💡

Technical Report For Simple Simulator Circuit

- Firstly , I got the logic & Backend core inspiration from https://github.com/FarahaniMehrshad/NaiveCircuitSimulator & https://github.com/chewoo/L2Spice
- I enhanced my code syntax many times using AI
- The Code consists of Main Three Classes :
    - Node
    - Element
    - Circuit Core
    - Circuit GUI (Made totally with AI , Not explained in the Report)

Breakdown For Every Class :

- Node Class :
    - Attributes :
        - Name
        - List of the elements that is connected to this node
- Element Class :
    - Attributes :
        - Name
        - Voltage
        - Current
        - Resistance
        - First Node
        - Second Node
        - Right Element & Left Element (will be more explained in merge & unmerge)
        - Children Connections   (will be more explained in merge & unmerge)
- Circuit Core :
    - Attributes :
        - List of elements in the circuit
        - List of nodes in the circuit
        - Connection Types :

        ```
            NONE = 0
            SERIES = 1
            PARALLEL = 2
        ```

    - Functions :
        - addWire / addResistor / addBattery
            - Validate & Call the constructor of the Element Class :

            ```
            if negative_side == positive_side:
             raise ValueError("Cannot connect node to itself")
            ```

```
return self.add_element(name, 0, 0, 0, negative_side, positive_side)add_eleme
nt(name, 0, 0, 0, negative_side, positive_side)
```

- Search Element node :
  - Search for name of the element / node of the element object :

```
def search_element(self, name):
 return next((element for element in self.elements if element.name == name),
None)

     def search_node(self, name):
        return next((node for node in self.nodes if node.name == name), None)
```

- Search or create a Node if not exist :
  - Search for name of the Node and create a new one if not exist :

```
    def search_or_create_node(self, name):
        node = self.search_node(name)
        if not node:
            node = Node(name)
            self.nodes.append(node)
        return node
```

- Connection :
  - It determines the kind of connection between two node :

```
    def connection(self, el1, el2):
        # If only two elements left, consider them series
        if len(self.elements) == 2:
            return self.SERIES

        common_nodes = []
        if el1.node1 == el2.node1:
            common_nodes.append(el1.node1)
        if el1.node2 == el2.node2:
            common_nodes.append(el1.node2)
        if el1.node1 == el2.node2:
            common_nodes.append(el1.node1)
        if el1.node2 == el2.node1:
            common_nodes.append(el1.node2)

        # Series
        if len(common_nodes) == 1:
            common_node = common_nodes[0]
            if len(common_node.elements) == 2:
                return self.SERIES

        # Parallel
        if len(common_nodes) == 2:
            return self.PARALLEL

        return self.NONE
```

- Merge :
  - It takes two components and merge them in a new (Assumption) one with new values :

```
    def merge(self, el1, el2):
        cxn = self.connection(el1, el2)
        if cxn == self.NONE:
            raise ValueError("Cannot merge elements")

        # Battery direction handling
        if self.is_battery(el1) and self.is_battery(el2):
            if el1.node1 == el2.node1 or el1.node2 == el2.node2:
                el2.voltage *= -1

        name = f"{el1.name}+{el2.name}"
        node1, node2 = None, None

        if cxn == self.SERIES:
            resistance = el1.resistance + el2.resistance
            voltage = el1.voltage + el2.voltage

            # Find common and opposite nodes
            common_node = None
            if el1.node1 == el2.node1:
                common_node = el1.node1
            elif el1.node2 == el2.node2:
                common_node = el1.node2
            elif el1.node1 == el2.node2:
                common_node = el1.node1
            elif el1.node2 == el2.node1:
                common_node = el1.node2

            node1 = el1.node1 if el1.node1 != common_node else el1.node2
            node2 = el2.node1 if el2.node1 != common_node else el2.node2

        elif cxn == self.PARALLEL:
            if el1.resistance < 0.000001 or el2.resistance < 0.000001:
                raise ValueError("Short circuit")

            if abs(el1.voltage) > 0.00001 or abs(el2.voltage) > 0.00001:
                raise ValueError("Cannot merge parallel elements with voltag
e")

            resistance = 1.0 / (1.0 / el1.resistance + 1.0 / el2.resistance)
            node1, node2 = el1.node1, el1.node2

        new_element = self.add_element(name, voltage, 0, resistance, node1.na
me, node2.name)
        new_element.left = el1
        new_element.right = el2
        new_element.children_connections = cxn

        self.elements.remove(el1)
        self.elements.remove(el2)

        return new_element
```

- It stores the elements that were merged in the attributes (Left & Right) -and stores the type of connection too in (Children_Connection)
- It removes those children components from the circuits list

