Design Of Mergesort Experiments

Table of Contents

1 Introduction

This document describes the design of mergesort experiments.

2 Experiments

3 Shuffle

3.1 Interface

The user interface for the shuffle experiment consists of the following:

arrays
: There are three arrays present in the interface. One array for the final result of the shuffling operation and two called `Left` and `Right` for the inputs to the shuffle operation.

buttons
: There are two buttons.

  LEFT Button
  : The LEFT button is used to pick the first element from the `Left` array and place it at the end of the result array.

  RIGHT Button
  : The Right button is used to pick the first element from the `Right` array and place it at the end of the result array.

3.1.1 Sorting status of the arrays

Each array in the system is either sorted or unsorted. This sorting status of the array is indicated by the background color of the array.

Unsorted
: If the array is unsorted it will have a red background color.

Sorted
: If the array is sorted it will have a green background color.

3.2 Interactions

The shuffle experiment consists of the following interactions:

- Pick an element from one of the input arrays

The next section describes this interaction in detail.

3.2.1 Pick an element from one of the input arrays

There are two input arrays `Left` and `Right` in this experiment setup. The user has to move all the elements from each of these arrays into the initially empty `Result` array. The user has two buttons to do this. The `LEFT` button and the `RIGHT` button.

As described in the interface section, the `LEFT` button picks the first element from the `Left` array and places it at the end of the `Result` array. The `RIGHT` button is similar.

When we pick an element from one of the input arrays and place it in the `Result` array, that corresponding element is removed from the input array.

For example, consider the following setting:

```
LEFT = [10, 40, 50, 8]

RIGHT = [9, 3, 7]
```

```
RESULT = []
```

If we click on the `RIGHT` button, the system results in the following state:

```
LEFT = [10, 40, 50, 8]

RIGHT = [3, 7]

RESULT = [9]
```

Observe that the `Right` array goes from `[9, 3, 7]` to `[3, 7]` and the `RESULT` array goes from `[]` to `[9]`.

3.2.2 The final state of the system

Due to the terminating nature of the system, there is always a limited number of interactions possible that change the state of the system. The final state of the system is reached when all the elements from both the `Left` and `Right` arrays have moved to the `Result` array.

For example, one of the possible final state of the system in the example above can be as follows:

```
LEFT = []

RIGHT = []

RESULT = [9, 10, 40, 50, 8, 3, 7]
```

This is only one of the possible final states. There can be several possible final states because the final state is determined only by the length of the `Left`, `Right` and `Result` arrays and not the contents of the `Result` array (which can vary depending on the order of interactions).

The following conditions should be satisfied for the system to be in the final state:

- `Left` and `Right` arrays should be empty.
- length of `Result` = initial length of `Left` + initial length of `Right`.

4 Merge Strategy

4.1 Unsorted-Merge as a constrained shuffle experiment

The unsorted-merge experiment is a special case of the shuffle experiment with additional constraints. The constraints are applied to the interactions.

4.1.1 Interaction Contraints

The user can only pick an element to place in the result array from the input array that has larger number as the first element.

For example, consider the following scenario:

```
Left = [50, 30, 20, 55]
Right = [22, 80, 90]
```

In the above case, the shuffle experiment would allow the user to pick any element 50 or 22 to place in the result array. But, in the unsorted-merge experiment the user can only select the first element from the `Left` array (50) because it is greater than 22.

In case one of the arrays is empty, the user can pick the remaining numbers from the non-empty array one-by-one.

4.2 Interface

The interface for the unsorted-merge experiment is similar to the shuffle experiment, because of the similarity discussed above.

4.3 Interactions

The interactions are similar to the shuffle experiment except the fact that depending on which array has the larger first element, one of the `LEFT` or `RIGHT` buttons will always be disabled. If both arrays have equal first elements then and only then both `Left` and `Right` can be enabled at the same time.

5 Merge Algorithm

5.1 Interface

The interface of the sorted-merge experiment is similar to the unsorted merge experiment because the sorted merge experiment is a special case of the unsorted merge experiment.

There is functionally no difference between the sorted-merge and unsorted-merge experiments. The only difference is that the `Left` and `Right` arrays are gaurenteed to be sorted.

Due to the nature of the system, it is implied that the `Result` array will also be always sorted. So, none of the arrays will ever be unsorted in this experiment.

5.2 Interactions

The interactions in this experiment as exactly the same as that in the unsorted-merge experiment. Only the results of these interactions differ in the fact that the resulting array in this experiment will always be sorted whereas in the unsorted-merge experiment if the resulting array is sorted it is only by coincidence.

