Universidad de Ingeniería y Tecnología

Ciencia de la Computación

Minimización AFD's

Proyecto - Teoría de la Computación I

Autores:
Barrueta Aspajo, Frings
Riveros Soto, Andres
Rodríguez Llanos, Renato

Docente:
Gutierrez Alva, Juan

Diciembre 2020

1 Introducción

El diseño de autómatas para poder describir una gramática regular recae el problema de poder diseñar un automáta con estados redundantes, definimos un estado redundante como estado el cual tiene un comportamiento similar a otro ya existente. Entonces, podemos diseñar automatas que describan un solo lenguaje regular de diferentes maneras. El problema de diseñar automatas regulares con estados redundantes es que cuando se trabaja con compiladores es una prioridad que el autómata sea eficiente.

2 Algoritmo de Brzozowski

2.1 Algoritmo de construcción de subconjuntos *(subset construction)*

El algoritmo de construcción de subconjuntos toma como entrada un $AFN(N, \Sigma, \delta_N, n_0, N_A)$ y retorna un $AFD(D, \Sigma, \delta_D, d_0, D_A)$.

Algorithm 1 Subset Construction Algorithm

$q_0 \leftarrow \epsilon - closure(\{q_0\})$
$Q \leftarrow q_0$
$WorkList \leftarrow \{q_0\}$
while WorkList $\neq \phi$ **do**
  remove q from WorkList
  for each character $c \in \Sigma$ **do**
    t $\leftarrow$ ϵ-closure(Delta(q, c))
    $T[q, c] \leftarrow t$;
    if $t \notin Q$ **then**
      add t to Q and to WorkList
    end if
  end for
end while

El algoritmo construye un conjunto Q de q_i elementos cuales q_i son subconjuntos de N. El algoritmo se detiene cuando cada estado corresponde a un estado del nuevo AFD construído. Esta construcción construye elementos de Q a partir de las transiciones del AFD. Es decir, q_i es una representación valida del AFN. Este algoritmo tiene complejidad $O(n^2)$ ya que Q puede ser tan grande como 2^n.

2.2 Algoritmo de Inversión

El algoritmo de inversión es mucho más simple que el algoritmo de construcción de subconjuntos. Este algoritmo únicamente invierte todas las transiciones de los estados del AFD. Y, aumenta un estado que apunta a todos los elementos del conjunto de estados finales con transición ϵ, y se convierte en el estado inicial del AFD inverso. Nos podemos dar cuenta que este algoritmo convierte un AFD en un AFN-ϵ ya que aumenta un estado que transiciona a otros estado mediante transiciones epsilon.

2.3 Brzozowski

Ahora, si aplicamos el algoritmo de construcción del subconjuntos al estado inicial de un AFN que transiciona mediante transiciones ϵ hacia varios estados, el algoritmo tomara todas esas transiciones y las agrupará en una sola en un AFD. El algoritmo de construcción de subconjuntos siempre $AFD's$ que no tienen rutas duplicadas. El algoritmo de Brzozowski utiliza esta propiedad del algoritmo de contrucción de subconjuntos para crear $AFD's$ mínimos a partir de un AFN.

Algorithm 2 Brzozowski Algorithm

$subset(reverse(subset(reverse())))$

La primera aplicación de $subset(reverse())$ elimina los estados duplicados del AFD original. La segunda $subset(reverse(subset(reverse())))$ elimina todos los estado que hayan podido surgir del AFN creado en el primer $reverse$.

