Predicting Forest Cover Type

Hassenforder Gaspard, Bohane Alexander, Siret Charles: GitHub

École Polytechnique

April 2023

Contents

1 Introduction

Before describing the algorithm, let's give a formal definition of connected components in the graph theory context. Let $G = (V, E)$ be an undirected graph where V is the set of vertices and E is the set of edges. $C = (C_1, C_2, \ldots, C_n)$ is the set of disjoint connected components in this graph where $(C_1 \cup C_2 \cup \ldots \cup C_n) = V$ and $(C_1 \cap C_2 \cap \ldots \cap C_n) = \emptyset$. For each connected component $C_i \in C$, there exists a path in G between any two vertices v_k and v_l where $(v_k, v_l) \in C_i$. Additionally, for any distinct connected component $(C_i, C_j) \in C$, there is no path between any pair v_k and v_l where $v_k \in C_i, v_l \in C_j$. Thus, the problem of finding all connected components in a graph is finding the C satisfying the above conditions.

Finding connected components in graphs is a cornerstone of graph theory with applications spanning numerous fields. Graphs are intrinsically useful to represent relationships and interactions, providing a versatile framework to represent and analyze complex networks. Connected components are pivotal in understanding the structure of a graph, representing sets of vertices within which every pair of vertices is connected by a path.

1. **Social Network Analysis**: In the realm of social networks, connected components can provide insight into the structure of communities, showing how groups of individuals are linked. This analysis can reveal information patterns and influential nodes within these networks, critical for understanding social dynamics and for targeted marketing strategies.

2. **Computer Networks**: Analyzing the resilience of computer networks against failures involves examining the network's decomposition into connected components upon node or link removal. This analysis is vital for designing robust networks that maintain connectivity under adverse conditions, ensuring reliable communication and data integrity.

3. **Image Processing**: Connected component labeling in digital images is a fundamental step in object recognition algorithms, where pixels belonging to the same object are identified and categorized together.

The pursuit of efficient algorithms for identifying connected components is driven by the necessity to handle large-scale graphs encountered in these domains. This paper explores an algorithm aimed at enhancing the computational efficiency and accuracy of finding connected components.

2 CCF Methods

The main idea of this algorithm is to successively apply CCF-Iterate and CCF-Dedup until no new pairs are found, indicating that all nodes are correctly mapped to their smallest component ID, thereby identifying all connected components.

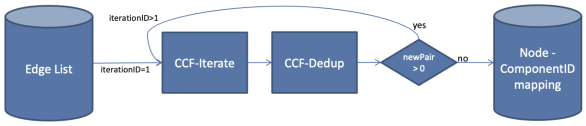

Figure 1: Connected Component Finder(CCF) Module

Our data is usually in the form (node1, node2), signifying that node1 is connected to node2. However, as we are agnostic to the direction of the graph and consider connectivity bidirectional, the mapping phase of this algorithm transforms (node1, node2) into (node1, node2), (node2, node1).

2.1 CCF-Iterate

CCF-Iterate, looks at every node and identifies the minimum neighbour (each node is associated with a number), if the node in question is the minimum out of all his neighbours we do nothing. If not we create a new edge between every non minimum node to the minimum node, for every new edge connected a variable called `NewPair`

This graph transformation preserves connectivity as it only creates new connections between nodes that have an intermediary node connecting them.

CCF-Iterate
map(key, value)
  emit($key, value$)
  emit($value, key$)

reduce($key, < iterable > values$)
  $min \leftarrow key$
  for each ($value \in values$)
    if($value < min$)
      $min \leftarrow value$
    valueList.*add*($value$)
  if($min < key$)
    emit(key, min)
    for each ($value \in valueList$)
      if($min \neq value$)
        Counter.NewPair.*increment*(1)
        emit($value, min$)

Figure 2. Alg.1 - CCF-Iterate

Figure 2: Pseudocode for CCF-Iterate with Map and Reduce

Following the CCF-Iterate job, there might be duplicate entries for node-component ID mappings due to the iterative process. The CCF-Dedup job deduplicates these entries, preparing the dataset for the next iteration of CCF-Iterate if

necessary.

```
CCF-Dedup
map(key, value)
    temp.entity1 ← key
    temp.entity2 ← value
    emit(temp, null)

reduce(key, < iterable > values)
    emit(key.entity1, key.entity2)
```

Figure 3: Pseudocode for CCF-Dedup

At the end of the process, we have that every node in a single component is connected to the minimum of that component and only to the minimum.

