Tyler Whitmarsh

Jeremy Pearson

CS 454

Spring 2023

Final Report

DFA Minimization

Problem Statement:

Implement an algorithm to minimize a DFA. The standard algorithm that runs in O(m n2) time can be improved.

Solution Outline:

Remove Unreachable:

This version of DFA minimization just removes all the unreachable states of the DFA. This is useful since by removing these unreachable states it simplifies the DFA without affecting the language that the DFA accepts. We first find all reachable states and then go through and cull the unreachable states.

This algorithm works by two main steps. The first step is finding the reachable states. This is accomplished by creating a set called reachable and a stack called visited that you push the start state onto. Then pop the top of the stack and call it current. Add this state to reachable if it was not already there. Then for each state that current can transition to add those states to the visited stack. Then repeat this process until the visited stack is empty. Next is the second step which is to generate the new DFA only using the states that are present in the reachable set.

While this is not a true minimization algorithm since it does not combine equivalent states it is a good preprocessing in order to clean up unused states. In our implementation of the next algorithm we first remove the unreachable states before running the Hopcroft algorithm.

With an average runtime of O(n + m),where n is the number of states and m is the number of transitions, this algorithm is an efficient way to clean up a DFA before further processing.

Remove Non Distinguishable States Using Hopcroft Algorithm:

This version of DFA minimization involves grouping the states of the DFA into equivalence classes. The resulting DFA will have states equal to the number of equivalence classes obtained from Hopcroft's algorithm. Hopcroft's algorithm starts by initializing two sets P and W. P represents the equivalence classes while W is our worklist that contains sets of states. Initially P is divided into two sets, the set of accepting states and the set of non accepted states. We initialize W to be a copy of P. Then while W is not empty, we take a set S from W, and for each symbol in the DFA's alphabet, we find the set of states that can be reached from S that symbol. This set is called X. Now, For each set Y in P such that the intersection of X and Y, and the difference between Y and X is not empty, replace Y with these two sets. If Y is in W replace it with the two new sets as well. Otherwise, add the smaller of the two new sets. The idea of the algorithm is to repeat this partition refinement until our working set can no longer be refined, giving us the final equivalence classes for this DFA.

Once we have our equivalence classes, we can construct a new DFA using the equivalence classes as our states. The delta function would be determined by following the transitions of the original DFA and finding the corresponding equivalence class for each transition. The starting state would be the equivalence class containing the original start state. The accepting state would be any equivalence class that contains accepting states. The time complexity of this algorithm is in the worst case scenario is O(ns log n) where n is the number of states on the input DFA and s is the size of the alphabet.

Data Structures and Algorithms:

Remove Unreachable:

Our implementation of the remove unreachable algorithm used two data structures

Set: The remove unreachable algorithm uses a set to store which states are reachable. The operation add is used to add new states that have been reached. This operation has a time complexity of O(1). The reachable set is also copied onto the new_dfa states attribute which has a time complexity of O(n). Finally the intersection of the reachable set is used in order to find the final states to be added to the new dfa. This has a time complexity of O(min(len(s), len(t))) where s is the first set and t is the second set.

List: The remove unreachable algorithm uses a list in order to represent the visited stack since python does not have a stack data structure. The two operations that this visited list uses are pop() and append(). It uses pop to get to the top element of the list to check if it is in the reachable set or not. The time complexity of this operation is O(1). The append operation is used to add states to the queue that are transitions of currently reachable states. The time complexity of this operation is O(1).

Hopcroft's Algorithm:

Our implementation of Hopcroft's algorithm uses only one data structure which is a set.

Set: Hopcroft's algorithm involves the use of two main set operations, intersection and difference. The time complexity of set intersection in Python is O(n) n being the smaller of the two sets being compared, while the set difference of X and Y is O(Y). The intersection of two sets refers to the set of elements that are common to both sets. In the context of Hopcroft's algorithm, it is used to identify states that share common transitions and potentially exhibit similar language recognition behavior. By taking the intersection of X and Y, we can see all states in Y that have transitions to the states in X and thus might belong to the same equivalence class. The difference between two sets refers to the set of elements in the first but not in the second. In the context of Hopcroft's algorithm, it is used to identify states in one set that have transitions to certain states but not others. By taking the difference of X and Y, the algorithm identifies the subset of states in Y that have transitions to states outside of X and suggest they might not belong to the same equivalence class.

