Universidad de Ingeniería y Tecnología

CIENCIA DE LA COMPUTACIÓN

AUTÓMATA SINCRONIZABLE

Proyecto de Teoría de la Computación

Autor:
Neira Riveros, Jorge Luis

Profesor:
Gutierrez Alva, Juan Gabriel

August 4, 2021

***Abstract*—Sea A un autómata finito determinista (AFD), se dice que una cadena w es sincronizadora si envía a todos los estados de A hacia un mismo estado. Entonces se dice que A es un autómata sincronizable.**

En este trabajo se abordan 3 problemas relacionados con autómatas sincronizables y las implementaciones de los algoritmos que los resuelven. El algoritmo MIN-SINC, de complejidad exponencial, devuelve una cadena sincronizadora de tamaño mínimo de un autómata si es sincronizable. El algoritmo DEC-SINC devuelve si un autómata es o no es sincronizable en tiempo cuadrático. Y el algoritmo CAD-SINC devuelve una cadena sincronizadora, no necesariamente mínima, en tiempo cúbico. Luego, probaremos estas implementaciones en el problema Magical Stones 2 de CS Academy.

Finalmente, realizamos pruebas de las implementaciones en autómatas generados aleatoriamente de acuerdo al número de estados.

1. INTRODUCCIÓN

Un autómata finito es un modelo matemático abstracto con reglas definidas para decidir las transiciones entre sus estados. En Ciencia de la Computación, encontramos muchos ejemplos de sistemas de estados finitos, debido a esto, la teoría de autómatas es uno de los temas fundamentales de esta área. En su forma más básica, una autómata representa a una máquina capaz de recibir y procesar alguna cadena de un determinado lenguaje.

Un autómata se puede definir como una 5-tupla (Q, Σ, δ, q_0, F), donde Q son los estados, Σ el alfabeto, δ la función de transición, y F el conjunto de estados finales. Este autómata procesa una cadena w del alfabeto caracter por caracter, y si llega a un estado final de F entonces la acepta, caso contrario la rechaza.

Uno de los temas más resaltantes en la teoría de autómatas es la sincronización de autómatas. Se dice que un autómata es sincronizable si existe una cadena sincronizadora w que envía cualquier estado del autómata hacia un mismo estado. Más formalmente, sea el autómata A definido por (Q, Σ, δ, q_0, F), y w una cadena sincronizadora de A, entonces $\delta(q, w) = q'$ para todo $q \in Q$ y un $q' \in Q$ [3]. Conocer si un autómata es sincronizable es útil para distintas aplicaciones, ya que si se detecta algún conflicto con el funcionamiento del autómata, entonces, con una cadena sincronizadora, es posible llevarlo desde cualquier estado a un estado en específico (q'), dando lugar al reseteo de la máquina.

Dentro del área de sincronización de autómatas, existen muchas preguntas, aun hay algunas sin resolver, pero, en este trabajo, estudiaremos 3 problemas relacionados a este tema. A partir de aquí, definimos el autómata A como una 3-tupla (Q, Σ, δ) ya que no necesitamos el estado inicial ni los finales. Como ejemplo utilizaremos un autómata de Cerny de tamaño 4, cuya cadena sincronizadora tiene tamaño $(n-1)^2$ para n estados [1].

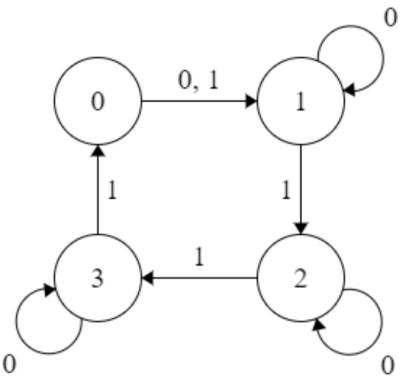

Fig 1.1: Autómata de Cerny de tamaño 4

2. DEFINICIÓN DE LOS PROBLEMAS

A. *Problema* MIN-SINC

El problema MIN-SINC consiste en encontrar una cadena sincronizadora de tamaño mínimo para un autómata en caso sea sincronizable, de lo contrario, detecta si no lo es.

Este problema tiene una solución directa pero ineficiente en términos de complejidad, la idea es encontrar una cadena que transforme el conjunto de estados Q de un autómata A a un único estado q'. Esto se consigue con una solución de fuerza bruta a partir del autómata potencia $P(A)$.