2.2 CCF-Iterate with Secondary Sorting

The secondary sort enhancement to the CCF algorithm improves space and time efficiency, particularly for large connected components. It utilizes a custom partitioning and sorting approach within the MapReduce framework to ensure that the values (node IDs) are passed to the reducer in a sorted manner. This way, the reducer can identify the minimum node ID (the first value) without needing to store all values in memory, reducing space complexity and potentially improving processing time.

Efficiency

The secondary sort approach avoids the need for storing all adjacent node IDs in memory, thus reducing the space complexity from $O(N)$ to $O(1)$ for the reduce phase, where N is the size of the largest connected component. This makes the algorithm significantly more efficient for processing large graphs with components containing millions of nodes.

Implementation

During the map phase, for each edge, both directions (a, b) and (b, a) are emitted to ensure that the adjacency list for each node can be constructed. In the reduce phase, thanks to the secondary sorting, the minimum adjacent node ID is received as the first value, allowing the reducer to immediately emit the necessary pairs without additional iterations over the data.

```
CCF-Iterate (w. secondary sorting)
map(key, value)
    emit(key, value)
    emit(value, key)

reduce(key, < iterable > values)
    minValue ← values.next()
    if(minValue < key)
        emit(key, minValue)
        for each (value ∈ values)
            Counter.NewPair.increment(1)
            emit(value, minValue)
```

Figure 4: Pseudocode for CCF-Iterate with secondary sorting

Here we have a visualization of a graph at every step of the algorithm. The process going from the first graph to the second is explained as follows:

1. Starting with A, it is the smallest of all its neighbors; therefore, we create no edge.
2. Selecting B, since C and D are bigger than its neighbor A, an edge is created between them, and an edge between B and A is created.
3. Selecting C, it simply connects to B as it is its smallest and only neighbor.
4. Selecting D, B is the smallest between B and E; therefore, the edges (D, B) and (E, B) are created.
5. Selecting E, it simply connects to D as it is its smallest and only neighbor.
6. Selecting F, it is the smallest of all its neighbors; therefore, we create no edge.
7. Selecting G, it simply connects to F as it is its smallest and only neighbor.
8. Selecting H, it simply connects to F as it is its smallest and only neighbor.

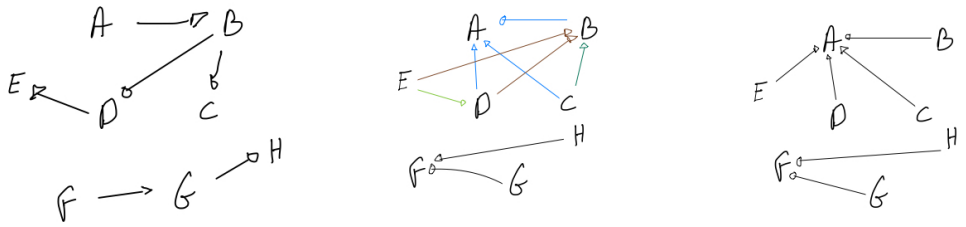

Figure 5: Graphs after each iteration of CCF-Iterate

3 Datasets

We have chosen 2 different datasets to test out these 2 algorithms, they both come from the Stanford Network Analysis Project (SNAP), a collection of more than 50 large network datasets from tens of thousands of nodes and edges to tens of millions of nodes and edges. In includes social networks, web graphs, road networks, internet networks, citation networks, collaboration networks, and communication networks.

3.1 LiveJournal

LiveJournal is a free on-line community with almost 10 million members; a significant fraction of these members are highly active. (For example, roughly 300,000 update their content in any given 24-hour period.) LiveJournal allows members to maintain journals, individual and group blogs, and it allows people to declare which other members are their friends they belong.

Statistic	**Value**
Nodes	4847571
Edges	68993773
Nodes in largest Connected Component	4843953
Edges in largest Connected Component	68983820
Diameter	16
Size	1.08 GB

Table 1: Statistics for Livejournal

Note that the largest Component consists of 99.9998% of the edges and 99.999% of the nodes of the entire graph.

