Minimización de AFD'S

Eduardo Salas
Jorge Nicho
Alvaro Aguirre
Teoría de la computación
UTEC

July 21, 2020

Introducción

Un autómata finito determinista permite evaluar una entrada de números o caracteres llamada cadena a través de estados que permiten comprobar si la cadena es válida o no. Todos los autómatas tienen un estado inicial, donde empieza la evaluación, y un estado final donde la cadena es aceptada. Los AFD son útiles para diseñar software y supervisar el comportamiento de circuitos digitales o para escanear largos textos y encontrar palabras o frases específicas dentro de ellos.

Definición del problema

Puede existir un AFD que tenga estados equivalentes. Esto quiere decir que es posible juntarlos y crear un AFD con menos estados. En este trabajo explicaremos dos maneras para poder minimizar un AFD. Es ideal que esté minimizado, ya que se reduce el tiempo de procesamiento de las cadenas recibidas, brindando mayor eficiencia.

1 Estado del Arte

Dado un AFD M = (Q,$\sum$,δ , q_0, F), se trata de encontrar un AFD M' con $L(M) = L(M')$ y tal que M' tenga el mínimo número de estados posible. El método consiste en encontrar todos los estados que son equivalentes, es decir, que son indistinguibles en el autómata. Por cada clase de estados equivalentes, el autómata mínimo necesitará un solo estado. Hemos hallado 3 algoritmos para lograr este objetivo: Hopcroft, Moore y Myhill Nerode.

1.1 Algoritmo de Moore

El método para minimizar un autómata mediante el algoritmo de moore consiste en encontrar todos los estados que son indistinguibles entre sí y sustituirlos por un único estado. Para ello lo principal es averiguar qué estados lo son y cuáles no. El tiempo de ejecución de este algoritmo en el peor caso es $O(n^2 s)$, donde s es el número de sustituciones que se realizan en cada paso y n es el número de estados.
El método para saber qué estados son indistinguibles es el siguiente:

1. Estados separados en grupos que tienen la misma salida inmediata para la misma entrada.
2. Distinguir estados cuyos siguiente estado (s) están en diferentes grupos.
3. Reagrupar los estados y repetir el paso anterior hasta que no hay más estados son distinguibles.

1.1.1 Ejemplo

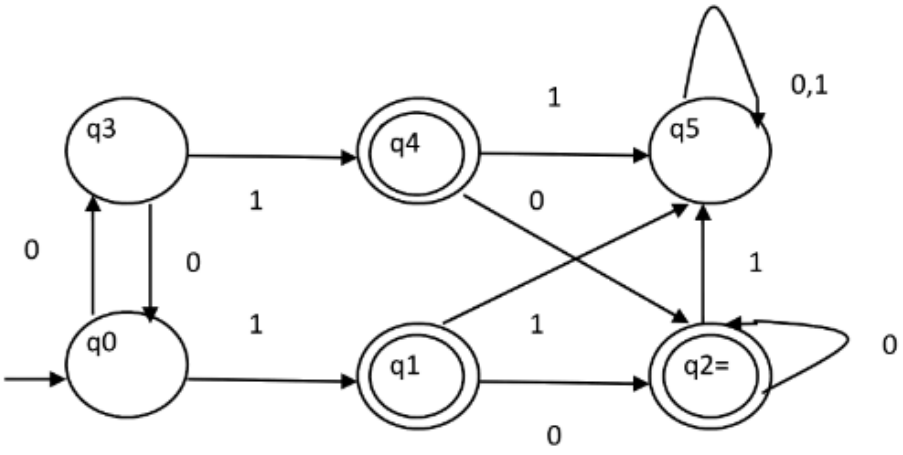

Figure 1: Autómata ejemplo

Paso 1: Separamos los estados de aceptación y los de no aceptación.

$$\mathbb{P}_0 = \{\{q_1, q_2, q_4\}, \{q_0, q_3, q_5\}\}$$

Figure 2: Paso 1

El conjunto de estados lo llamaremos P_o siendo k = 0 ; quiere decir que tienen 0 transiciones iguales, solo están divididos por aceptación y no aceptación.