- Unmerge :
  - It calculates the currents & voltages of elements :

```
def unmerge(self, element):
    if not element.left and not element.right:
        if element not in self.elements:
            self.elements.append(element)
        if element not in element.node1.elements:
            element.node1.elements.append(element)
        if element not in element.node2.elements:
            element.node2.elements.append(element)

        if not self.is_battery(element):
            element.voltage = element.current * element.resistance

        return

    left = element.left
    right = element.right
    current = element.current

    # Divide current based on connection type
    if element.children_connections == self.SERIES:
        left.current = current
        right.current = current
    elif element.children_connections == self.PARALLEL:
        # Distribute current based on resistance
        if left.resistance < 0.000001:
            left.current = current
        elif right.resistance < 0.000001:
            right.current = current
        else:
            ratio = left.resistance / right.resistance
            left.current = current / (ratio + 1)
            right.current = ratio * current / (ratio + 1)

    self.unmerge(left)
    self.unmerge(right)

    # Remove merged element
    element.node1.elements.remove(element)
    element.node2.elements.remove(element)
    self.elements.remove(element)
```

- **Check if the element is merged (has left and right connections):**
  - If both `element.left` and `element.right` are `None`, it means the element hasn't been merged.
    - If the element isn't already in the `self.elements` list (which stores all elements in the circuit), it will be added.
    - Similarly, if the element isn't already in the `node1.elements` or `node2.elements` lists (which track all elements connected to the respective nodes), it will be added to those as well.
- **Voltage Calculation:**
  - If the element isn't a battery (`self.is_battery(element)` returns `False`), the voltage is calculated using the formula `V = I * R` (Ohm's Law), where `current` is the current flowing through the element and `resistance` is its resistance.
- **Handle Merged Elements (Left and Right Connections):**
  - If the element is merged (i.e., it has `left` and `right` elements):
    - The `current` is divided between the `left` and `right` elements based on the connection type (`SERIES` or `PARALLEL`).

- **Handling Current in Series and Parallel Connections:**
  - **Series Connection:**
    - In a series connection, the same current flows through both the left and right elements, so both `left.current` and `right.current` are set to the original `current`.
  - **Parallel Connection:**
    - In a parallel connection, the current is distributed based on the resistance of the two elements:
      - If one of the elements has almost zero resistance (near a short circuit), the current flows entirely through that element.
      - If both resistances are non-zero, the current is distributed based on the ratio of their resistances. The formula used here ensures that the element with the smaller resistance gets a larger share of the current.
- **Recursive Unmerge:**
  - After adjusting the currents for the left and right elements, the `unmerge` function is called recursively on both `left` and `right`. This ensures that if the left and right elements are also merged, their currents are also appropriately adjusted.
- **Remove the Merged Element:**
  - Finally, the function removes the element from the respective node's `elements` list (`node1.elements` and `node2.elements`), and also from the main `self.elements` list. This effectively "unmerges" the element from the circuit.

- Validate :
  - Making some validations :

```
    def validate(self):
        if not self.elements:
            raise ValueError("No elements in the circuit")

        battery_count = sum(1 for element in self.elements if self.is_battery
(element))
        resistor_count = sum(1 for element in self.elements if element.resist
ance > 0)

        if battery_count == 0:
            raise ValueError("No voltage source")
        if resistor_count == 0:
            raise ValueError("No resistor")

        for element in self.elements:
            if len(element.node1.elements) < 2 or len(element.node2.elements)
< 2:
                raise ValueError("Elements are not properly connected")
```

- Remove and bind elements :
  - Used to remove an element and reconnect the rest elements together

```
    def remove_and_bind_element(self, element):


        # Save the first node and identify the second node
        saved_node = element.node1
        not_saved_node = element.node2

        # Check for parallel connections that would create a short circuit
        is_parallel = False
```

```
        # Check parallel connections on the first node
        for neighbor in saved_node.elements:
            if neighbor == element:
                continue
            if self.connection(element, neighbor) == self.PARALLEL:
                is_parallel = True
                break

        # If not parallel on first node, check second node
        if not is_parallel:
            for neighbor in not_saved_node.elements:
                if neighbor == element:
                    continue
                if self.connection(element, neighbor) == self.PARALLEL:
                    is_parallel = True
                    break

        # Throw error if parallel connections exist and element is a wire
        if is_parallel and self.is_wire(element):

            raise ValueError("Short circuit detected")

        # Rebind elements if no parallel connections
        if not is_parallel:
            # Create a copy of the elements list to avoid modification during
iteration
            original_elements = not_saved_node.elements.copy()

            for other in original_elements:
                # Skip the element being removed
                if other == element:
                    not_saved_node.elements.remove(element)
                    continue

                # Rebind the node references
                if other.node1 == not_saved_node:
                    other.node1 = saved_node
                elif other.node2 == not_saved_node:
                    other.node2 = saved_node

                # Remove from old node and add to saved node
                not_saved_node.elements.remove(other)
                saved_node.elements.append(other)

        # Remove the now-empty node if no elements are connected
        if len(not_saved_node.elements) == 0:

            self.nodes.remove(not_saved_node)
            not_saved_node = None

        # Remove the element from its original nodes
        element.node1.elements.remove(element)
        if not_saved_node is not None:
            element.node2.elements.remove(element)

        # Remove the element from the circuit

        self.elements.remove(element)
```

