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 computation graphs 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. Furthermore, operands are ordered from left to right; the left-most operand is the first operand of the node, and the right-most operand is its last operand. 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 or $\emptyset$. Numerical nodes are primary nodes

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

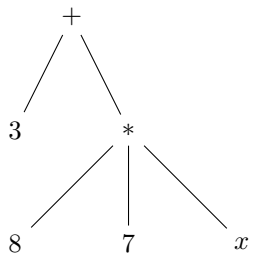

Figure 1: Graph for $3 + 8 * 7 * x$

in the computational graph as this structure is designed to perform numerical computation. 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 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):

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

- 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.

For performance reason, subscript nodes are further subdivided into basic *subscript nodes* and *lookup nodes*. The distinction is that each entry in a basic subscript node points to a constant value and only support numerical index. In contrast, entries in a lookup node may contain other nodes, and the index can be either numerical or string, allowing for more flexibility at a cost of larger overhead.

3.3 Priority Node

Priority nodes are string nodes that uses the value of the first (string) operands that returns anything other than a $\emptyset$. 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.

3.4 Conditional Node

Conditional nodes have four operands, and formulas are of the form

$$y = \begin{cases} x_3, & P(x_1, x_2), \\ x_4, & \text{otherwise}, \end{cases}$$

where P is a predicate specific to each type of conditional nodes. The types of x_3 and x_4 must agree, i.e., both are numerical nodes, or both are string nodes.

Supported formulas include

- Threshold node (thres): $P(x_1, x_2) = (x_1 \geq x_2)$,
- Matching node (match): $P(x_1, x_2) = (x_1 = x_2)$.

Due to the definition of the predicates, x_1 and x_2 of a threshold node must be numerical nodes, while x_1 and x_2 of a matching node only need to have the same trype.

3.5 Read Node and Data Node

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 node application, see Section 4.

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_k^a. If a read node has no aggregation method, its key must map to a unique node.

Data nodes specify the mapping that read nodes use to map their keys to appropriate nodes. Each data node contains one *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 a data node with a data objects X as data_X.

Additionally, a data node may be marked as a *data reset node*, indicating that its operand does not use the data from the ancestors of the reset node. We denote the data reset node with a data object X as data_X^*

4 Read Node Application

Read node applications update the read nodes by replacing them with the (aggregated) nodes given by data nodes. The operation is as followed, for each read node with key k and aggregation method a,

1. Finds all data nodes $\{\text{data}_i\}_i$ among its ancestors,
2. Map the data object D in each data node data_i using the key k to a node $D[k]$, ignoring any data objects that do not contain the key k,
3. Check if there are at least one mapped nodes,
   (a) If there are mapped nodes, aggregate the mapped nodes using the aggregation method a, and replace the read node with the aggregated method,
   (b) Otherwise, the read node remains unchanged, and the operation proceeds to the next read node,
4. If there are unconsidered read nodes, go back to Step 1.

After all read nodes are considered, replace all data nodes with their operands. In Step 3a, if the replaced node contains read nodes, they are also included in later iterations of Step 1. In Step 1, if the ancestor is a data reset node, all ancestors of the data reset node is ignored. Unapplied read nodes in Step 3b are considered to be an *input* to the graph. That is, during the computation of the graph, these input nodes are replaced with constant nodes with appropriate values.

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.

5 Graph Optimization

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 representing the modified formulas.

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

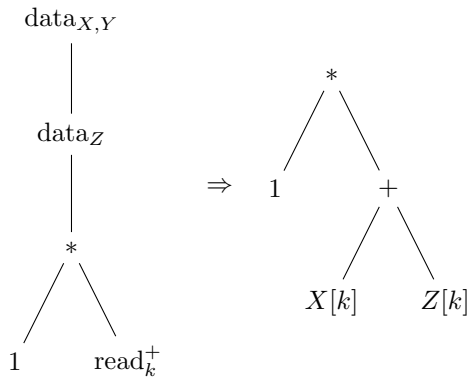

Figure 2: Process of *reading* read_k using $\text{data}_{X,Y,Z}$ assuming $X[k]$ and $Z[k]$ exist and $Y[k]$ does not

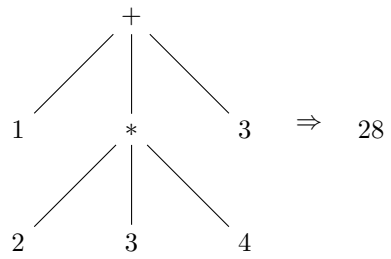

Figure 3: Constant folding of $1 + 2 * 3 * 4 + 3$ into 28

5.1 Constant Folding

The values of some nodes don't change across computations. We can directly replace it with fewer computations. If all operands of a node are Constant Nodes, we can replace it with the result of the calculation (as another Constant 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 replaced with the result 22.

5.2 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.