Paso 2: Separamos aquellos estados en cada conjunto de P_o que tenga 1 estado de transición distinguible.

$$\delta(q_1, 0) = \delta(q_2, 0) = q_2 \ y \ \delta(q_1, 1) = \delta(q_2, 1) = q_5$$

Figure 3: Paso 2

Por lo tanto en el primer conjunto, todos los valores son indistinguibles. Quiere decir que este no será particionado en P_1. Ahora veamos el segundo conjunto:

$$\delta(q_0, 0) = q_3 \ y \ \delta(q_3, 0) = q_0$$
$$\delta(q_0, 1) = q_1 \ y \ \delta(q_3, 1) = q_4$$

Figure 4:

Los movimientos de q_0 con 0 y q_3 con 0 son q_3 y q0 que están en el mismo conjunto de partición en P_o. Lo mismo sucede en q_0 con 1 y q_3 con 1, que son q_1 y q_4 , que también están en la misma partición en P_o. Caso contrario sucede con q_5, veamos:

$$\delta(q_0, 0) = q_3 \ y \ \delta(q_5, 0) = q_5 \ y \ \delta(q_0, 1) = q_1 \ y \ \delta(q_5, 1) = q_5.$$

Figure 5:

Vemos que no pertenecen al mismo conjunto en P_o. Es por ello que P_1 quedaría de la siguiente manera:

$$P_1 = \{\{q_1, q_2, q_4\}, \{q_0, q_3\}, \{q_5\}\}$$

Este es el caso final puesto que no se pueden realizar más particiones. El autómata quedaría de la siguiente manera.

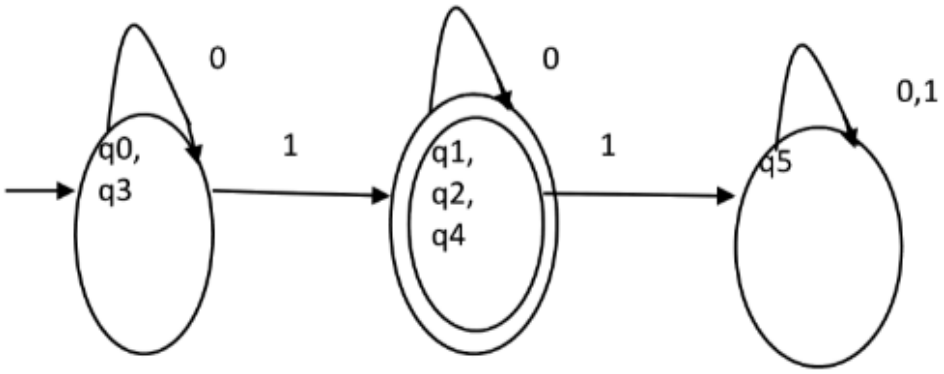

Figure 6: Aútomata final

1.1.2 Pseudocódigo:

```
MooreReductionProcedure(fsm first_fsm){
        fsm new_fsm;
        vector<vector<states>> groups
        vector<states> acceptance;
        vector<states> no_acceptance;
        for i in first_fsm.states
                if(i acceptance)
                        acceptance<=i;
                else
                        no_acceptance<=i;
        end for
        groups<=acceptance and no_acceptance;

        for i in groups.size
                for j in groups[i].size = vec
                        if(distinguishable[j])
                                groups<=distinguishable[j]
                        else
                                continue;
        end for

        append_states(new_fsm,groups)

        return new_fsm;
}
```

1.2 Algoritmo de llenado de tabla

El objetivo es identificar a dos estados distintos A y B que puedan ser reemplazados por un solo estado C que se comporte igual a ambos. Se dice que A y B son estados distinguibles si un input 1 lleva al primer estado a un estado de aceptación y al otro a uno de no aceptación. Otra forma de darse cuenta es ver que un estado sea de aceptación y el otro no. En la Figura 1 podemos aplicar lo explicado anteriormente. El método de llenado de tabla consiste en juntar pares de estados distinguibles y los que sobren serán equivalentes.

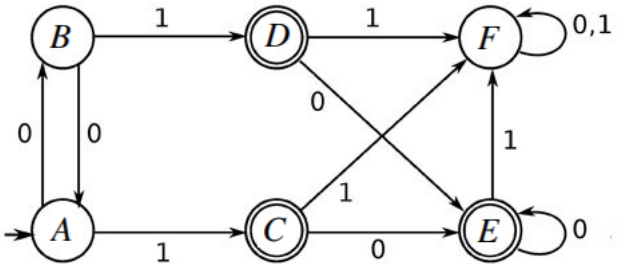

Figure 7: Ejemplo algoritmo de llenado de tabla

Primero nos damos cuenta de que los estados A, B y F son estados de no aceptación mientras que C , D y E sí. Marcamos todos los pares (EF, EA, EB, *etc*). Luego nos damos cuenta de que $(A, 1)$ lleva a un estado de aceptación C mientras que $(F, 1)$ nos lleva a F, un estado de no aceptación. Por lo tanto no son equivalentes y marcamos este par en la tabla. Luego repetimos este análisis con todos los pares aún no marcados. El resultado final se aprecia en ***Figure 8***.

