Implementation of Hopcroft's Algorithm

Hang Zhou

19 December 2009

Abstract

Minimization of a *deterministic finite automaton*(DFA) is a well-studied problem of formal language. An efficient algorithm for this problem is proposed by Hopcroft in 1970 and we have already learned this algorithm in the course. The goal of this writing is to discuss how this algorithm is carried out in time complexity $O(MNlog_2N)$(where M is the size of alphabet and N is the number of states). The writing is in three sections: first, we review the original algorithm; second, we provide corresponding data structures for carrying out the algorithm efficiently; and at last we prove that the algorithm is indeed in time $O(MNlog_2N)$ using these data structures.

1 Introduction:

Let $\mathcal{A}$ be a *deterministic finite automaton* (Q, A, E, I, F), where Q is the finite set of states, A the alphabet, E the set of transition edges, I and F the sets of initial and final states respectively. Let M be the size of alphabet and N be the number of states. For convenience, let A be the set of integers from 1 to M.

The problem of *minimization* is to find an automaton $\mathcal{A}'$ with the minimum number of states which is equivalent to $\mathcal{A}$.

Definition 1. Let $\mathcal{P}$ be a *partition* of Q. The number of classes in $\mathcal{P}$ is bounded by N. Hence every class of $\mathcal{P}$ can be entitled with a unique *name* between 1 and N.

From the definition, for every pair (C, a), where C is a class and a is a letter, it corresponds to a unique element in $\{1, .., N\} \times \{1, .., M\}$.

Definition 2. Let $\mathcal{A}$ be a deterministic finite automaton (Q, A, E, I, F). Let $a \in A$ and $B, C \subseteq Q$. A pair (C, a) is said to *split* B, if $B{\cdot}a \nsubseteq C$ and $B{\cdot}a \cap C \neq \emptyset$ with $B \cdot a = \{x \cdot a | x \in B\}$. In this case, B is split by (C, a) into two parts: $B' = \{x \in B | x \cdot a \in C\}$ and $B'' = \{x \in B | x \cdot a \notin C\}$.

Recall Hopcroft's algorithm in our course:

```
HOPCROFT'S-MINIMIZATION-1ST-VERSION()
 1  P ← {F, F^c}
 2  S ← ∅
 3  for a ∈ A
 4      do INSERT (min(F, F^c), a) to S
 5  while S ≠ ∅
 6      do DELETE (C, a) from S
 7          DECIDE subset T of P with element split by (C, a)
 8          for each B ∈ T
 9              do B', B'' ← SPLIT(B, C, a)
10                  REPLACE B by B' and B'' in P
11                  for b ∈ A
12                      do if (B, b) ∈ S
13                            then REPLACE (B, b) by (B', b) and (B'', b) in S
14                            else INSERT (min(B', B''), b) to S
```

2 Data structure

In this section we provide several data structures for carrying out Hopcroft's algorithm efficiently.

Let INVERSE $= a^{-1}C$, which equals to $\{x | a \cdot x \in C\}$. We will show in the next section that the execution time of the *while* loop is $O(|\text{INVERSE}|)$.

In order to achieve the effect of $\mathcal{T}$, we start by calculating the approximate subset $\mathcal{T}'$ of $\mathcal{P}$, which contains element B such that $B \bigcap \text{INVERSE} \neq \emptyset$. It is easier to obtain $\mathcal{T}'$ after scanning all elements in C. It is in $O(1)$ time to check whether an element in $\mathcal{T}'$ is also in $\mathcal{T}$. So the previous algorithm can be rewritten as following:

```
HOPCROFT'S-MINIMIZATION-2ND-VERSION()
 1  P ← {F, F^c}
 2  S ← ∅
 3  for a ∈ A
 4      do INSERT (min(F, F^c), a) to S
 5  while S ≠ ∅
 6      do DELETE (C, a) from S
 7          INVERSE ← a^{-1}C
 8          DECIDE subset T' of P with element B such that B ⋂ INVERSE ≠ ∅
 9          for each B ∈ T'
10              do if B · a ⊈ C
11                    then B', B'' ← SPLIT(B, C, a)
12                         REPLACE B by B' and B'' in P
```