2.4 Código

2.4.1 Construcción de subconjuntos

```
Automata Automata::powerset(tuple<int,vector<state_afn>,vector<bool>> afn){
    set<int> initialState;
    initialState.insert(get<0>(afn));
    ↪ // Estado inicial
    set<int> q_0 = clausura(initialState,get<1>(afn));
    vector<set<int>> Q;
    ↪ // Vector de estados
    Q.push_back(q_0);

    map<set<int>,tuple<set<int>,set<int>>> transitions;                    // Mapa de
    ↪ transiciones

    queue<set<int>> workList;
    ↪ // Vector de control de estados
    workList.push(q_0);

    while(!workList.empty()){
        set<int> q = workList.front();
        workList.pop();
        bool flagNull = false;
        tuple< set<int>,set<int>> trans;
        for(int i = 0; i<2; i++){
            set<int> t = clausura(delta(q,i,get<1>(afn)),get<1>(afn));

            if(t.size() == 0){
                flagNull = true;
                continue;
            }
            if(i == 0) get<0>(trans) = t;
            else get<1>(trans) = t;

            auto it = find(Q.begin(),Q.end(),t);
            if(it == Q.end()){
                Q.push_back(t);
                workList.push(t);
            }
        }
        if(!flagNull) transitions[q] = trans;
    }

    // Parseamos el afn
    Automata afdResult;
    afdResult.initialState = 0;
    afdResult.states = vector<state>(Q.size());
    afdResult.stateFinal = vector<bool>(Q.size(),false);
```

```
    for(size_t i = 0; i<Q.size(); ++i){
        tuple<set<int>,set<int>> trans = transitions[Q[i]];
        auto transZero = find(Q.begin(),Q.end(),  get<0>(trans));
        auto transOne = find(Q.begin(),Q.end(),  get<1>(trans));
        afdResult.states[i].adjacentes[0] = transZero - Q.begin();
        afdResult.states[i].adjacentes[1] = transOne - Q.begin();

        // Pasamos los valores finales
        for(size_t j = 0; j<get<2>(afn).size(); ++j){
            if(get<2>(afn)[j] == true){
                auto itFinal = find(Q[i].begin(),Q[i].end(), j);
                if(itFinal != Q[i].end()){
                    afdResult.stateFinal[i] = true;
                }
            }
        }
    }

    return afdResult;
}
```

2.4.2 Inverso

```
tuple<int,vector<state_afn>,vector<bool>> Automata::reverse(){
    //create a new initial state
    int newState = states.size();
    vector<state_afn> afn(newState+1);                                          //
    ↪  Creamos un afn
    vector<bool> finalstatesAfn(newState+1,false);                              //
    ↪  Estado finales del AFN
    //create temp to store the current inititial state
    int oldNewstate = initialState;
    ↪  // Guardamos el valor inicial

    // STEP 1
    // invertimos transiciones
    for(size_t i=0; i<states.size(); i++){
    ↪  // Recorremos cada estado
        for(size_t j=0; j<2; ++j){
        ↪  // Recorremos las transiciones de cada estado
            if(i == states[i].adjacentes[j]){
                afn[i].adjacentes[j].push_back(states[i].adjacentes[j]);
            }
            else{
                int destinationState = states[i].adjacentes[j];
                ↪  // Guardamos el destino
                afn[destinationState].adjacentes[j].push_back(i);
                ↪  // Almacenamos en el destino el valor de salida
            }
        }
    }
```

```
    // STEP 2
    // Agregamos un estado y lo unimos a todos los estados de aceptacion
    for(size_t i = 0; i<stateFinal.size(); i++){
        if(stateFinal[i] == true){
            afn[newState].adjacentes[2].push_back(i);
        }
    }

    // STEP 3
    // Convertimos el antiguo estado inicial en estado final
    finalstatesAfn[oldNewstate] = true;
    return {newState, afn, finalstatesAfn};
}
```

2.4.3 Brzozowski

```
Automata Automata::brzozowski(){
    return powerset((powerset(this->reverse())).reverse());
}
```

3 Algoritmo de Myhill - Nerode

La finalidad del algoritmo de llenado tabla es marcar e identificar a todos los pares de estados distinguibles en una tabla, el resto serán equivalente. Por ejemplo, los pares estados distintos X y Y que puedan ser reemplazados por un solo estado Z que tenga el mismo comportamiento son pares equivalentes.

3.1 Método del algoritmo

1. Dibujar una tabla para todo los pares de estados(P,Q)
2. Marque todos los pares de estados donde P sea un estado de aceptacion y Q no sea un estado de aceptación.
3. Si hay pares sin marcar (P,Q) tal que [δ(P, x),δ(Q, x)] esté marcado, entonces marque [P,Q] donde "X" es un símbolo de entrada. Repita esto hasta que no se pueden hacer más marcas.