5.3 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

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

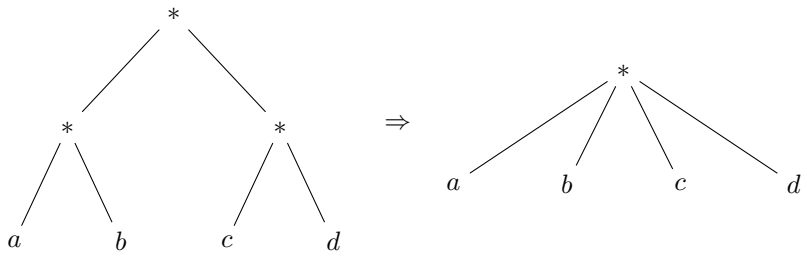

Figure 4: Flattening of $(a * b) * (c * d)$ into an equivalent $a * b * c * d$

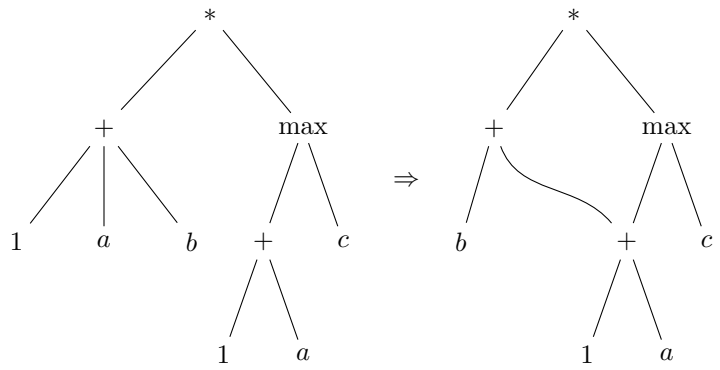

Figure 5: Deduplication of $(1 + a)$ in $(1 + a + b) * \max(1 + a, c)$

- 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 $1 + a + b$.

5.4 Pruning

5.5 Reaffining

As mentioned in Section 4, unapplied read nodes are treated as an input to the graph, and are replaced with appropriate values during graph computation. These read nodes normally represent an affine computation of some external factors, such as the total "DEF" contribution from all artifacts ($\text{DEF}_{\text{artifact}}$), which is affine w.r.t. individual "DEF" contributation of an artifact in each equipment slot ($\text{DEF}_{\text{artifact}[slot]}$). In particular, $\text{DEF}_{\text{artifact}} = \sum_{slot} \text{DEF}_{\text{artifact}[slot]}$.

As such, the graph computation generally includes an *affine combinator*, which is a functionality that combines the values from different factors ($\text{DEF}_{\text{artifact}[slot]}$) into a single

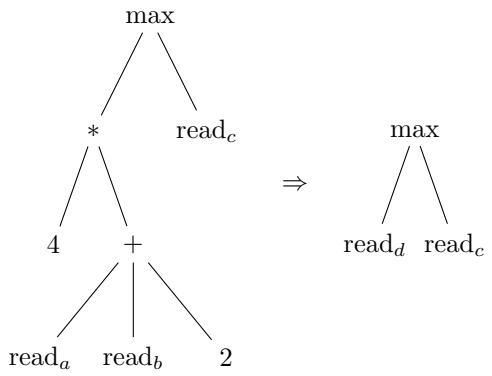

Figure 6: Reaffining of $\max\{4 * (a + b + 2), c\}$ into $\max\{d, c\}$

value ($\text{DEF}_{\text{artifact}}$) before replacing the corresponding unapplied read node. Therefore, it is useful to offload the affine computation to the affine combinator instead of using generic compute nodes within the graph.

The *reaffining* helps offloading the affine computation in the graph by replacing read nodes that are part of an affine function with a single read node representing that entire computation.

Figure 6 shows an example of reaffining $\max\{4 * (a + b + 2), c\}$ where

- $a = C_a + \sum_{i=1}^{n_a} a'_i$,
- $b = C_b + \sum_{i=1}^{n_b} b'_i$, and
- $c = C_c + \sum_{i=1}^{n_c} c'_i$

are affine w.r.t. the inputs a'_i, b'_i, and c'_i. In this figure, the node $4 * (a + b * 7)$ is affine w.r.t. its non-constant operands a and b, so reaffining replaces it with a new read node $read_d$.

Prior to reaffining, the affine combinator needs to compute a and b before replacing the nodes read_a and read_b. Even then, the graph still needs to compute the addition $a+b+2$ as well as the multiplication $4*(a+b+2)$. After reaffining, the graph can instead compute $d'_i = 4 * (a'_i + b'_i)$ and $C_d = 4 * (C_a + C_b + 2)$ a priori and only compute $d = C_d + \sum_{i=1}^{n_d} d'_i$ for the left half of the equation.

Not only does reaffining reduce the number of nodes in the graph, it also helps with the pruning process by combining the computation for an accurate estimation of the range of the value of each node.

6 Team Buff