1 Motivation

The current calculation system is long overdue for an overhaul. It works well for baked-in sets of formulas but has shortcomings when specification involves more complicated procedures. For example, Ganyu's A1, which increases the Frost Flake CRIT Rate by 20%, requires the Frost Flake calculation to use a new formula with a different CRIT Rate calculation instead of the preexisting one. Using different formulas for similar calculations, such as in this scenario, also prevents the optimizer from merging those calculations.

The goal of the new formula engine includes

- High flexibility when editing the formula, including when the operations may depend on one another,
- Ability to indicate that a portion of the code is reused at multiple places to avoid duplicate computations,
- Fast repeated calculations (comparable to the old one) where each round has slightly different parameters, and
- Ability to add multiple *final formulas* together to form multi-variate optimization.

2 Design

The design of *Waverider* takes a lot of inspirations from computational graph in TensorFlow[1]. It uses Directed Acyclic Graphs (DAG) to represent formulas. In particular, each node in the graph represents a single formula. Some formulas rely on the values of another formulas, which we refer to as *operands*. This dependency is indicated using graph edges; edges leaving a node point toward its operands.

In this document, we omit the arrow heads for graph edges, and instead use the convention that the edges point downward, from higher nodes toward lower nodes. Nodes with no outward edges are *leaf nodes*, representing constants and variables. Nodes with no inward edges are *root nodes*. They represent the final formulas that the entire graph represents. Note that some graphs may have multiple root nodes.

Figure 1 shows an example of a computation graph, representing a formula $3 + 8 * 7 * x$. The leaf nodes are nodes $3, 8, 7$, and x, all representing either constants or variables. The internal nodes are nodes $+$ and $*$. The $*$ node represents the product of its operands, i.e., the formula $8 * 7 * x$, while the $+$ node represents the sum $3 + (8 * 7 * x)$, which is also the formula that the entire graph represents.

We now discuss available types of supported nodes in more details.

3 Node Types

Throughout this section, we use y for the value of the current node and $x_1, x_2, \ldots, x_n$ for the values of its operands.

[1]`https://www.tensorflow.org/guide/intro_to_graphs`

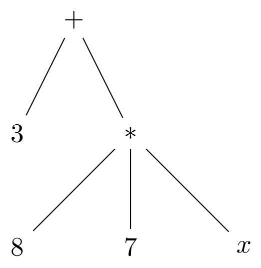

Figure 1: Graph for $3 + 8 * 7 * x$

3.1 Constant Node

Constant nodes are nodes that represent constant values of any kind. They have no children, and therefore are leaf nodes. The current design supports two variants, numerical constants and string constants. Nodes $3, 8$, and 7 in Figure 1 are examples of constant nodes.

3.2 Compute Node

Most of internal nodes are *compute nodes*. They are nodes that compute their values using only the values of their operands. Supported formulas includes

- Enemy Resistance (res):

$$y = \begin{cases} 1 - x_1/2, & x_1 \in (-\infty, 0), \\ 1 - x_1, & x_1 \in [0, 3/4), \\ 1/(1 + 4x_1), & x_1 \in [3/4, \infty), \end{cases}$$

- Threshold (thres):

$$y = \begin{cases} x_3, & x_1 \geq x_2, \\ 0, & \text{otherwise}, \end{cases}$$

- Summation (+): $y = \sum_i x_i$,
- Production ($*$): $y = \prod_i x_i$,
- Maximum (max): $y = \max_i \{x_i\}$,
- Minimum (min): $y = \min_i \{x_i\}$, and
- Sum-Fractional (frac): $y = x_1/(x_1 + x_2)$.

Note that some formulas have a fixed number of operands, such as Enemy Resistance, while others support a variable number of operands, such as Summation.

3.2.1 Subscript Node

Subscript nodes are special nodes that is accompanied by a dictionary. These nodes lookup the dictionary using the *index*, which is the first operand x_1. Indices can be either integers or strings. A subscript node with an accompanied dictionary X is denoted by lookup_X.