3.2 Explicación del algoritmo

El algoritmo retorna un tabla marcada con 1 para los estados distinguibles y con 0 para los pares de estados equivalentes. Esto funciona recorriendo toda los pares de estados de la tabla hasta encontrar un par de estado (R, S) tal que $[\delta(R, x), \delta(S, x)]$ está marcado, por tanto marcamos el par de estado actual (R, S) con el cual hemos llegado a $[\delta(R, x), \delta(S, x)]$ y aumento en 1 el número de cambios realizado en la tabla. Este aumento en 1 se realiza debido a hemos agregado una nueva marca (X, Y) en la tabla. Estos nuevos cambios pueden influenciar a los pares de estados anteriores que aún no han sido marcados. Ejecutar un nuevo recorrido de toda la tabla al agregar nuevos cambios en la tabla puede ocasionar una complejidad de $O(n^4)$.

3.3 Código del algoritmo

```
vector<vector<bool>> Automata::equivalenceAlgorithm(){
    int nStates = states.size();
    vector<vector<bool>> marked(nStates,vector<bool>(nStates));
    for(int i = 0; i < nStates;i++){
        for(int j = 0; j < nStates;j++){
            marked[i][j] = false;
        }
    }
    // mark pairs Qi | F and Qj | F
    for(int i = 0; i < nStates; i++){
        if(stateFinal[i]){
            for(int j = 0; j < nStates;j++){
                if(i == j) continue;
                if(!stateFinal[j]){
                    marked[i][j] = 1;
                    marked[j][i] = 1;
                }
            }
        }
    }

    int changes = 1;
```

```
    while(changes != 0){
        changes = 0;
        for(int i = 0; i < nStates; i++){
            for(int j = 0; j < nStates; j++){
                if(i == j) continue;
                for(int k = 0; k < 2; k++){
                    if(marked[states[i].adjacentes[k]][states[j].adjacentes[k]] &&
                    ↪  (!marked[i][j] && !marked[j][i])){
                        marked[i][j] = 1;
                        marked[j][i] = 1;
                        changes++;
                    }
                }
            }
        }
    }
    return marked;
}
```

4 Optimización del algoritmo de Myhill Nerode

La finalidad del algoritmo es marcar a todos los pares de estados distinguibles en una tabla como en el caso anterior. Con la diferencia que aplicará otra estrategia para obtener el mismo resultado en un menor tiempo de ejecución al llenar la tabla.

4.1 Método del algoritmo

1.Inicializar, para cada par de estados {r,s}, una lista de dichos pares {p,q} que "dependen de" {r,s}. Si se determina que {r,s} es distinguible, entonces {p,q} es distinguible.

2. Inicialmente creamos la lista examinando cada par de estado {p,q}, y para cada entrada símbolo de entrada a, incluimos{p,q} en la lista de los pares de estados {δ(p,a),δ(q,a)}, que son los estados sucesores de p y q para la entrada a.

1. Si se detecta que {r,s} es distinguible, entonces se recorre la lista de {r,s}.
2. Cada par de dicha lista que no esté marcado como distinguible, se marca como tal y se coloca en la cola de pares cuyas listas hay que comprobar de forma similar

4.2 Explicación del algoritmo

1. Dibujar una tabla para todos los pares de estados (P, Q)
2. Marque todos los pares de estados donde P sea un estado de aceptación y Q no sea un estado de aceptación. Y guardar dichos pares en un queue.
3. Aplicar reverse al afd actual invirtiendo los estados y obteniendo estados con múltiples transiciones con el lenguaje {0,1}.
4. Realizar un bfs a partir de los pares de estados distinguibles de la cola. A través de los estados obtenidos por el reverse. Si (P,Q) está marcado y [δ(P, x),δ(Q, x)] no lo está. Marcó los pares de estados [δ(P, x),δ(Q, x)] y se añaden dichos pares a la cola.

4.3 Reverse

```
vector<state_afn> Automata::reverse_2(){
    vector<state_afn> afn(states.size());                                    //
    ↪  Creamos un afn

    for(int i=0; i < states.size(); i++){
    ↪  // Recorremos cada estado
        for(int j=0; j < 2; ++j){
        ↪  // Recorremos las transiciones de cada estado
            if(i == states[i].adjacentes[j]){
```

```
                afn[i].adjacentes[j].push_back(i);
            }
            else{
                int destinationState = states[i].adjacentes[j];
                ↪ // Guardamos el destino
                afn[destinationState].adjacentes[j].push_back(i);
                ↪ // Almacenamos en el destino el valor de salida
            }
        }
    }
    return afn;
}
```