Time Complexity Tests:

Input Test 1: DFA1

- Unreachable States Runtime: 2.9300000000002936e-08 seconds
- Non Distinguishable Runtime: 2.8599999999996683e-08 secondss
- Non Distinguishable(no pre-processing): 2.8600000000000154e-08 seconds

Input Test 2: DFA2

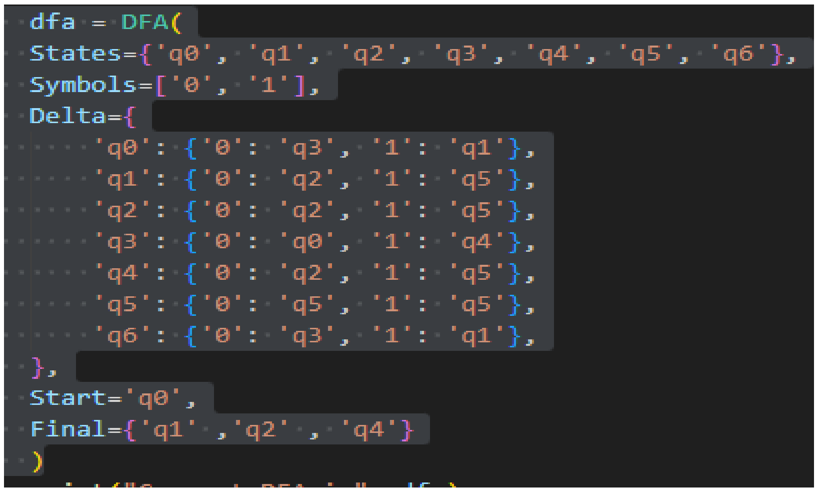
- Unreachable States Runtime: 2.8600000000000154e-08 seconds
- Non Distinguishable Runtime:2.8300000000001934e-08 seconds
- Non Distinguishable(no pre-processing): 3.849999999999687e-08 seconds

Input Test 3: DFA3

- Unreachable States Runtime: 2.860000000000362e-08 seconds
- Non Distinguishable Runtime: 2.860000000000362e-08 seconds
- Non Distinguishable(no pre-processing): 2.9000000000001246e-08 seconds

Sample Input/Output

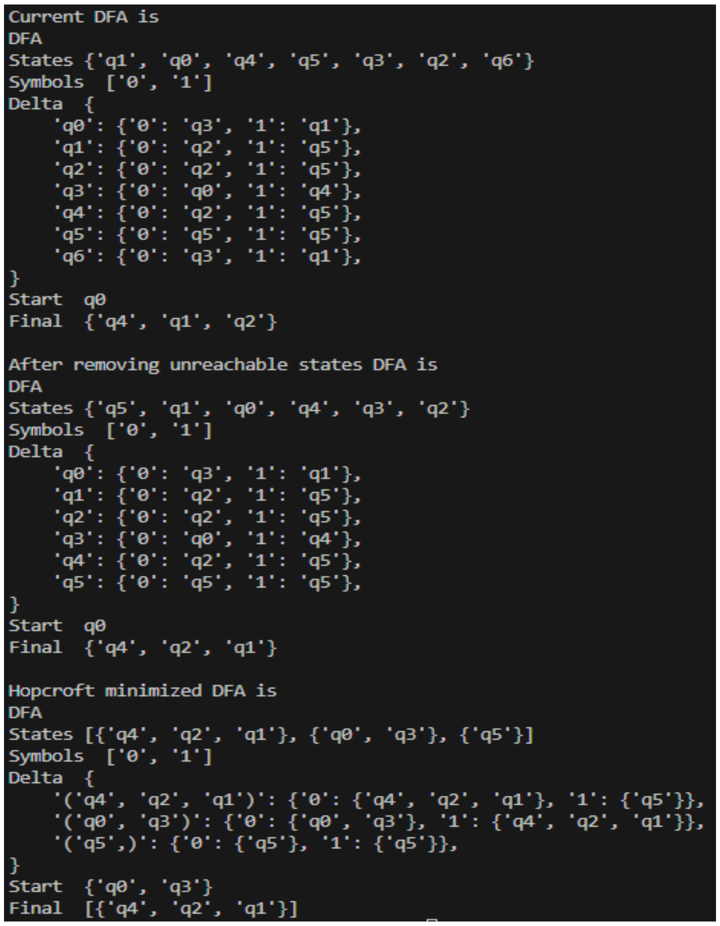
Input: DFA1



```
dfa = DFA(
States={'q0', 'q1', 'q2', 'q3', 'q4', 'q5', 'q6'},
Symbols=['0', '1'],
Delta={
     'q0': {'0': 'q3', '1': 'q1'},
     'q1': {'0': 'q2', '1': 'q5'},
     'q2': {'0': 'q2', '1': 'q5'},
     'q3': {'0': 'q0', '1': 'q4'},
     'q4': {'0': 'q2', '1': 'q5'},
     'q5': {'0': 'q5', '1': 'q5'},
     'q6': {'0': 'q3', '1': 'q1'},
},
Start='q0',
Final={'q1' ,'q2' , 'q4'}
)
```