13 **for** $b \in A$
14 **do if** $(B, b) \in S$
15 **then** REPLACE (B, b) by (B', b) and (B'', b) in S
16 **else** INSERT $(min(B', B''), b)$ to S

Here a partition $\mathcal{P}$ will be characterized by four arrays `Class`, `Part`, `Card`, `Place`:
• `Class[x]` : the name of the class which contains state x;
• `Part [i]` : a pointer to a doubly linked list with all states in class named i;
• `Card [i]` : the size of class named i;
• `Place[x]` : a pointer to the position of state x in the linked list `Part[p]`, where `p=Class[x]`.
The structures above enable us to move a state from one class to another in $O(1)$ time.

An integer variable `Counter` maintains the number of classes in $\mathcal{P}$. It is originally set to 2 (or 1, if $F = Q$), and increases during the execution of the program.

In order to calculate INVERSE $= a^{-1}C$, we use an array `Inv` indexed by $\{1, ..., N\} \times \{1, ..., M\}$ such that `Inv`$[x, a]$ is a pointer to the linked list of state y satisfying $y \cdot a = x$. In this way, INVERSE need not be constructed *physically*, but only be *read* in time proportional to its size when we traverse the list `Part[Class[C]]` and, for each state y in this list, traverse `Inv`$[y, a]$. So the expression "for all $x \in$ INVERSE" is carried out as : "for all $y \in C$ and all $x \in$ `Inv`$[y, a]$".

The list S should efficiently support these operations: *insert*, *search*, and *delete* a "random" element. Noting that the size of S is bounded by MN, we will use a linked list, together with a boolean array `InList`, which is indexed by $\{1, ..., N\} \times \{1, ..., M\}$(`InList[C,a]` is true if $(C, a) \in S$). In this way, *insert* operation is done in $O(1)$: insert this element into linked list and update its corresponding value in `InList`; *search* operation is done in $O(1)$ thanks to `InList`; *delete* operation is done in O(1): delete the head element from the linked list and update its corresponding value in `InList`; at last, to test whether $S \neq \emptyset$ is also done in $O(1)$ because of the linked list structure.

In order to obtain $\mathcal{T}'$, we use the following three date structures:
• `Involved` : a linked list of the names of classes in $\mathcal{T}'$;
• `Size` : an integer array, such that for each class B with name i, `Size`$[i]$ is the cardinality of $B \bigcap a^{-1}C$. This array should be cleared to 0 every time the *while* loop is executed;
• `Twin` : an integer array, such that `Twin`$[i]$ is the name of the new class created while splitting the class named i. This array should also be initialized every time the *while* loop is executed.

3 Analysis of time complexity

We now analyze the time complexity of Hopcroft's algorithm of the 2^{nd} version above. Here we are only concerned with the time complexity, while not caring about the space complexity.

Lemma 3. *The number of classes created during the execution is bounded by* $2N-1$. *In addition, the number of iterations of the while loop is bounded by* $2MN$.

Proof. Consider a binary tree which corresponds to the split operations such that each node is a subset of Q. The root of the tree is the whole set Q. It has two children, and their union equals to Q. Each non-leaf node P in this tree has two children P' and P'', which is the result of a split operation applied to P. Such a binary tree has at most N leaves, so at most $2N-1$ nodes, hence we prove the first half of the lemma. In order to bound the number of iterations of the *while* loop, it suffices to prove that the number of pairs (C,a) inserted to S is bounded by $2MN$, because in each iteration we delete a pair from S. Consider some pair (C,a) inserted to S. Here C is an element of current partition and corresponds to some node in the binary tree above, so the choice of C has at most $2N-1$ different possibilities; and because the choice of a has M different possibilities, we conclude that the number of pairs (C,a) is bounded by $2MN$. □

The initialization before the *while* loop is in time $O(MN)$. The number of iterations of the *while* loop is bounded by $2MN$ (by Lemma 3), so the global execution time of line 6 is $O(MN)$. And the number of classes created is bounded by $2N-1$ (also by Lemma 3), so the global execution time of line 13 to line 16 is $O(MN)$.