4.4 Código del Algoritmo

```
vector<vector<bool>> Automata::secondPart(){
    int nStates = (int)states.size();
    vector<vector<bool>> marked(nStates,vector<bool>(nStates));
    queue<pair<int,int>> q;
    for(int i = 0; i < nStates;i++){
        for(int j = 0; j < nStates;j++){
            marked[i][j] = false;
        }
    }
    for(int i = 0; i < nStates; i++){
        if(stateFinal[i]){
            for(int j = 0; j < nStates;j++){
                if(i == j) continue;
                if(!stateFinal[j]){
                    marked[i][j] = 1;
                    marked[j][i] = 1;
                    q.push({i,j});
                }
            }
        }
    }
    set<pair<int,int>> s;
    vector<state_afn> afn = reverse_2();

    while(!q.empty()){
        pair<int,int> e = q.front();
        q.pop();
        for(int i = 0; i < 2; i++){
            vector<int> adjA = afn[e.first].adjacentes[i];
            vector<int> adjB = afn[e.second].adjacentes[i];
            if(adjA.size() == 0 || adjB.size() == 0) continue;
            else{
                for(int a : adjA){
                    for(int b : adjB){
```

```
                        if(a == b || marked[a][b] || marked[b][a]) continue;
                        if(s.find({a,b}) == s.end() && s.find({b,a}) == s.end()){
                            s.insert({a,b});
                            q.push({a,b});
                            marked[a][b] = 1;
                            marked[b][a] = 1;
                        }
                    }
                }
            }
        }
    }
    return marked;
}
```

5 Algoritmo de Huffman - Moore

5.1 Pseudocódigo del algoritmo (Hopcroft Cap 4.4.3)

Ahora estamos en condiciones de enunciar sucintamente el algoritmo para minimizar un AFD $A = (Q, \Sigma, \delta, q_0, F)$.

1. Utilizamos el algoritmo de llenado de tabla para determinar todos los pares de estados equivalentes.
2. Dividimos el conjunto de estados Q en bloques de estados mutuamente excluyentes aplicando el método descrito anteriormente.
3. Construimos el AFD equivalente con menor número de estados B utilizando los bloques como sus estados. Sea γ la función de transiciones de B. Supongamos que S es un conjunto de estados equivalentes de A y que a es un símbolo de entrada. Así, tiene que existir un bloque T de estados tal que para todos los estados q pertenecientes a S, $\delta(q,a)$ sea un miembro del bloque T. En el caso de que no sea así, entonces el símbolo de entrada a lleva a los dos estados p y q de S a estados pertenecientes a bloques distintos, y dichos estados serán distinguibles por el Teorema 4.24. Este hecho nos lleva a concluir que p y q no son equivalentes y por tanto no pertenecían a S. En consecuencia, podemos hacer $\gamma(S,a) = T$. Además:
   a) El estado inicial de B está en el bloque que contiene el estado inicial de A.
   b) El conjunto de estados de aceptación de B está en el conjunto de bloques que contienen los estados de aceptación de A. Observe que si un estado de un bloque es un estado de aceptación, entonces todos los restantes estados de dicho bloque serán también estados de aceptación. La razón de ello es que cualquier estado de aceptación es distinguible a partir de cualquier estado de no aceptación, por lo que no podemos tener estados de aceptación y de no aceptación en un bloque de estados equivalentes.

5.2 Explicación del algoritmo

En este algoritmo, el propósito consiste en conseguir un AFD equivalente que acepta el mismo lenguaje pero que contiene menos estados (minimización) que cualquier AFD que acepte el mismo lenguaje. Primero, el programa recibe como input un AFD. Luego, llamamos a la función de Huffman-Moore. Definimos nuestras variables necesarias tal como esta en el código y creamos un vector donde por default estaran todos los estados del AFD. Este vector nos servirá para saber cuales son los estados del AFD minimizado ya que solo pushea un estado por cada bloque de estados equivalentes.
A continuación, utiliza el algoritmo del problema 2 (equivalencia de estados) para saber que estados son equivalentes. Esta información es importante ya que el algoritmo necesita agrupar bloques de los estados equivalentes. Pero, antes de pushear los estados equivalentes verificamos si esta en nuestro vector los estados si esta; lo pusheamos y lo eliminamos para evitar los casos en que se repita, posteriormente, tambien pusheamos los estados equivalentes con ese estado en un vector de vectores y tambien lo eliminamos de nuestro vector para saber que ya fue tomado y si vuelve a aparecer ya no necesitaria ser tomado. Por otro lado, si no esta en nuestro vector no hacemos nada. Este paso lo repetimos hasta recorrer toda la matriz de equivalencia de estados.
Finalmente, necesitamos construir el AFD equivalente con el menor numero de estados a partir de el vector de estados minimos y el vector de vectores de los estados equivalente de cada estado. Para ello, implementamos una función adicional llamada createautomata donde le pasamos los datos mencionados anteriormente y nos devuelve el automata.