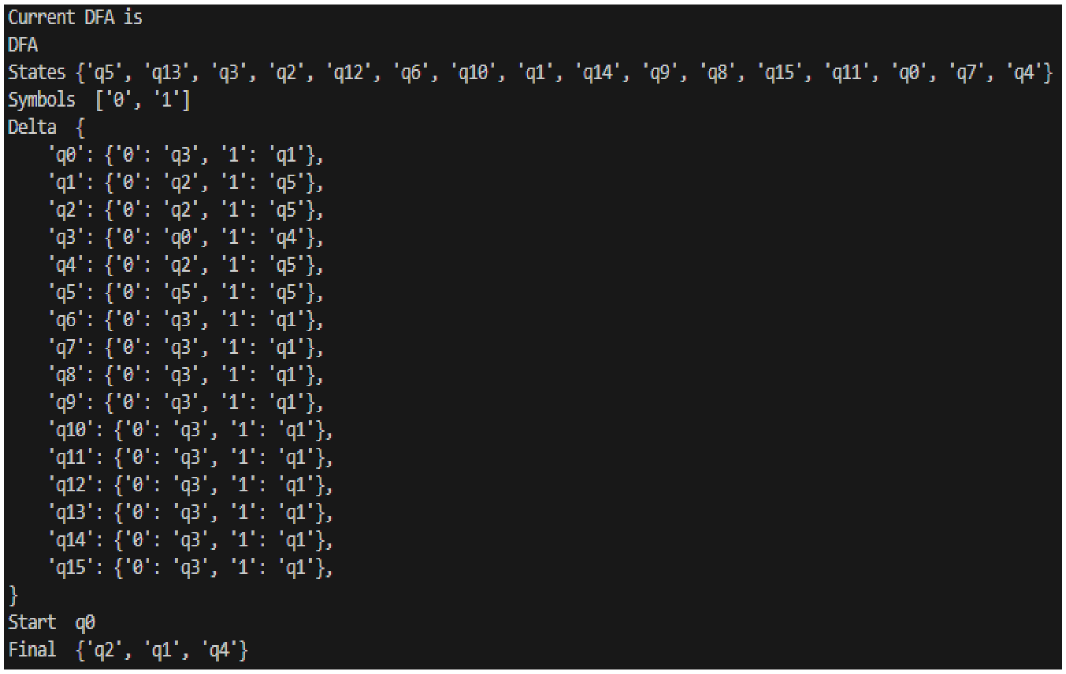
Output:

:



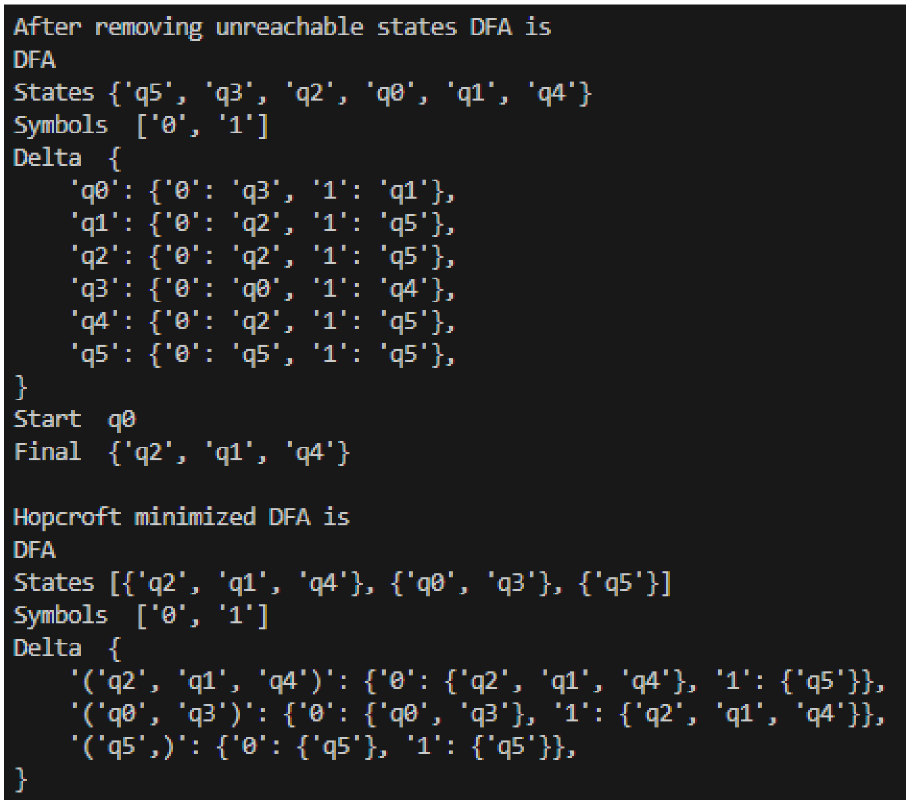
```
Current DFA is
DFA
States {'q1', 'q0', 'q4', 'q5', 'q3', 'q2', 'q6'}
Symbols  ['0', '1']
Delta  {
    'q0': {'0': 'q3', '1': 'q1'},
    'q1': {'0': 'q2', '1': 'q5'},
    'q2': {'0': 'q2', '1': 'q5'},
    'q3': {'0': 'q0', '1': 'q4'},
    'q4': {'0': 'q2', '1': 'q5'},
    'q5': {'0': 'q5', '1': 'q5'},
    'q6': {'0': 'q3', '1': 'q1'},
}
Start  q0
Final  {'q4', 'q1', 'q2'}

After removing unreachable states DFA is
DFA
States {'q5', 'q1', 'q0', 'q4', 'q3', 'q2'}
Symbols  ['0', '1']
Delta  {
    'q0': {'0': 'q3', '1': 'q1'},
    'q1': {'0': 'q2', '1': 'q5'},
    'q2': {'0': 'q2', '1': 'q5'},
    'q3': {'0': 'q0', '1': 'q4'},
    'q4': {'0': 'q2', '1': 'q5'},
    'q5': {'0': 'q5', '1': 'q5'},
}
Start  q0
Final  {'q4', 'q2', 'q1'}

Hopcroft minimized DFA is
DFA
States [{'q4', 'q2', 'q1'}, {'q0', 'q3'}, {'q5'}]
Symbols  ['0', '1']
Delta  {
    '('q4', 'q2', 'q1')': {'0': {'q4', 'q2', 'q1'}, '1': {'q5'}},
    '('q0', 'q3')': {'0': {'q0', 'q3'}, '1': {'q4', 'q2', 'q1'}},
    '('q5',)': {'0': {'q5'}, '1': {'q5'}},
}
Start  {'q0', 'q3'}
Final  [{'q4', 'q2', 'q1'}]
```