6 Recursive Mergesort

6.1 Interface and Interactions

The interface for the recursive merge experiment consists of a directed acyclic graph that represents the steps in the recursive split and merge operations in the mergesort algorithm.

The graph has the following properties:

6.1.1 Nodes

Function Node
nodes that represent functions. Source nodes of a function node represent inputs to the function and target node represents output of the function.

Applied Function Node
Function whose result has been evaluated and is stored in the target of the outgoing nodes.
Applicable Function Node
Function that has not been evaluated. Such a function node has no outgoing edge.
Composite Function
Function that is a composition of other functions. A composite function node can be expanded using an expansion rule that is specific to the function.
Primitive Function
Primitive function. Primitive function nodes cannot be expanded.

Data Node
nodes that represent data (arrays).

6.1.2 Mapping of node type to visual representation

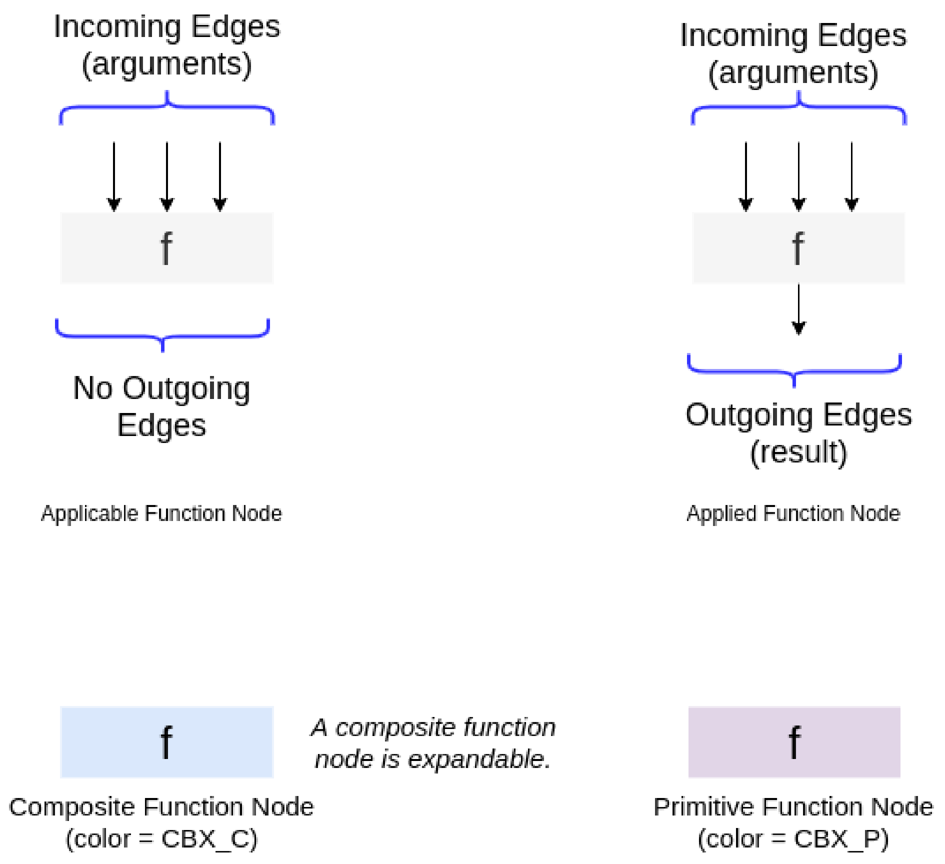

Incoming Edges (arguments)
f
No Outgoing Edges
Applicable Function Node
Incoming Edges (arguments)
f
Outgoing Edges (result)
Applied Function Node
f
A composite function node is expandable.
Composite Function Node (color = CBX_C)
f
Primitive Function Node (color = CBX_P)

Figure 1: Visual Representations of Different Types of Function Nodes

Table 1: visual-mapping

Node Type	Visual Representation	Text	Box (T/F)	Box Color
Data Node	None. Only values as text.	values	F	NA
Applied Function Node	Box with function name.	function name	T	default
Applicable Function Node	Box with function name.	function name	T	default
Composite Function Node	Box with function name.	function name	T	`CBX_C`
Primitive Function Node	Box with function name.	function name	T	`CBX_P`

NOTE : color names are placeholders for color codes. Colors seens in the diagrams in this document may not match those in the actual working system.

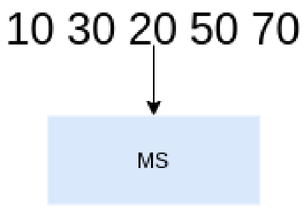
6.1.3 Initial

Initially the following elements are visible on the screen:

Input array
The array that we want to sort.