3.2 Google web graph

Google web graph is a dataset where nodes represent web pages and directed edges represent hyperlinks between them. The data was released in 2002 by Google as a part of Google Programming Contest

Statistic	**Value**
Nodes	875713
Edges	5105039
Nodes in largest Connected Component	855802
Edges in largest Connected Component	5066842
Diameter	21
Size	75.4 MB

Table 2: Statistics for Google Web graph

Note that the largest Component consists of 97% of the edges and 99% of the nodes of the entire graph.

3.3 Epinions social network graph

This is a who-trust-whom online social network of a a general consumer review site Epinions. Members of the site can decide whether to "trust" each other. All the trust relationships interact and form the Web of Trust which is then combined with review ratings to determine which reviews are shown to the user.

Statistic	**Value**
Nodes	75879
Edges	508837
Nodes in largest Connected Component	75877
Edges in largest Connected Component	508836
Diameter	14
Size	5.7 MB

Table 3: Statistics for Epinions social network graph

From this observation, it becomes apparent that the graph's largest connected component encompasses all but two nodes, and all but one edge, indicating that the graph consists of only two separate components. This means that there are two individuals who exclusively trust one another, while all other individuals are connected through a chain of mutual trust with everyone else in the graph.

4 Results

4.1 Iterations

According to the paper [1], the worst case scenario for the number of iterations needed with this algorithm is d+1 where d is the diameter of the network. The worst case happens when the min node in the largest connected component is an end-point of the largest shortest-path. The best case scenario takes d/2+1 iterations when the min node should be at the center of the largest shortest-path.

Dataset	Number of Iterations	Worst Case Scenario	Best Case Scenario	Diameter
Epinions	4	15	8	14
Google	7	22	11	21
LiveJournal	6	17	9	16

Table 4: Table of number of iterations

In our investigation, we encountered an intriguing anomaly where our algorithm completed its task with significantly fewer iterations than anticipated by the best case scenario. This outcome is puzzling and deviates from our initial predictions.

4.2 Connected Component Size Distribution

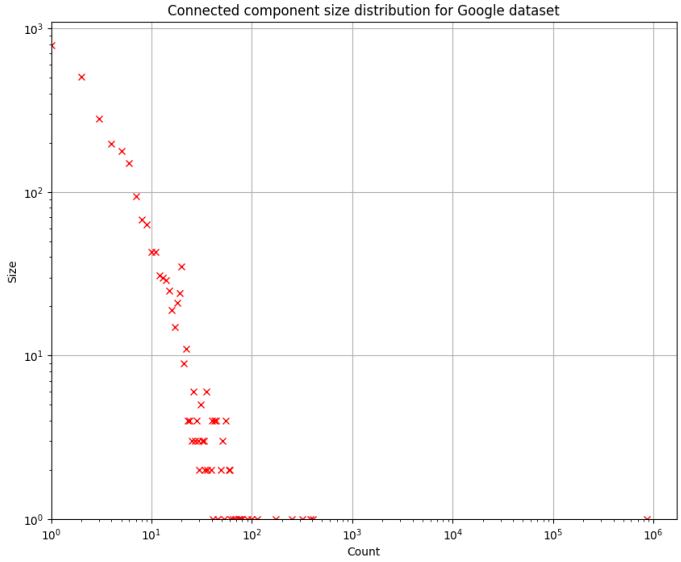

Figure 6: Connected Component Size Distribution for Google Graph

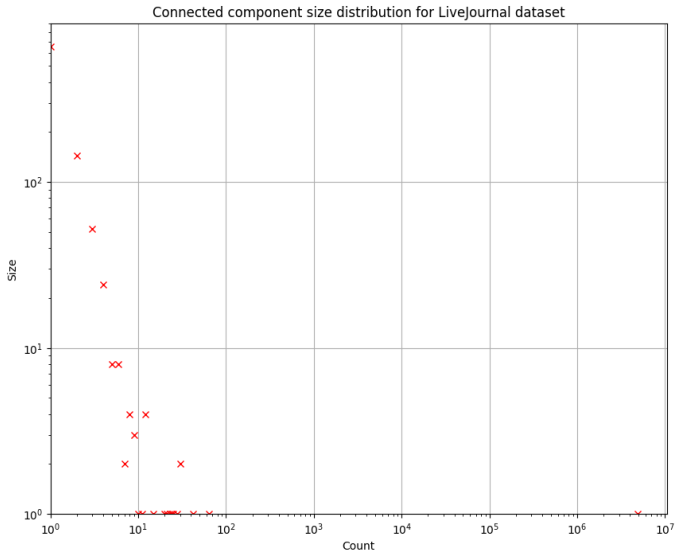

Figure 7: Connected Component Size Distribution for LiveJournal Graph

There is no point of plotting the distribution for Epinions as it only has 2 components

