Detailed Algorithm Analysis: AST-Automaton Minimization

Nnamdi Michael Okpala

December 13, 2024

1 Core Data Structures

1.1 StateNode Representation

For each state-node pair (q, n) where $q \in Q$ and n is an AST node:

$$\text{StateNode}(q, n) = \begin{cases} \text{type}: & Q \times N \\ \text{transitions}: & \Sigma \to (Q \times N) \\ \text{ast\_children}: & \text{Set}(N) \\ \text{equivalence\_class}: & \mathbb{N} \end{cases}$$

1.2 Tree Structure

The AST node structure:

$$\text{Node} = \begin{cases} \text{value}: & \Sigma \cup Q \\ \text{children}: & \text{List(Node)} \\ \text{parent}: & \text{Node} \cup \{\perp\} \\ \text{type}: & \{\text{STATE}, \text{SYMBOL}\} \end{cases}$$

Algorithm 1 Initialize Combined Structure

```
procedure InitializeStructure(A, T)
    stateNodes ← ∅
    nodeMap ← new HashMap()
    for q ∈ Q do
        n ← FindCorrespondingNode(q, T)
        sn ← new StateNode(q, n)
        stateNodes ← stateNodes ∪ {sn}
        nodeMap[q] ← sn
    end for
    for sn ∈ stateNodes do
        for σ ∈ Σ do
            nextState ← δ(sn.state, σ)
            nextNode ← nodeMap[nextState]
            sn.transitions[σ] ← nextNode
        end for
        sn.ast_children ← GetASTChildren(sn.node)
    end for
    return (stateNodes, nodeMap)
end procedure
```

2 Algorithm Details

2.1 Initialization Phase

2.2 Equivalence Class Construction

Algorithm 2 Build Initial Equivalence Classes

```
procedure BuildEquivClasses(stateNodes)
    classes ← new List()
    accepting ← {sn ∈ stateNodes : sn.state ∈ F}
    nonAccepting ← {sn ∈ stateNodes : sn.state ∉ F}
    classes.add(accepting)
    classes.add(nonAccepting)
    changed ← true
    while changed do
        changed ← false
        for class ∈ classes do
            splits ← SplitByTransitions(class)
            if |splits| > 1 then
                classes.remove(class)
                classes.addAll(splits)
                changed ← true
            end if
        end for
    end while
    return classes
end procedure
```

2.3 AST-Aware State Splitting

2.4 Minimization Algorithm

3 Complexity Analysis

3.1 Time Complexity

For an automaton with $|Q|$ states and AST with $|N|$ nodes:

Algorithm 3 Split States Based on AST Structure

```
procedure SplitByTransitions(class)
    splits ← new Map()
    for sn ∈ class do
        signature ← ComputeSignature(sn)
        if splits.containsKey(signature) then
            splits[signature].add(sn)
        else
            splits[signature] ← {sn}
        end if
    end for
    return splits.values
end procedure
procedure ComputeSignature(sn)
    sig ← ∅
    for σ ∈ Σ do
        nextSN ← sn.transitions[σ]
        sig ← sig ∪ {(σ, nextSN.equivalence_class)}
    end for
    for child ∈ sn.ast_children do
        sig ← sig ∪ {('AST', child.type)}
    end for
    return Hash(sig)
end procedure
```

Algorithm 4 Combined AST-Automaton Minimization

```
procedure MinimizeAutomaton(A, T)
    (stateNodes, nodeMap) ← InitializeStructure(A, T)
    classes ← BuildEquivClasses(stateNodes)
    minimizedStates ← ∅
    for class ∈ classes do
        representative ← SelectRepresentative(class)
        minimizedStates ← minimizedStates ∪ {representative}
    end for
    minimizedAST ← BuildMinimizedAST(minimizedStates)
    return (minimizedStates, minimizedAST)
end procedure
procedure BuildMinimizedAST(minimizedStates)
    root ← new ASTNode()
    for state ∈ minimizedStates do
        node ← CreateASTNode(state)
        for σ ∈ Σ do
            if state.transitions[σ] ∈ minimizedStates then
                AddTransitionToAST(node, σ, state.transitions[σ])
            end if
        end for
        AddNodeToAST(root, node)
    end for
    return root
end procedure
```

- Initialization: $O(|Q| \log |N|)$
- Building equivalence classes: $O(|Q|^2|\Sigma|)$
- AST optimization: $O(|Q| \log |N|)$

Total worst-case complexity: $O(|Q|^2|\Sigma| + |Q| \log |N|)$

3.2 Space Complexity

$$Space(Q, N) = O(|Q| + |N| + |Q||\Sigma|)$$

4 Key Properties

Theorem 1 (Correctness)**.** *The minimized automaton accepts the same language as the original automaton.*

Proof. For any word $w \in \Sigma^*$:

1. States in the same equivalence class are indistinguishable
2. The AST structure preserves all transition paths
3. Therefore, acceptance of w is preserved

□

Theorem 2 (AST Consistency)**.** *The minimized AST preserves all valid transition paths of the original automaton.*

Proof. By construction, each transition in the minimized automaton corresponds to a valid path in the minimized AST. □

5 Implementation Notes

5.1 Data Structure Optimizations

- Use hash tables for $O(1)$ state lookups
- Implement lazy AST node creation
- Cache transition computations
- Use bit vectors for set operations

5.2 Memory Management

- Pre-allocate node pools for common operations
- Implement reference counting for AST nodes
- Use flyweight pattern for shared state data
- Employ memory-mapped structures for large automata