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 different types of available nodes in more details.

3 Node Types

We distinguish nodes into numerical nodes and string nodes, based on the type of values they return. That is, numerical nodes represent formulas returning some numerical values, while formulas in string nodes return string values. Numerical nodes are primary nodes in the computational graph as this structure is designed to perform numerical computation.

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

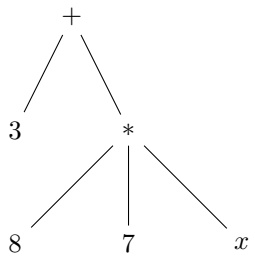

Figure 1: Graph for $3 + 8 * 7 * x$

The string nodes are used primarily to maintain the states of the graph, such as the element type of the dmg, the type of the weapon that the character uses, etc.

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.

3.1 Constant Node

Constant nodes are nodes that represent constant values of any kind, both numerical and string. They have no children, and therefore are leaf nodes. Nodes $3, 8$, and 7 in Figure 1 are examples of constant nodes.

For numerical constant nodes, we use the constant values to represent those nodes. For string nodes, we also use double quotes to distinguish them from variable names, e.g., "sword", "electro", and "avg".

3.2 Compute Node

Most of internal nodes are *compute nodes*. They are numerical nodes that compute their values using only the values of their numerical 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 compute nodes. Each subscript node is accompanied by a dictionary. It uses the only numerical operand x_1 as an index to lookup the dictionary. A subscript node with a dictionary X is denoted by lookup_X.

3.3 Read Nodes and Data Nodes

For general graph 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*. For more information on read operation, see Section 4.1.

Each read node contains a string array *key* used to identify nodes that it represents and an *aggregation method* for combining all nodes that match the key into a single node. The supported aggregation methods 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^a_k.

For some read nodes, the keys are combinations of an array string *prefix* followed by a string computed using a string node called *suffix*. We denote such read nodes with prefix p, suffix node s, and an aggregated method a by read^a_{p+s}. Note that the string node s in this case is not treated as an operand, and the read node read^a_{p+s} remains a leaf node of the graph.

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 $\text{data}_{X_1,X_2,\ldots,X_n}$.

3.4 Input Nodes

Input nodes identify the formula where the actual value of the node is fetched from outside the graph. Each input node is associated with a single string *key*. We denote an input node with key k by input_k.

Note that the main difference between read nodes and input nodes is the read node fetch values from the associated data node, which is within the graph, while input nodes fetch data from other sources entirely outside of the graph. Another difference is that read nodes may compute portion of the keys using string nodes, while the keys of input nodes are known at construction time.

3.5 Priority Nodes

Priority nodes are string nodes that uses the value of the first (string) operands that returns a non-empty string. That is,

- If $x_1 \neq \emptyset, y = x_1$,

- If $x_1 = \emptyset, x_2 \neq \emptyset, y = x_2$,
- If $x_1 = x_2 = \emptyset, x_3 \neq \emptyset, y = x_3$,
- etc.

4 Operations on Graph

There are multiple optimizations one can do to a graph to reduce computation time and precompute portions of the graph. This section list a few operations available on a graph. 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 Read Operation

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 is replaced with the vacuous sum of its operation. In particular, the replacement value for each operation is shown in the following table.

operation	$+$	$*$	min	max
replacement	0	1	∞	$-\infty$

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.

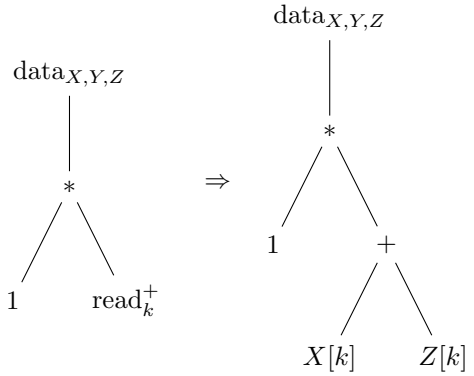

Figure 2: Process of *reading* read_k using $\text{data}_{X,Y,Z}$ assuming $X[k]$ and $Z[k]$ exist

[2] https://en.wikipedia.org/wiki/Action_at_a_distance_(computer_programming)

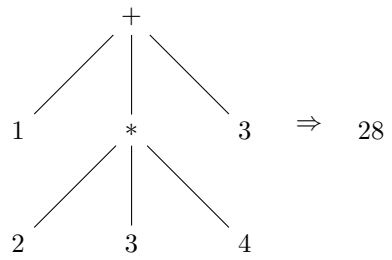

Figure 3: Constant folding of $1 + 2 * 3 * 4 + 3$ into 28

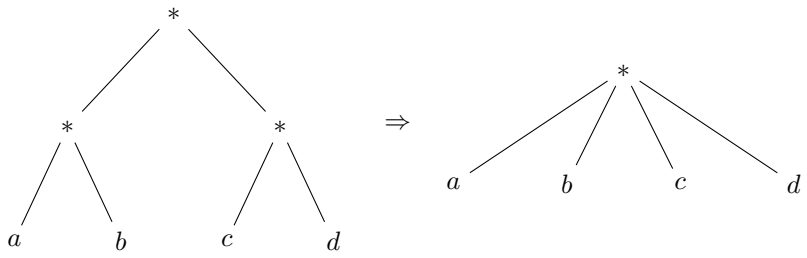

Figure 4: Flattening of $(a * b) * (c * d)$ into an equivalent $a * b * c * d$

4.2 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.

4.3 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.

4.4 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

[3] `https://en.wikipedia.org/wiki/Monoid#Commutative_monoid`

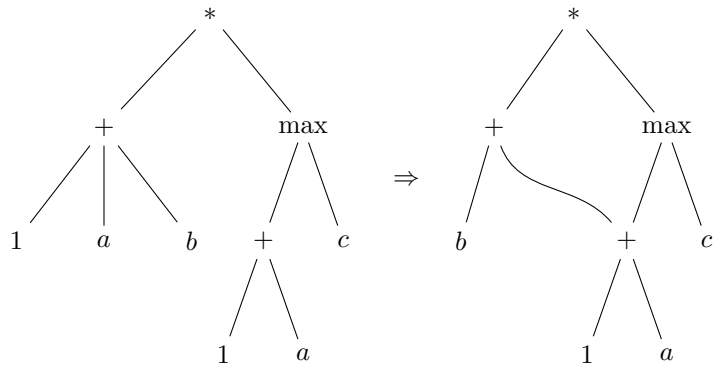

Figure 5: Deduplication of $(1 + a)$ in $(1 + a + b) * \max(1 + a, c)$

- 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 $1 + a + b$.