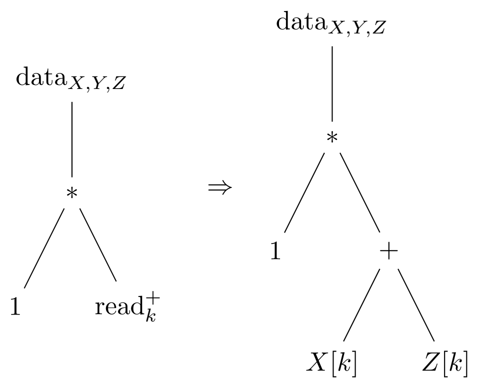

Figure 2: Process of *reading* read_k using $\text{data}_{X,Y,Z}$ assuming $X[k]$ and $Z[k]$ exist

3.3 Read Nodes and Data Nodes

For graphs to *take input* and for general modification, we use *read nodes* and *data nodes.* Read nodes act as placeholders for portions of the graph that are not yet available during graph construction. Data nodes then provide appropriate replacement for each read node during an operation called *read operation.*

Each read node contains a *key* used to identify nodes that it represents and an *aggregation method* for combining multiple nodes that match its key into a single node. The supported aggregation method includes summation, production, minimum, and maximum. Read nodes have no operands, and therefore are leaf nodes. We denote each read node with key k and aggregation method a by read_k^a.

Data nodes specify the mapping that read nodes use to map their keys to appropriate nodes. Each data node contains one or more *data objects*, each of which is a dictionary mapping each key into a single node. The data node has one operand, which is simply forwarded when evaluating, i.e., $y = x_1$. We denote the data node with data objects $X_1, X_2, \ldots, X_n$ as $\text{data}_{X_1,X_2,\ldots,X_n}$.

Read operations update the read nodes by replacing it with the (aggregated) nodes given by data nodes. During a read operation, each read node finds the nearest data node among its ancestors. Then, for each data object in the data node, it maps its key to a node. The mapped nodes are then aggregated using the aggregation method of the read node. The read node is then replaced with the resulting aggregated node. If the data object does not contain a key k for a read node read_k, the data object is ignored. If no data object contains key k, the read node remains unchanged.

Figure 2 illustrates the reading operation at a single read node. In this setup, the operand of a data node $\text{data}_{X,Y,Z}$ has one read node read_k^+. Furthermore, we assume that $X[k]$ and $Z[k]$ exist, but $Y[k]$ does not. During the read operation, the read node looks up $X[k], Y[k]$, and $Z[k]$ and combines the matching nodes using its aggregation method, $+$. Since $Y[k]$ does not exist, it is not included in the aggregated node.

The reason for this design is two-folded. On the one hand, we can use a read node as an *input* and replace it with a constant node when computing the final value(s). However, on the other hand, we can also use read nodes to describe modifications as the formula for read_k can be changed by changing the list of data object within the corresponding data node. We will discuss the process of formula modification in more details in Section **??**.

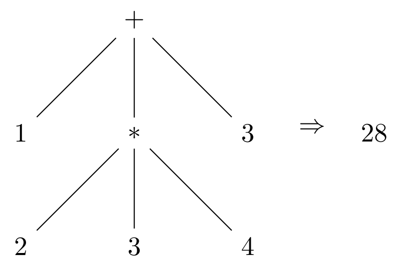

Figure 3: Constant folding of $1 + 2 * 3 * 4 + 3$ into 28

4 Optimization

There are multiple optimizations one can do to a graph to reduce computation time. The metric used when optimizing the graph is the number of nodes $|V|$, as well as the number of edges $|E|$ in the graph. A proper optimization would reduce either $|V|$ or $|E|$ while not increasing the other.

