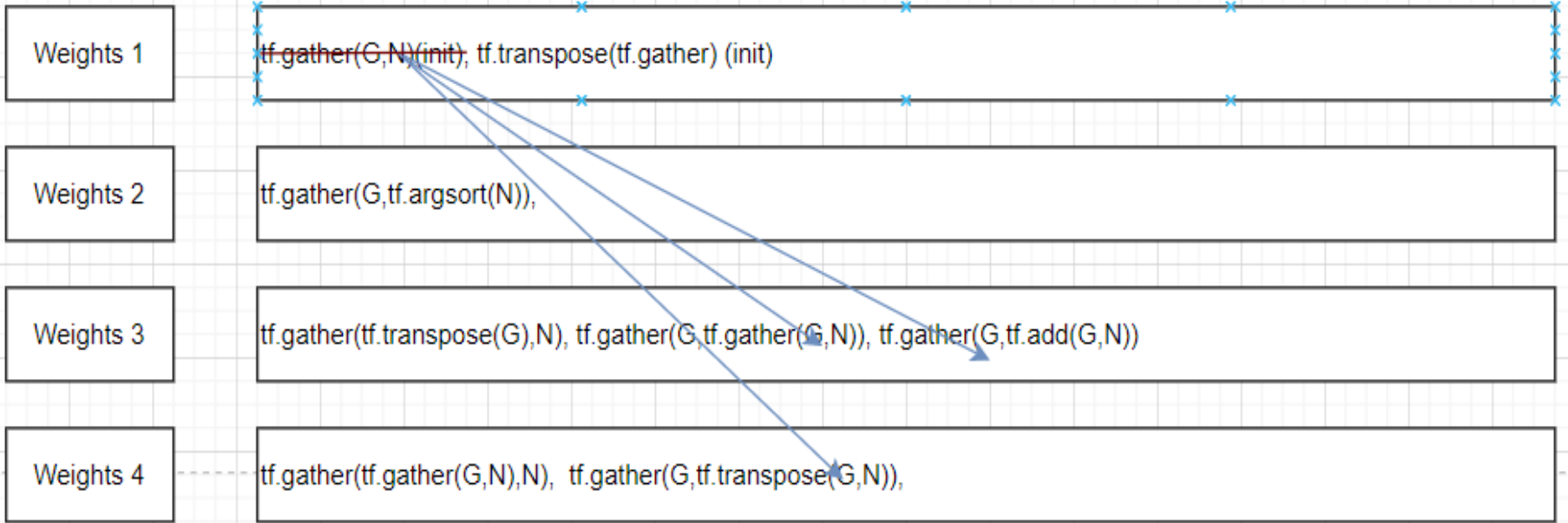
3 search on tf.gather(G,N)

Start from N, and remove tf.gather(G,N) after that.

search space:
tf.add() (2T),
tf.gather() (1G 1N),
tf.argsort() (1N),
tf.transpose() (1G)

Input tensors:
in1: [10, 20, 30, 40, 50, 13, 17, 19, 21, 22, 23]
in2: [1, 1, 1, 1, 1, 0, 0, 0, 2, 2, 2]
Output tensor: [13, 17, 19, 10, 20, 30, 40, 50, 21, 22, 23]
Description: Reorder segments
Specification: in1->G->out, in2->N->out.

Small graph requirement:
1 G 1 N at least.



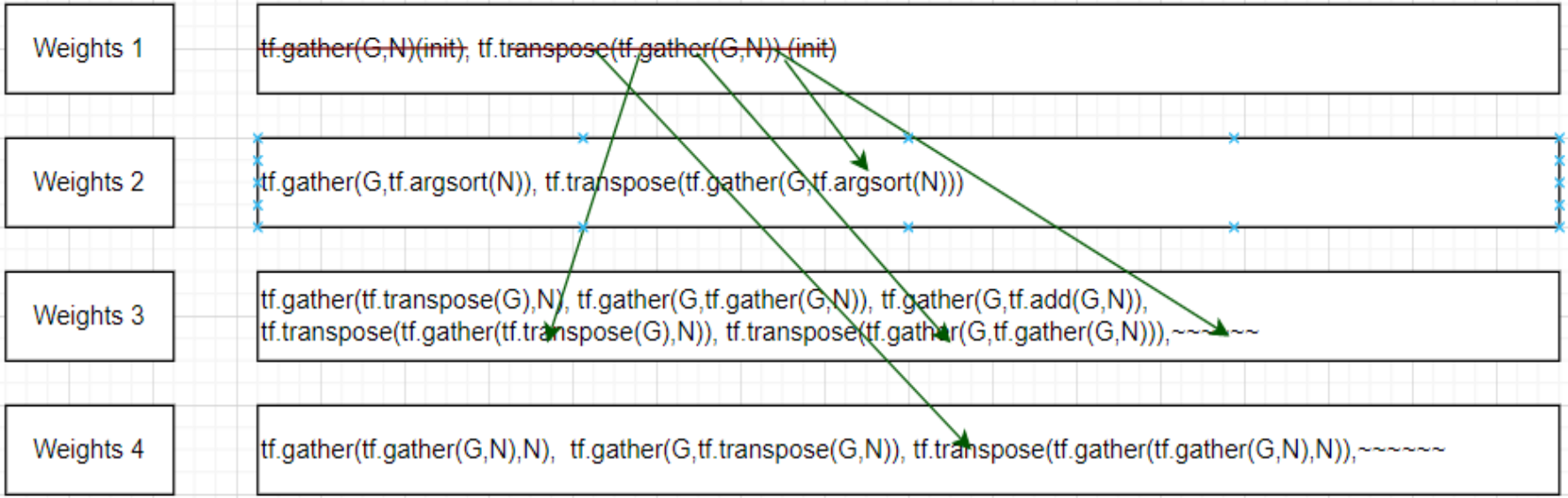
4 search on tf.transpose(tf.gather(G,N))