Input: DFA2



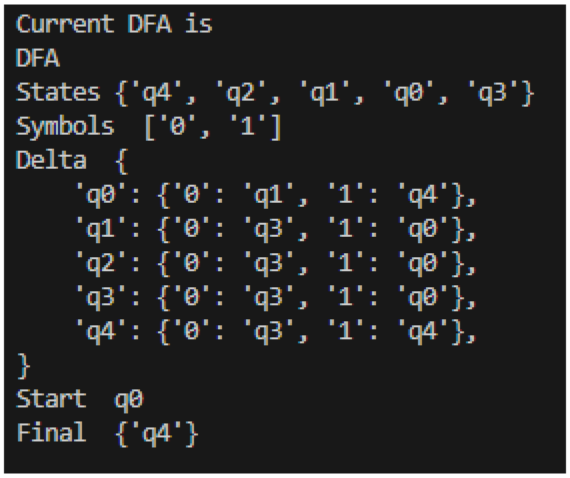
```
Current DFA is
DFA
States {'q5', 'q13', 'q3', 'q2', 'q12', 'q6', 'q10', 'q1', 'q14', 'q9', 'q8', 'q15', 'q11', 'q0', 'q7', 'q4'}
Symbols  ['0', '1']
Delta  {
    'q0': {'0': 'q3', '1': 'q1'},
    'q1': {'0': 'q2', '1': 'q5'},
    'q2': {'0': 'q2', '1': 'q5'},
    'q3': {'0': 'q0', '1': 'q4'},
    'q4': {'0': 'q2', '1': 'q5'},
    'q5': {'0': 'q5', '1': 'q5'},
    'q6': {'0': 'q3', '1': 'q1'},
    'q7': {'0': 'q3', '1': 'q1'},
    'q8': {'0': 'q3', '1': 'q1'},
    'q9': {'0': 'q3', '1': 'q1'},
    'q10': {'0': 'q3', '1': 'q1'},
    'q11': {'0': 'q3', '1': 'q1'},
    'q12': {'0': 'q3', '1': 'q1'},
    'q13': {'0': 'q3', '1': 'q1'},
    'q14': {'0': 'q3', '1': 'q1'},
    'q15': {'0': 'q3', '1': 'q1'},
}
Start  q0
Final  {'q2', 'q1', 'q4'}
```

Output:



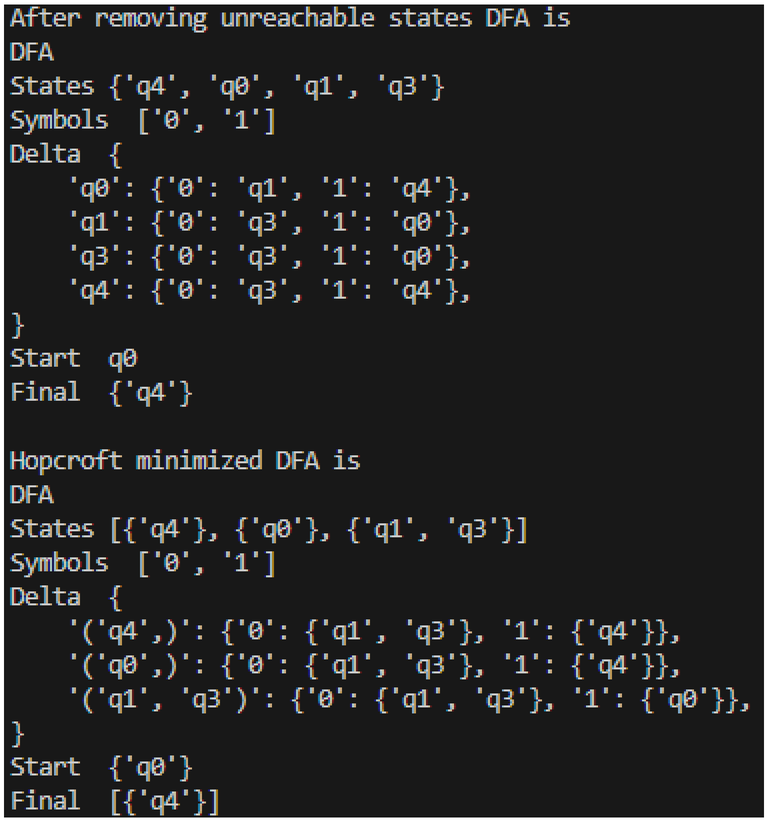
```
After removing unreachable states DFA is
DFA
States {'q5', 'q3', 'q2', 'q0', 'q1', 'q4'}
Symbols  ['0', '1']
Delta  {
    'q0': {'0': 'q3', '1': 'q1'},
    'q1': {'0': 'q2', '1': 'q5'},
    'q2': {'0': 'q2', '1': 'q5'},
    'q3': {'0': 'q0', '1': 'q4'},
    'q4': {'0': 'q2', '1': 'q5'},
    'q5': {'0': 'q5', '1': 'q5'},
}
Start  q0
Final  {'q2', 'q1', 'q4'}

Hopcroft minimized DFA is
DFA
States [{'q2', 'q1', 'q4'}, {'q0', 'q3'}, {'q5'}]
Symbols  ['0', '1']
Delta  {
    '('q2', 'q1', 'q4')': {'0': {'q2', 'q1', 'q4'}, '1': {'q5'}},
    '('q0', 'q3')': {'0': {'q0', 'q3'}, '1': {'q2', 'q1', 'q4'}},
    '('q5',)': {'0': {'q5'}, '1': {'q5'}},
}
```