4.3 Python using spark

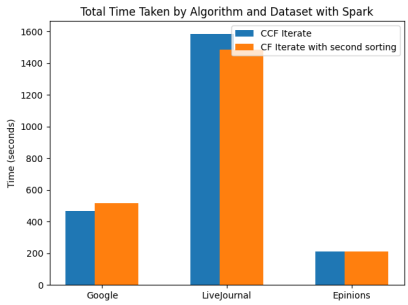

Figure 8: Computation time for each dataset and each method using Spark

As explained by the authors, the "secondary sorting" improvement is only relevant with relatively large graphs (with millions of nodes). As we can see above that for the Google graph the CCF-Iterate with secondary sort is slower than the classical CCF-Iterate, the Epinions graph is so small we cant see the difference. For our largest graph (Livejournal with 69 Million nodes), we can clearly see the advantages of secondary sort.

4.4 Scala using Databricks

The computation time using Databricks is significantly higher than on a Jupyter notebook; however, they were run on different machines, so we can attribute it to that. But what is very interesting is that here, the CCF-Iterate with secondary sorting outperforms CCF-Iterate on every graph, even the smallest one. It seems like the secondary sorting optimization works better on Databricks. This may be thanks to Databricks using Spark as its core engine, it adds

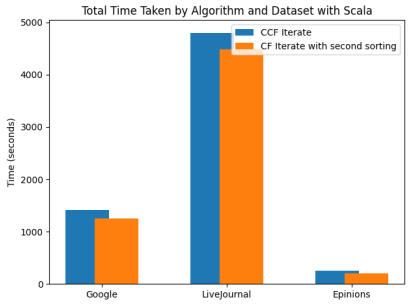

Figure 9: Computation time for each dataset and each method using scala

its own optimizations and features on top and therefore may handle shuffling and sorting more efficiently so that we don't need very large graphs to see the advantages of CCF-Iterate with secondary sorting.

References

[1] Kardes, H., Agrawal, S., Wang, X., & Sun, A. (2014). CCF: Fast and scalable connected component computation in MapReduce `https://www.cse.unr.edu/~hkardes/pdfs/ccf.pdf`