Inherent from tf.gather(), remove tf.transpose(tf.gather(G,N)) after that

search space:
tf.add() (2T),
tf.gather() (1G 1N),
tf.argsort() (1N),
tf.transpose() (1G)

Input tensors:
in1: [10, 20, 30, 40, 50, 13, 17, 19, 21, 22, 23]
in2: [1, 1, 1, 1, 1, 0, 0, 0, 2, 2, 2]
Output tensor: [13, 17, 19, 10, 20, 30, 40, 50, 21, 22, 23]
Description: Reorder segments
Specification: in1->G->out, in2->N->out.

Small graph requirement:
1 G 1 N at least.



5 test on tf.gather(G,tf.argsort(N))

FIND SOLUTION 😀

search space:
tf.add() (2T),
tf.gather() (1G 1N),
tf.argsort() (1N),
tf.transpose() (1G)

Input tensors:
in1: [10, 20, 30, 40, 50, 13, 17, 19, 21, 22, 23]
in2: [1, 1, 1, 1, 1, 0, 0, 0, 2, 2, 2]
Output tensor: [13, 17, 19, 10, 20, 30, 40, 50, 21, 22, 23]
Description: Reorder segments
Specification: in1->G->out, in2->N->out.

Small graph requirement:
1 G 1 N at least.

Weights 1	~~tf.gather(G,N)(init)~~, tf.~~transpose(tf.gather(G,N)) (init)~~
Weights 2	tf.gather(G,tf.argsort(N)), tf.transpose(tf.gather(G,tf.argsort(N)))
Weights 3	tf.gather(tf.transpose(G),N), tf.gather(G,tf.gather(G,N)), tf.gather(G,tf.add(G,N)), tf.transpose(tf.gather(tf.transpose(G),N)), tf.transpose(tf.gather(G,tf.gather(G,N))),~~~~~~
Weights 4	tf.gather(tf.gather(G,N),N), tf.gather(G,tf.transpose(G,N)), tf.transpose(tf.gather(tf.gather(G,N),N)),~~~~~~

More explanation on Algorithm 1

Initialization of Q

-method1 (used)

Method:

For the initialized un-filled query *q* should satisfy the “least graph requirement”, like “1G 1N” before. And all the possible smallest combinations should be considered,

Pros:

Good start of the program. Make things easier to handle during the search.

Cons:

Initialized set will be a large set when the search space is large. It can cause lot of time to generate them and evaluate them at the same time.

Initialization of Q

-method2 (improved)

Method:

Start only with potential single operation. Prioritize those operations that directly satisfy the user requirements. When the search reaches these operations not satisfying the “least graph requirements”, we can expand them to satisfy the requirements.

Pros:

Make the initialization set small. And can inherit from the previous search experience.

Cons:

Harder to handle.

inheritance mechanism

Concept of Abstract Terminal: Abstract Terminal refers to the compressed sequence of sub-operation relationship. And the compressing rule follows the $\mathcal{GTN}$ abstract semantics.

Examples: $\mathcal{G}->\mathcal{G}$ can be compressed into single $\mathcal{G}$. $\mathcal{G}->\mathcal{N}->\mathcal{T}$ can be compressed into single $\mathcal{N}$.

Theorem of Inheritance During the process of one search, a graph waited to be expand like $F(g(\square,\square))$, in which F is a series of combined operations, and g is one of the most inner single operations waiting to be expand. The g's expansion can inherit from other **similar** operation *if and only if* g has the same **abstract terminal** and the same starting point.

3 More improvement on Abstract Semantics

Improvement on class N –reason

Note that the property of N makes the class G and class T lose their influence during the expanding process. Therefore, we need to find a way to either make these two classes keep their influence or make N less influential to them.

Improvement method 1: guidance in the middle

Note that the set N is a large set and it can sometimes cause the pruning less efficient when there are lots of N involved. We can add some guidance in the middle to utilize the G sub-operation relation.

$$P(\mathrm{r} \in \mathcal{G}) = \begin{cases} 1(user) \\ \frac{count(e \in out \cap e \in in)}{count(e \in out)} \end{cases}$$

Use Above equation to calculate the **distance** between the output tensor and the false prediction.

Improvement on class N -method 2: divide the class N

Actually, another method which I prefer is to divide the class N into more sub-classes which are at the same level of importance. For example, we can get class I, which represents the indexing operation, and the class S, which represents the shaping operation. Then, these two classes should have the same level of importance, making the abstract semantics below:

I->S = S (since S is the last)

G->I = I (same as N)

I->G = I …