- Código del algoritmo

```
// Problema 4
Automata Automata::huffman_moore(){
    // Paso 1
    // Armamos la tabla de estados distinguibles
    vector<vector<bool>> tableDistinct = this->equivalenceAlgorithm();
    vector<vector<int>> temp_equivalence_states;
    vector<int> temp_states;
    vector<int> back_up;
    int k=0;

    //pusheamos todos los estados del input
    for(int i=0; i<states.size();i++){
        temp_states.push_back(i);
    }

    // Paso 2
    // Particion de estados
    for(int i = 0; i < tableDistinct.size(); i++){
        //si encontramos el indice pusehamos y lo borramos para evitar el caso en
        ↪  que se repita
        //si no, no hacemos nada
        if (find(temp_states.begin(), temp_states.end(), i) != temp_states.end()){
            back_up.push_back(i);
            temp_states.erase(remove(temp_states.begin(), temp_states.end(), i),
            ↪  temp_states.end());
            for(int j = tableDistinct.size()-1; j>0+i; j--){
                //verificamos que este en el vector, sino no hacemos nada
                if (find(temp_states.begin(), temp_states.end(), j) !=
                ↪  temp_states.end()){
                    // agregar bloques de estados que son equivalentes entre sí
                    if(tableDistinct[j][i]==0){
                        //redefinimos el tamaño del vector donde estaran los
                        ↪  equivalentes
                        temp_equivalence_states.resize(back_up.size());
                        //pusheamos todos los equivalentes de un indice
                        temp_equivalence_states[k].push_back(j);
                        //eliminamos del vector para evitar el caso en que se
                        ↪  repita
                        temp_states.erase(remove(temp_states.begin(),
                        ↪  temp_states.end(), j),temp_states.end());
                    }
                }
            }
        }
        k++;
    }

    //Paso 3
    //Construimos el AFD equivalente con menor numero de estados
    Automata MinAutomata;
    return create_automata(MinAutomata,back_up,temp_equivalence_states);
```

```
}
```

6 Algoritmo de Hopcroft

El algoritmo de Hopcroft nos permite fusionar estados equivalentes de un *AFD*. Este algoritmo se basa en perfeccionar la partición de estados de un *AFD* en grupos según su comportamiento. Para referirnos a estados con comportamiento similar utilizamos la grupos con clases de equivalencia(*reflexiva*, *transitiva*, *simétrica*).

Algorithm 3 Hopcroft Algorithm

```
1.  P := {F, Q \ F}
2.  W := {F, Q \ F}
3.  while W is not empty do
4.    choose and remove a set A from W
5.    for each c in Σ do
6.      let X be the set of states for which a transition on c leads to a state in A
7.      for each set Y in P for which X ∩ Y is not empty and Y \ X is notempty do replace Y in P by
        the two sets X ∩ Y and Y \ X do
8.        if Y ∈ W then
9.          replace Y in W by the same two sets
10.       else
11.         if |X ∩ Y| ≤ |Y \ X| then
12.           add X ∩ Y to W
13.         else
14.           add X \ Y to W
15.         end if
16.       end if
17.     end for
18.   end for
19. end while
```

La primera parte del algoritmo hace la partición más evidente, manteniendo la relación de equivalencia presentado en el algoritmo de Myhill Nerode. Partimos nuestro conjunto de estados en estados de aceptación y estado de no aceptación. Luego, el algoritmo toma conjuntos de estados A de la partición actual, el conjunto W. Este conjunto W podemos llamarlo conjunto el conjunto de distinguidores ya que nosotros tomaremos los conjuntos de estados de ahí y haremos la particiones. Entonces, del conjunto A y caracter c del alfabeto, creamos dos conjuntos un conjunto donde tenemos estados que con transiciones c llegan a algún estado de A y el otro conjunto su contrapuesto. El algoritmo termina cuando el conjunto de distinguiores W se queda vacío. Ahora para el criterio de las particiones definimos un lemma.