5 Appendix:

5.1 Python

```
def ccf_iterate(rdd: RDD) -> Tuple[RDD, int]:
    # Map part of algorithm need to transform (A,B) into (A,B,),(B,A)
    inverted_rdd = rdd.map(lambda pair: (pair[1], pair[0]))
    full_graph = rdd.union(inverted_rdd).groupByKey().mapValues(list).partitionBy
        (100).cache()

    # Processing the graph: converting to integers, extracting min, and filtering
    processed_graph = (
        full_graph # Ensure all are integers
        .map(lambda node: (int(node[0]), list(map(int, node[1])))  # Attach
            minimum value from list
        .map(lambda node: (node[0], node[1], min(node[1]))) # (key, list of
            neighbours, min neighbour)
        .filter(lambda node: node[0] > node[2])  # if key is the the minimum then
             no need to anything as all nodes already connected to min(key)
        .map(lambda node: (node[0], [value for value in node[1] if value != node
            [2]], node[2]))  # Remove min value from list (key, list of non min
            neighbours, min neighbour)
    )

    # Summing the lengths of the modified lists to count new pairs
    total_new_pairs = processed_graph.map(lambda node: len(node[1])).sum()

    # Emitting pairs for final output and deduplication
    new_pairs_1 = processed_graph.map(lambda node: (node[0], node[2])) #
        connecting key and min neighbour
    new_pairs_2 = processed_graph.flatMap(lambda node: [(value, node[2]) for
        value in node[1]]) # connecting all non min neighbours with min neighbour
    all_emits = new_pairs_1.union(new_pairs_2)
```

```
    # Final deduplication and preparation of output
    output_graph = (
        all_emits
        .map(lambda pair: ((pair[0], pair[1]), None))
        .reduceByKey(lambda x, _: x)  # Using _ for unused variable in lambda
        .map(lambda pair: pair[0])
    )

    return output_graph, total_new_pairs
```

```
def ccf_iterate_sorting(rdd: RDD) -> Tuple[RDD, int]:
    # Map part of algorithm need to transform (A,B) into (A,B,),(B,A)
    inverted_rdd = rdd.map(lambda pair: (pair[1], pair[0]))
    combined_graph = rdd.union(inverted_rdd).groupByKey().mapValues(list).
       partitionBy(100).cache()

    # Clean and prepare the graph
    prepared_graph = (
        combined_graph
        .map(lambda node: (int(node[0]), sorted(set(map(int, node[1])))))  #
           Ensure uniqueness and sort
        .map(lambda node: (node[0], node[1], node[1][0]))  # (key, list of
           neighbours, min neighbour)
        .filter(lambda node: node[0] > node[2])  # if key is the the minimum then
            no need to anything as all nodes already connected to min(key)
        .map(lambda node: (node[0], node[1][1:], node[2]))  # Remove min value
           from list (key, list of non min neighbours, min neighbour)
    )

    # Calculate the new pairs and prepare for further actions
    count_new_pairs = prepared_graph.map(lambda node: len(node[1])).sum()
    new_pairs_1 = prepared_graph.map(lambda node: (node[0], node[2]))# connecting
        key and min neighbour
    new_pairs_2 = prepared_graph.flatMap(lambda node: [(value, node[2]) for value
        in node[1]]) # connecting all non min neighbours with min neighbour
    total_emission = new_pairs_1.union(new_pairs_2)

    # Deduplicate pairs
    output_graph = (
        total_emission
        .map(lambda pair: ((pair[0], pair[1]), None))
        .reduceByKey(lambda x, y: x)
        .map(lambda pair: (pair[0][0], pair[0][1]))
    )

    return output_graph, count_new_pairs
```

```
def find_connected_components(graph, method):
    start_time = time.time()
    new_pairs = 1
    iter = 0
    while new_pairs != 0: # stop iterating when algorithm stabilises and doesn't
       create new_pairs
        print(method.__name__)

        try:
```

```
            graph, new_pairs = method(graph)
        except Exception as e:
            print("Method not found or failed with error:", e)
            break

        print("Iteration: ", iter, " - New pairs: ", new_pairs)

        iter += 1

    end_time = time.time()
    return graph, iter, end_time - start_time

# Example of running function
data_google = sc.textFile(file_path)

# clean the data, since from same source they have the same format
def format_data(x: str) -> Tuple[int, int]:
    if "#" in x:
        return None
    else:
        return tuple(map(int, x.split("\t")))

data_google = data_google.map(format_data).filter(lambda x: x is not None)

# Initialize an empty dictionary
results = {}
methods = [ccf_iterate, ccf_iterate_sorting]

for method in methods:
    result_google, num_iter, time_taken = find_connected_components(data_google,
       method)
    results["Google"] = [time_taken, num_iter, method.__name__]
    print(f"{method.__name__} took {time_taken} seconds and {num_iter} iterations
       to complete, and found {n_components} components in graph:")
```