- **Saving the Nodes**:
    - The function starts by saving the two nodes (`node1` and `node2`) that the element is connected to. These are stored as `saved_node` and `not_saved_node`.
- **Checking for Parallel Connections (Short Circuit Detection)**:
    - The function checks if removing the element will create a parallel connection (which could cause a short circuit).
    - First, it checks the elements connected to `saved_node`. If any of them are in parallel with the current element, the `is_parallel` flag is set to `True`.
    - If no parallel connections are found for the first node, the function then checks the second node (`not_saved_node`) in the same way.
    - If a parallel connection exists and the element being removed is a wire, a `ValueError` is raised, indicating a potential short circuit.
- **Rebinding Connected Elements**:
    - If no parallel connections were found, the function proceeds to rebind the elements connected to `not_saved_node` to `saved_node`.
    - It creates a copy of the list of elements connected to `not_saved_node` to avoid modifying the list while iterating through it.
    - For each connected element (`other`), the function checks if it's already connected to `not_saved_node` and, if so, changes its node reference to `saved_node`. Then, the element is moved from the list of `not_saved_node` to `saved_node`.
- **Removing Empty Node**:
    - After re-binding the elements, the function checks if `not_saved_node` is now empty (i.e., no elements are connected to it).
    - If the node is empty, it is removed from the list of nodes in the circuit and set to `None`.
- **Removing the Element**:
    - Finally, the function removes the element from the list of elements connected to `node1` and `node2`, and then removes it from the overall list of elements in the circuit.

▪ Solve :

- simplify a circuit by removing unnecessary elements and merging the remaining ones to calculate the final current

```
def solve(self):


    self.validate()

    # Remove wires
    i = 0
    while i < len(self.elements):
        element = self.elements[i]
        if self.is_wire(element):
            self.remove_and_bind_element(element)
            i = 0  # Restart the process
        else:
            i += 1

    allow_merge_with_battery = False
    while len(self.elements) != 1:
        merged = False
        for i in range(len(self.elements) - 1):
            for j in range(i + 1, len(self.elements)):
                el1, el2 = self.elements[i], self.elements[j]
                cxn = self.connection(el1, el2)
```

```
                    if cxn == self.NONE:
                        continue
                    if not allow_merge_with_battery:
                        if self.is_battery(el1) or self.is_battery(el2):
                            continue

                    self.merge(el1, el2)
                    merged = True
                    break
                if merged:
                    break

            if not merged:
                if not allow_merge_with_battery:
                    allow_merge_with_battery = True
                else:
                    self.is_dirty = True
                    raise ValueError("Circuit cannot be reduced to a single e
lement")

        # Calculate final current
        leftover_element = self.elements[0]
        leftover_element.current = leftover_element.voltage / leftover_elemen
t.resistance

        self.unmerge(leftover_element)
```

- **Validate the Circuit:**
  - Before simplifying, the circuit's validity is checked with the `validate` method. This ensures there are no structural issues with the circuit.
- **Remove Wires**
  - The method loops through all elements in the circuit to remove any wires using `remove_and_bind_element`.
  - If a wire is removed, the loop restarts (`i = 0`) to account for any changes to the circuit structure. Otherwise, the index increments (`i += 1`).
- **Merge Elements**
  - A flag `allow_merge_with_battery` starts as `False`. This prevents merging batteries with other elements at first.
  - The method iterates through pairs of elements (`el1` and `el2`) to determine if they can be merged:
    - If `connection(el1, el2)` returns a value indicating a connection (`SERIES` or `PARALLEL`), the elements are eligible for merging.
    - Batteries are skipped unless `allow_merge_with_battery` is `True`.
    - If the elements are merged, the loop breaks to restart with the updated circuit.
  - If no elements are merged in an iteration:
    - `allow_merge_with_battery` is set to `True`, allowing batteries to be considered for merging in subsequent iterations.
    - If batteries are already allowed and no merges occur, the circuit is marked as unsolvable and an exception is raised.
- **Final Calculation**
  - Once the circuit is reduced to a single element, the current is calculated using Ohm's Law: I=RV, where I is the current, V is the voltage, and R is the resistance.

- **Unmerge Elements**
  - The `unmerge` method is called on the last remaining element to restore the circuit to its original form, with updated values reflecting the calculations.