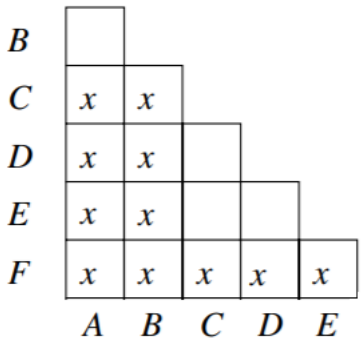

Figure 8: Tabla de pares

Después listamos los pares de estados no marcados: (A,B), (C,D), (C,E), (D,E). Gráficamente se muestra en ***Figure 9***. Finalmente dibujamos el nuevo autómata fusionando los estados equivalentes agrupados en la Figura 3 y el resultado se muestra en ***Figure 10***.

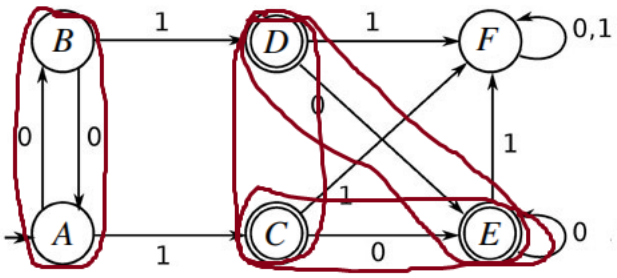

Figure 9: Agrupación visual de los estados equivalentes

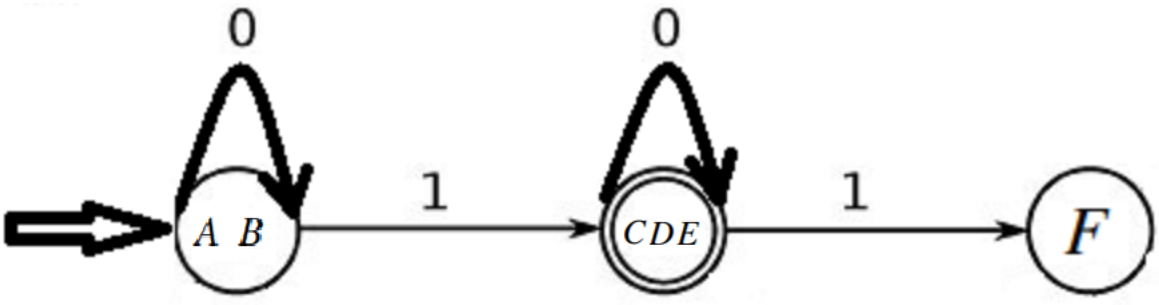

Figure 10: AFD minimizado con el algoritmo de llenado de tabla

Los pasos del algoritmo son:

1. Dar como input un AFD (P, Q), indicando sus transiciones, estado inicial y estado final.

2. Construir una matriz n x n, con una fila para cada estado y una columna para cada estado, donde n es la suma del número de estados en P y el número de estados en Q. Solo necesitamos considerar el área debajo de la diagonal de la matriz.

3. Marcamos con 0 (sin marcar) o 1 (marcado) todos los pares donde PF y Q F, donde Fes el conjunto de estados finales.

4. Si hay pares sin marcar (P, Q) de modo que [δ(P,x),δ(Q,x)] esté marcado, marque [P, Q], donde 'x' es un símbolo de entrada. Repita esto hasta que no se puedan hacer más marcas.

5. Combinamos todos los pares de estados que no han sido marcados y los juntamos en un solo estado correspondiente para formar el AFD minimizado.

6. Creamos los nuevos estados para el AFD minimizado.

7. Relacionamos cada estado para formar el AFD minimizado.

1.2.1 Pseudocódigo:

```
Table_Filling_Algorithm(){
        int n = numberOfNodes;
        int m = numberOfSymbols;

        for i=0 to n
            bool_finalState[i];

        for i=0 to n
            for j=0 to m
                t_initial_transition[i][j];

        for i=0 to n
            for j=0 to m
                t_mark[i][j] = false;

        Distinguishable(bool_finalState, t_initial_transition, t_mark);

        Merge(t_mark, t_initial_transition);

        Print(result);
}

Distinguishable(bool_finalState, t_initial_transition, t_mark){

    for i=o to n
        while j != a
            if (bool_finalState[i]==false and bool_finalState[i]==true) or (bool_finalState[
                i]==true and bool_finalState[i]==false)

    for i=o to n
        while j != a
            if !t_mark[i][j]
                var1 = t_initial_transition[i][k];
                var1 = t_initial_transition[j][k];
                if t_mark[var1][var2]
                    t_mark[i][j] = true;

    return t_mark;
}

Merge(t_mark, t_initial_transition){

    for i = 0 to n
        while j !=a
            if t_mark[i][j]
                if var1==i
                    result.pop_back();
                    combine = to_string(var1) + to_string(var2) + to_string(j);
                    result.push_back(combine);
                else if var2 == j
                    result.pop_back();
                    combine = to_string(var1) + to_string(var2) + to_string(j);
                    result.push_back(combine);
                else
                    combine = to_string(var1) + to_string(var2) + to_string(j);
                    result.push_back(combine);

                var1 = i;
                var2 = j;
                merge=true;
        if(!merge){
            var1=i;
            var2=j;
            result.push_back(combine);
        }
```

```
65
66      return result;
67
68  }
```