Input: DFA3



```
Current DFA is
DFA
States {'q4', 'q2', 'q1', 'q0', 'q3'}
Symbols  ['0', '1']
Delta  {
    'q0': {'0': 'q1', '1': 'q4'},
    'q1': {'0': 'q3', '1': 'q0'},
    'q2': {'0': 'q3', '1': 'q0'},
    'q3': {'0': 'q3', '1': 'q0'},
    'q4': {'0': 'q3', '1': 'q4'},
}
Start  q0
Final  {'q4'}
```

Output:



```
After removing unreachable states DFA is
DFA
States {'q4', 'q0', 'q1', 'q3'}
Symbols  ['0', '1']
Delta  {
    'q0': {'0': 'q1', '1': 'q4'},
    'q1': {'0': 'q3', '1': 'q0'},
    'q3': {'0': 'q3', '1': 'q0'},
    'q4': {'0': 'q3', '1': 'q4'},
}
Start  q0
Final  {'q4'}

Hopcroft minimized DFA is
DFA
States [{'q4'}, {'q0'}, {'q1', 'q3'}]
Symbols  ['0', '1']
Delta  {
    '('q4',)': {'0': {'q1', 'q3'}, '1': {'q4'}},
    '('q0',)': {'0': {'q1', 'q3'}, '1': {'q4'}},
    '('q1', 'q3')': {'0': {'q1', 'q3'}, '1': {'q0'}},
}
Start  {'q0'}
Final  [{'q4'}]
```

Conclusion:

Overall this gave us a much better understanding of DFA minimization. The initial algorithm of removing unreachable states was very straightforward. It was a good refresher on how to implement a breadth first search and gave us some insight on what to expect from the Hopcroft algorithm implementation. While implementing Hopcroft's algorithm, seeing how we create groups of states based on their equivalence class really showed why and when we should group states together. Surprisingly, one of the most challenging hurdles we had to overcome was figuring out how to reconstruct the resulting DFA once we had the partitioned states, but once we figured this out, the rest of the project came together very well. The thing we were most surprised by was the fact that preprocessing the dfa to remove unreachable states only gave a slight bump in performance. We did see a slight increase when we added a bunch of unreachable states in dfa2 but it was only minor.This could most likely be due to the fact that the dfa's we were working with were only a few states. We would have to imagine that a dfa with a couple thousand states would benefit greatly from the preprocessing of removing the unreachable states first.

Sources:

Wikipedia:

https://en.wikipedia.org/wiki/DFA_minimization

GeeksForGeeks:

https://www.geeksforgeeks.org/minimization-of-dfa/#

Python time complexity wiki

https://wiki.python.org/moin/TimeComplexity