Lemma 1 *Dado un carácter fijo c y una clase de equivalencia Y que se divide en clases de equivalencia B y C, solo uno de B o C es necesario para refinar toda la partición.*

Teniendo esto en cuenta, verificamos que la partición sea valida. Es decir, si la partición Y está dentro de W entonces esa partición ya no es valida, porque se vuelve inservible para hacer seguir partiendo estados, entonces si Y está en W, reemplazamos Y por las particiones. Luego, en caso de que Y no esté verificamos cuál es la partición más nos conviene guardar. Ya que debido al lemma anterior solo debmos guardar uno de los estados de la partición.

6.1 Código

```
Automata Automata::hopcroft(){

    // Creamos los conjuntos P y W
    set<set<int>> P;
    set<set<int>> W;

    reverse_2();

    set<int> notFinals;
    set<int> Finals;

    // Primera particion
    // Partimos los estados de aceptacion de los no aceptacion
    for(int i = 0; i < this->getSize(); i++){
        if(this->stateFinal[i] == true) Finals.insert(i);
        else notFinals.insert(i);
    }
    P.insert(Finals);
    P.insert(notFinals);
    W.insert(Finals);
    W.insert(notFinals);

    while(!W.empty()){
        // Escogemos y eliminamos un conjunto de W
        // y lo guardamos en A
        set<int> A = *(W.begin());
        W.erase(W.begin());

        for(int c = 0; c < 2; c++){

            // Creamos el conjunto X
            // X: Conjunto de estados que llegan a algun estado de A
            //    con transicion c.
            set<int> X = this->can_reach(A, c);
            for(auto Y = P.begin(); Y!=P.end(); Y++){
                set<int> Intersect = intersection(X,*Y);
                set<int> Difference = difference(*Y,X);
                if(!Intersect.empty() && !Difference.empty()){
                    P.erase(Y);
                    P.insert(Intersect);
                    P.insert(Difference);
                    if(W.find(*Y) != W.end()){
                        W.erase(Y);
                        W.insert(Intersect);
                        W.insert(Difference);
                    }
                    else{
                        if(Intersect.size() <= Difference.size()){
                            W.insert(Intersect);
                        }
```

```
                        else{
                            W.insert(Difference);
                        }

                    }
                }
            }
        }
    }

    Automata MinAutomata;
    MinAutomata.states = vector<state>(P.size());
    MinAutomata.stateFinal = vector<bool>(P.size(),false);

    // añadimos estado inicial y estados finales
    int i = 0;
    for(auto it = P.begin(); it!=P.end(); it++){
        for(auto it1=(it)->begin(); it1!=(it)->end(); it1++){
            if(*it1 == initialState){
                MinAutomata.initialState = i;
            }
            else if(stateFinal[*it1] == true){
                MinAutomata.stateFinal[i] = true;
            }
        }
        i++;
    }
    // añadimos los estados finales
    i = 0;
    for(auto it = P.begin(); it!=P.end(); it++){
        auto it1 = (it)->begin();
        for(int transition: {0,1}){
            int destination = states[*it1].adjacentes[transition];
            int pos = search_state(destination, P);
            MinAutomata.states[i].adjacentes[transition] = pos;
        }
        i++;
    }
    return MinAutomata;

}
```

7 Experimentación

Myhill Nerode	Myhill Nerode Op	Hopcroft	Huffman Moore
175	18	304	140
113	34	326	210
182	37	283	148
136	20	270	137
102	13	283	93
160	37	290	133
133	26	241	158
139	19	252	147

Table 1: Tiempo de ejecución en microsegundos

References

Hopcroft, John, Rajeev Motwani, and Jeffrey Ullman. *Introduction to Auto- mata Theory, Languages, and Computation, 2Nd Edition.*

Torczon, Keith Cooper Linda. *Engineering a Compiler*. Rice University, Houston, Texas: Morgan Kaufmann, 2011.