Llenar la tabla, analizar si dos estados son distinguibles y combinar los estados que podrían minimizarse lleva un tiempo polinómico. Debido a que se necesita una cantidad de pares igual a $n(n-1)/2$, tomando en cuenta que existen n estados podemos determinar que la complejidad es de al menos de $O(n^2)$ y tiene un límite de $O(n^4)$.

1.3 Algoritmo de Hopcroft

Este algoritmo nos sirve para minimizar el número de estados de un AFD y para determinar si dos AFD's son equivalentes. Su cumplejidad se encuentra en O(nlogn), donde 'n' es el numero de estados.
El algoritmo comienza con una partición. La partición inicial es una separación de los estados en dos subconjuntos de estados que claramente no tienen el mismo comportamiento entre sí: los estados de aceptación y los estados de rechazo. Luego, el algoritmo elige repetidamente un conjunto A de la partición actual y un símbolo de entrada c, Y se divide cada uno de los conjuntos de la partición en dos subconjuntos: el subconjunto de estados que conducen a A en símbolo de entrada c, y el subconjunto de estados que no conducen a A. Desde A ya se sabe que tiene un comportamiento diferente al de los otros conjuntos de la partición, los subconjuntos que conducen a A también tienen un comportamiento diferente de los subconjuntos que no conducen a una. Cuando no se pueden encontrar más divisiones de este tipo, el algoritmo termina.
El propósito de la declaración if (Y is in W) es parchear W, el conjunto de distintivos. Vemos en la declaración anterior, en el algoritmo, que Y acaba de dividirse. Si Y está en W, se ha vuelto obsoleto como un medio para dividir las clases en futuras iteraciones. Entonces, Y debe ser reemplazado por ambas divisiones debido a la observación anterior. Sin embargo, si Y no está en W, solo una de las dos divisiones, no ambas, debe agregarse a W debido al Lema anterior. Elegir la más pequeña de las dos divisiones garantiza que la nueva adición a W no sea más de la mitad del tamaño de Y. Esto provoca un descenso de la complejidad, lo que le permite llegar a O(nlogn) ya que los conjuntos extraídos de Q que contienen el estado objetivo de la transición tienen tamaños que disminuyen entre sí en un factor de dos o más, por lo que cada transición participa en O (log n) de los pasos de división en el algoritmo.
Una vez que el algoritmo de Hopcroft se ha utilizado para agrupar los estados del DFA de entrada en clases de equivalencia, se puede construir el DFA mínimo formando un estado para cada clase de equivalencia.

1.3.1 Pseudocódigo:

```
Hopcroft_Algorithm(){
        P := {F, Q \ F};
        W := {F, Q \ F};
        while (W is not empty) do
            choose and remove a set A from W
            for each c in    do
                let X be the set of states for which a transition on c leads to a state in A
                    for each set Y in P for which X     Y is nonempty and Y \ X is nonempty
                        do
                    replace Y in P by the two sets X     Y and Y \ X
                    if Y is in W
                        replace Y in W by the same two sets
                    else
                        if |X     Y| <= |Y \ X|
                            add X     Y to W
                        else
                            add Y \ X to W
                end;
            end;
        end;
}
```

1.4 Experimentación

	HOPCROFT	MOORE	MYHILL NERODE
	0,016	0,015	0,032
	0,017	0,018	0,023
	0,011	0,019	0,047
	0,016	0,017	0,035
	0,014	0,012	0,037
	0,014	0,011	0,035
	0,013	0,016	0,041
	0,013	0,012	0,041
	0,01	0,012	0,093
	0,014	0,015	0,095
PROMEDIO	0,0138	0,0147	0,0479

Figure 11: Resultados de la experimentación de los tres algoritmos en ms

1.5 Implementación en C++

Link de Github: https://github.com/THEFLILUX/TeoComp_Project/tree/master

1.6 Referencias

[1] John E. Hopcroft, Rajeev Motwani Jeffrey D. Ullman. (1979). Introduction to automata theory, languages, and computation. USA: Addison-Wesley.

[2] Daphne A. Norton (2009). Algorithms for Testing Equivalence of Finite Automata,

with a Grading Tool for JFLAP (Master's thesis). Rochester Institute of Technology, USA.

[3] Wiki site:https://en.wikipedia.org/wiki/DFA_minimization.