5.2 Scala

```
def ccfIterate(rdd: RDD[(Int, Int)]): (RDD[(Int, Int)], Long) = {
  // Map part of algorithm need to transform (A,B) into (A,B),(B,A)
  val invertedRDD = rdd.map(pair => (pair._2, pair._1))
  val fullGraph = rdd.union(invertedRDD).groupByKey().mapValues(_.toList).
     partitionBy(new HashPartitioner(100)).cache()

  // Processing the graph: converting to integers, extracting min, and filtering
  val processedGraph = fullGraph
    .map { case (node, neighbors) =>
      (node, neighbors.map(_.toInt), neighbors.map(_.toInt).min) // Ensure all
         are integers and attach minimum value from list (key, list of neighbors,
          min neighbor)
    }
    .filter { case (node, _, minNeighbor) =>
      node > minNeighbor // if key is the min we dont need to do anything
    }
    .map { case (node, neighbors, minNeighbor) =>
```

```
      (node, neighbors.filter(_ != minNeighbor), minNeighbor) // Remove min value
          from list ( key , list of non min neighbours , min neighbour )
    }

  // Summing the lengths of the modified lists to count new pairs
  val totalNewPairs = processedGraph.map { case (_, neighbors, _) =>
    neighbors.length
  }.sum().toLong

  // Emitting pairs for final output and deduplication
  val new_pairs_1 = processedGraph.map { case (node, _, minNeighbor) =>
    (node, minNeighbor)
  } // Creating edge between key and min neighbor
  val new_pairs_2 = processedGraph.flatMap { case (_, neighbors, minNeighbor) =>
    neighbors.map(neighbor => (neighbor, minNeighbor))
  } // Creating edges between non min neighbours and min neighbor
  val allEmits = new_pairs_1.union(new_pairs_2)

  // Final deduplication and preparation of output
  val output_graph = allEmits
    .map(pair => (pair, None))
    .reduceByKey((x, _) => x) // Using _ for unused variable in lambda
    .map(_._1)

  (output_graph, totalNewPairs)
}
```

```
def ccfIterateSorting(rdd: RDD[(Int, Int)]): (RDD[(Int, Int)], Long) = {
  // Map part of algorithm need to transform (A,B) into (A,B),(B,A)
  val invertedRDD = rdd.map(pair => (pair._2, pair._1))
  val combinedGraph = rdd.union(invertedRDD).groupByKey().mapValues(_.toList).
     partitionBy(new HashPartitioner(100)).cache()

  // Clean and prepare the graph
  val preparedGraph = combinedGraph
    .map { case (node, neighbors) => (node, neighbors.map(_.toInt).toSet.toList.
       sorted) } // Ensure uniqueness and sort
    .map { case (node, sortedNeighbors) => (node, sortedNeighbors,
       sortedNeighbors.head) } // Attach min value
    .filter { case (node, _, minNeighbor) => node > minNeighbor } // Filter based
        on condition
    .map { case (node, neighbors, minNeighbor) => (node, neighbors.tail,
       minNeighbor) } // Update list by removing min

  // Calculate the new pairs and prepare for further actions
  val countNewPairs = preparedGraph.map { case (_, neighbors, _) => neighbors.
     length.toLong }.reduce(_ + _)
    // Emitting pairs for final output and deduplication
  val new_pairs_1 = processedGraph.map { case (node, _, minNeighbor) =>
    (node, minNeighbor)
  } // Creating edge between key and min neighbor
  val new_pairs_2 = processedGraph.flatMap { case (_, neighbors, minNeighbor) =>
    neighbors.map(neighbor => (neighbor, minNeighbor))
  } // Creating edges between non min neighbours and min neighbor
  val allEmits = new_pairs_1.union(new_pairs_2)
```

```
  // Deduplicate pairs
  val output_graph = allEmits
    .map(pair => (pair, ()))
    .reduceByKey((x, _) => x)
    .map(_._1)

  (output_graph, countNewPairs)
}
```

```
def runCCFAlgorithm(data: RDD[(Int, Int)], ccfFunction: (RDD[(Int, Int)]) => (RDD
    [(Int, Int)], Long)): Unit = {
  // Initialize variables for the loop
  var currentGraph = data
  var newPairs = Long.MaxValue
  var iteration = 0

  // Start the timer
  val startTime = System.nanoTime()

  // Loop until no new pairs are generated
  while (newPairs != 0) {
    val result = ccfFunction(currentGraph)
    currentGraph = result._1
    newPairs = result._2
    iteration += 1
    println(s"Iteration␣$iteration:␣$newPairs␣new␣pairs␣found")
  }

  // Stop the timer
  val endTime = System.nanoTime()

  // Calculate and print the total time taken
  val duration_seconds = (endTime - startTime) / 1e9d // convert to seconds
  println(s"CCF␣algorithm␣stabilized␣after␣$iteration␣iterations.␣Total␣time␣
      taken:␣$duration_seconds␣seconds.")
}

// Example of running function
def cleandata(x: String): Option[(Int, Int)] = {
    if (x.contains("#")) None
    else Some(x.split("\t").map(_.toInt) match { case Array(a, b) => (a, b) })
}

val dataGoogle: RDD[(Int, Int)] = spark.sparkContext.textFile(filePath_google).
    flatMap(cleandata)
runCCFAlgorithm(dataGoogle, ccfIterate _)
```