It only rests to evaluate the global execution time of line 7 to line 12. In order to do this we take two steps: first, evaluate the sum of |INVERSE| in all execution of the *while* loop (this sum is noted as $\sum$|INVERSE| from now on), and then prove that the global execution time from line 7 to line 12 is O($\sum$|INVERSE|).

Proposition 4. *Let* $a \in A, p \in Q$ *being fixed. The number of times that we* DELETE (C,a) *from list* S, *satisfying* $p \in C$, *is bounded by* $log_2 N$.

Proof. We call a pair (C,a) to be *marked* if $p \in C$. Before the *while* loop, S contains at most one marked pair (this condition holds for all instances of a and p). Let (C,a) be the first marked pair which is deleted from S. If there exists a second marked pair (C'',a), then before DELETE(C'',a), there must be some pair (C',a) which is generated by k REPLACE operations starting from pair (C,a) $(k \geq 0)$, such that (C'',a) is a result of SPLIT (C',a). So $Card(C') \leq Card(C)$ and $Card(C'') \leq Card(C')/2$. Now we take (C'',a) as (C,a) and repeat in

the same way until no such DELETE operation occurs. So the number of such DELETE operations is not more than $log_2 n$ for a fixed pair of p and a. □

Corollary 5. *The sum of* |INVERSE| *in all execution of the* while *loop is bounded by* $MNlog_2N$.

Proof. Let (x, a, y) be a fixed triplet satisfying $y = x \cdot a$. The number of times that x is *read* in the list INVERSE satisfying $y = x \cdot a$ is exactly the same as the number of pairs (C, a) deleted from the list S, such that $y \in C$. From Proposition 4, this number is bounded by $log_2 N$, hence we prove the corollary. □

In order to show that the global execution time of line 7 to line 12 is $O(\sum |\text{INVERSE}|)$, we carry out this part of algorithm in the following way, by traversing the linked list INVERSE twice:

```
HOPCROFT'S-MINIMIZATION-2ND-VERSION-BOTTLENECK()
 1   create an empty list Involved
 2  for  all y ∈ C and all x ∈ Inv[y, a]
 3      do i ← Class[x]
 4         if Size[i] = 0
 5           then Size[i] ← 1
 6                insert i to Involved
 7           else Size[i] ← Size[i] + 1
 8  for  all y ∈ C and all x ∈ Inv[y, a]
 9      do i ← Class[x]
10         if Size[i] < Card[i]
11           then if Twin[i] = 0
12                  then Counter ← Counter + 1
13                       Twin[i] ← Counter
14                       delete x from class i and insert x into class Twin[i]
15
16  for  all j ∈ Involved
17      do Size[j] ← 0
18         Twin[j] ← 0
```

From above, it is clear that the execution time of this part in a single *while* loop is $O(|\text{INVERSE}|)$, so the global execution time of this part is $O(\sum |\text{INVERSE}|)$.

Theorem 6. *Let A be an alphabet of M letters and $\mathcal{A}$ be a DFA on this alphabet with N states, then Hopcroft's algorithm using the data structures in the previous section is in time $O(MNlog_2N)$ in the worst case.*

Remark: This complexity is also tight, which is proved in [2] in detail.

References

[1] D. Beauquier, J. Berstel, and P. Chrétienne. *Éléments d'algorithmique.* Masson, 1992.

[2] J. Berstel and O. Carton. *On the complexity of hopcroft's state minimization algorithm.* CIAA, 2004.

[3] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.