This section list a few candidates for optimizations. Note that the graph is designed to be immutable (to avoid *spooky action at a distance*[2]), so most of the *modifications* actually create a new graph that represents the modified formulas.

4.1 Constant Folding

The values of some nodes don't change across computations. We can directly replace it with fewer computations, such as

- If all operands of a node are Constant Nodes, we can replace it with the result of the calculation (as another Constant Nodes),
- We can apply read operation to each read node, and
- We can replace each data node with its only operand (x_1), after reading all read nodes.

Figure 3 shows an example of constant folding. Since the formula $1 + 2 * 3 * 4 + 3$ uses only constants in its computation, it can be replace with the result 22.

4.2 Flattening

Some operations, such as summation and production, are commutative monoid[3]. We can merge nodes with the same commutative monoid operation, e.g., as in Figure 4. However, if an operand x_i is used by multiple nodes, we don't merge those operands; it is more efficient to keep such nodes separated.

4.3 Deduplication

Some computations appear multiple times at different places in the hierarchy. We can replace the duplicated nodes with either of the duplicates so they share the result. These nodes include

[2]`https://en.wikipedia.org/wiki/Action_at_a_distance_(computer_programming)`
[3]`https://en.wikipedia.org/wiki/Monoid#Commutative_monoid`

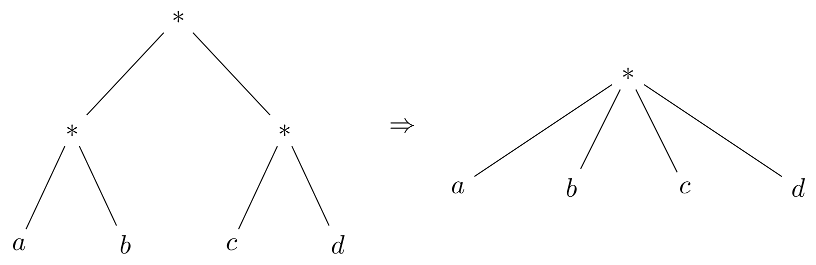

Figure 4: Flattening of $(a * b) * (c * d)$ into an equivalent $a * b * c * d$

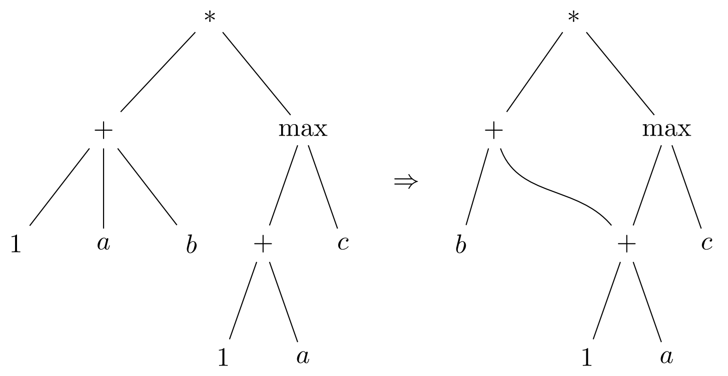

Figure 5: Deduplication of $(1 + a)$ in $(1 + a + b) * \max(1 + a, c)$

- Read Nodes with the same key,
- Any nodes with the same dependencies in the same order, and
- Any commutative monoid nodes with the same dependencies in any order.

Furthermore, if any two commutative monoid nodes have more than one common dependency, we can separate common nodes into a nested operand.

Figure 5 shows an example of deduplication on $(1 + a + b) * \max(1 + a, c)$ In this figure, the the node $1 + a + b$ and $1 + a$ share the same part of the computation $1 + a$. The $1 + a$ formula is then used to replace a portion of the formula $1 + a + b$.

4.4 Rules for Applying Optimization

Not all optimizations are beneficial everywhere. This section list the rules where the optimizations described in this section can be applied to reduce the metric ($|V|$ and $|E|$).

Describe the optimization application rules here.