MS Function Node

A clickable box that represents the **Mergesort** operation.



The MS function node is `Applicable` as well as `Composite`. Hence two interactions are possible here.

Possible Interactions:

1. Expand

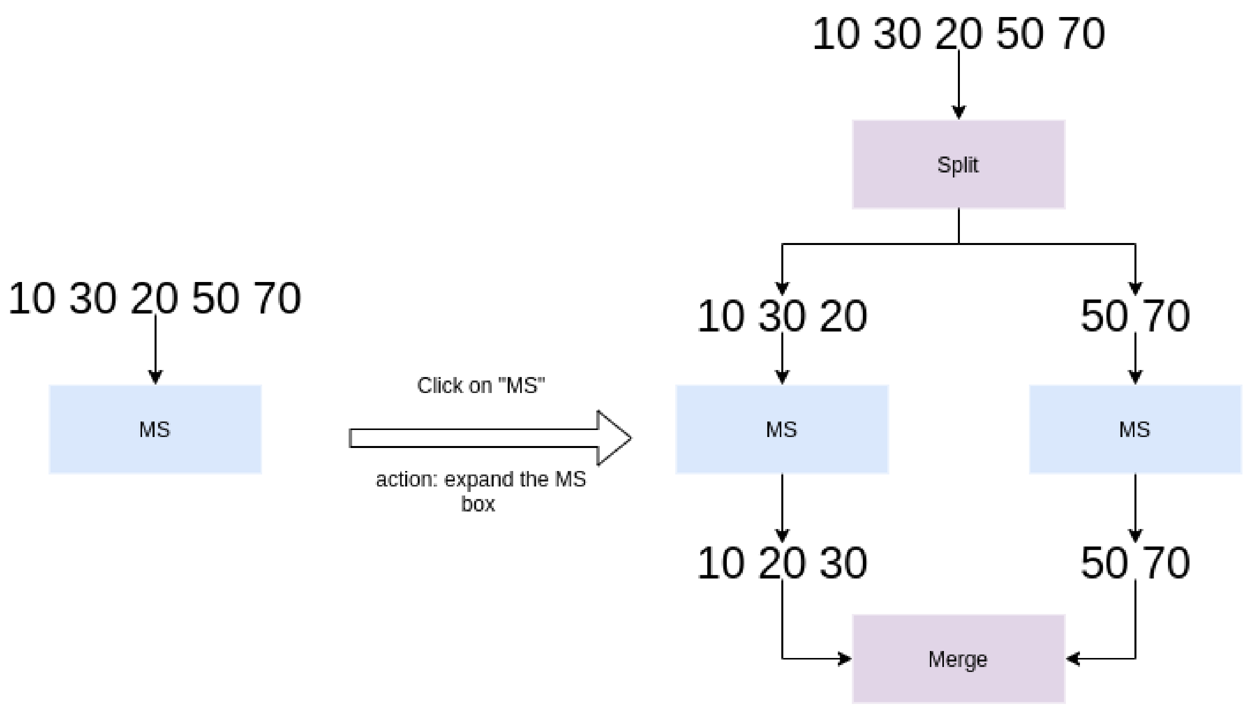
Expand the MS node.

2. Apply

Apply mergesort function to the incoming array.

Here we see the scenario where the MS node is expanded first in step 1 and then in the resulting graph, we apply the merge function.

6.1.4 Expand MS box



The user can click on the MS box. This corresponds to the MS expansion rule:

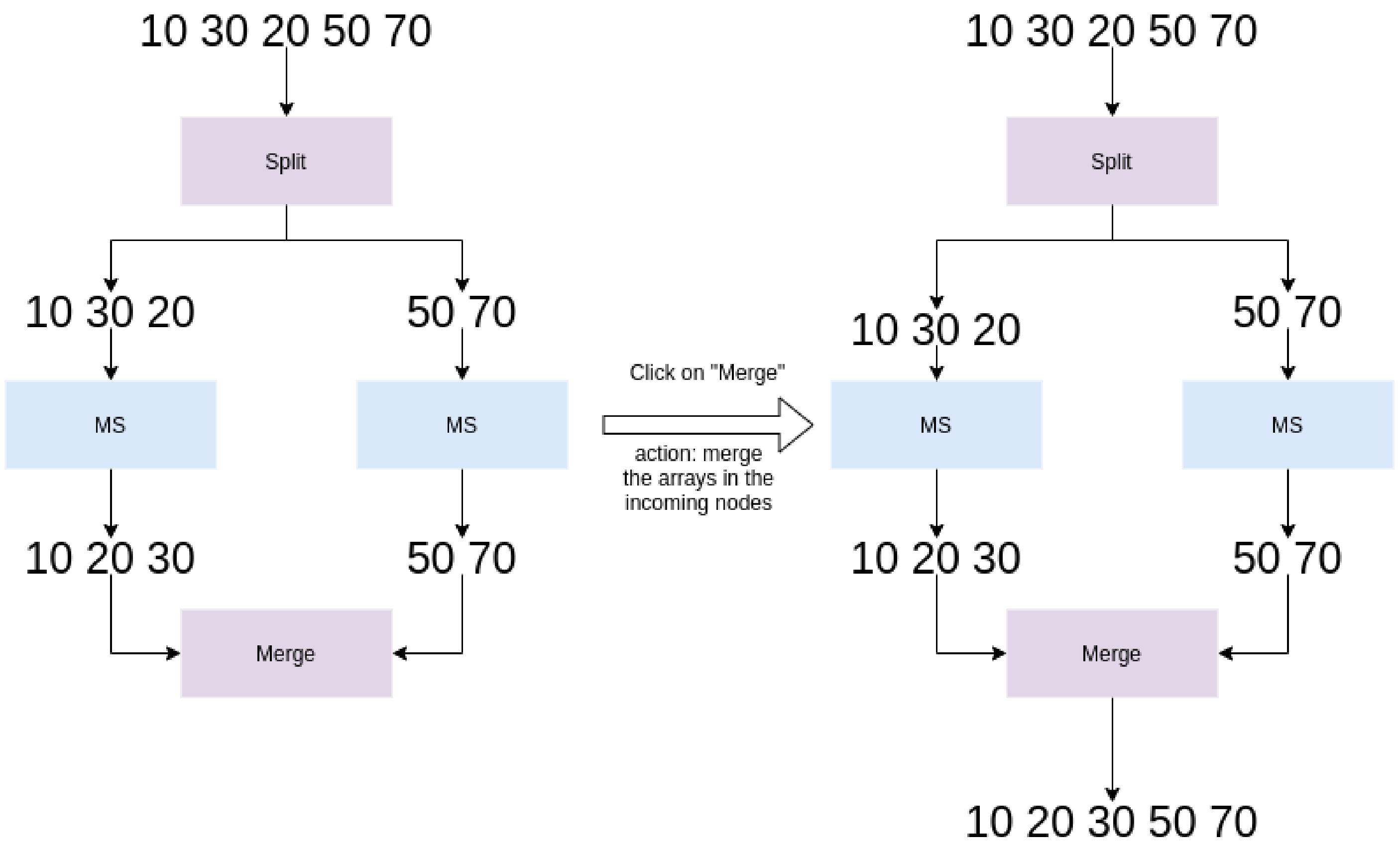
Listing 1: ms-expansion-rule

```
[(a -> ms)] ::= [ ( a -> split -> a1, a2 ),
                  ( a1 ->  ms -> a1' ),
                  ( a2 ->  ms -> a2' ),
                  ( (a1', a2') -> merge )
                ]
```

Notation:

- (n, …) -> m = many-to-one edge. **source nodes** => (n, …), **target node** => m
- n -> (m, …) = one-to-many edge. **source node** => n, **target nodes** => (m, …)
- [(n, … -> m, …), …] -> List of edges.
- x -> y = edge from node x to node y.
- a = input array node
- ms = merge sort function node
- a' = sort(a)
- a1 = sublist(a, 0, size(a)/2)
- a2 = sublist(a, size(a)/2, size(a))

6.1.5 Merge



The `Merge` node is a **primitive** node. This implies that it cannot be expanded. Hence the only action one can perform on a `Merge` node is function application (if the node is **applicable**).

In the example shown the above figure, the `Merge` node is **applicable**, so clicking on that node gives the result of applying merge on the incoming arrays.

7 Recursive Arbitrary Mergesort

7.1 INTERFACE

The interface for arbitrary merge is similar to that of recursive merge.

7.2 INTERACTIONS

The interactions in the arbitrary merge experiment are similar to the recursive merge experiment to some extent. The differences are dicussed below:

7.2.1 Merge Node

Recursive Merge

The merge node appears to the user as soon as an array is split into two at the middle. The user cannot apply the merge operation until the arrays that act as inputs to the merge node are not sorted, but the node is visible.

Arbitrary Merge

The user does not see the merge node automatically when the data node splits or is sorted. The user needs to click on two sorted data nodes in order for the merge node to appear. This difference lends the system the `aribitrary` nature. The user has the ability to select any two data nodes and merge them if they are sorted.

Date: 2021-04-19 Mon 00:00

Author: Venkatesh Choppella and Archit Goyal

Created: 2021-04-19 Mon 16:33

Validate