El autómata potencia $P(A)$ se define como un nuevo autómata donde: 1) los estados son subconjuntos no vacíos de Q, es decir, $P(Q)$, 2) se mantiene el alfabeto Σ, y 3) la función δ evalúa ahora cada estado simple de los conjuntos en $P(Q)$ [3].

Notamos que, una cadena sincronizadora, si existe, corresponde con una ruta o $path$ desde el estado en $P(A)$ que contiene a todos los estados Q hacia algún estado de tamaño 1. De esta forma definimos si el autómata A es sincronizable en tiempo exponencial, ya que necesitamos formar el autómata potencia.

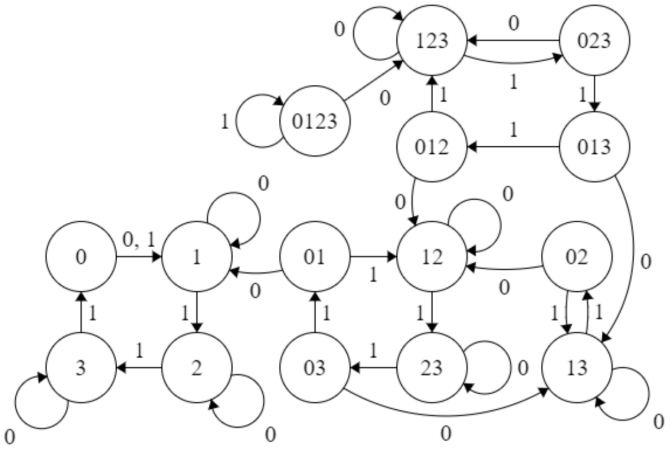

Fig 2.1: Autómata Potencia del autómata de Cerny de tamaño 4

El problema de obtener MIN-SINC pertenece a la clase NP: escogemos no determinísticamente una cadena w de longitud l y verificamos si es sincronizadora en tiempo polinomial ($l|Q|$) [2].

Para encontrar MIN-SINC en el autómata $P(A)$, realizamos un recorrido por anchura (BFS) desde el estado que contiene a Q. La cadena sincronizadora de tamaño mínimo se encontrará en la ruta hacia el estado de tamaño 1 más cercano al estado Q. El algoritmo tiene una complejidad $O(n2^n)$ debido a la construcción de $P(A)$ y $O(2^n)$ por el recorrido en anchura.

Para el autómata de Cerny de tamaño 4, una cadena sincronizadora de tamaño mínimo sería 011101110.

1) Algoritmo exponencial para MIN-SINC*:* Como el algoritmo es exponencial, entonces para valores n muy grandes, formar el autómata potencia costará mucho tiempo. En lugar de crearlo completamente, iniciaremos el BFS desde el estado que tiene a todos los estados del autómata, y, en cada iteración, formamos los nuevos estados que se generan al aplicar la función de transición, y evaluamos su cadena mínima. Como el recorrido es por anchura, entonces la cadena encontrada será mínima. Cuando encontremos un estado de tamaño 1, entonces terminamos el algoritmo. Si nunca encuentra un estado de tamaño 1, entonces el programa devuelve "NO" que indica que el autómata no es sincronizable.

function MIN-SINC(A, n)
  $P(A) \leftarrow Autómata - Potencia(A)$
  $q \leftarrow Q \in P(A)$ ▷ Q tiene a todos los estados de A
  $Qu = \{q\}$ ▷ Inicializa la cola
  $visited(q) = True$
  $shortString(q) =$ ""
  while $Q \neq 0$ **do**
    $u = DESENCOLAR(Qu)$
    $v0 = \delta(u, 0)$
    $v1 = \delta(u, 1)$
    if $visited(v0) = False$ **then**
      $visited(v0) = True$
      $shortString(v0) = shortString(u) +$ "0"
      $ENCOLAR(Qu, v0)$
    end if
    if $visited(v1) = False$ **then**
      $visited(v1) = True$
      $shortString(v1) = shortString(u) +$ "0"
      $ENCOLAR(Qu, v1)$
    end if
    if $v0.size = 1$ **then**
      return $shortString(v0)$
    end if
    if $v1.size = 1$ **then**
      return $shortString(v1)$
    end if
  end while
  return "NO"
end function

2) Implementación en C++ del algoritmo exponencial para MIN-SINC*:* La implementación se realiza en C++, la encontramos al final del documento. Como se observa, cada estado es representado como un vector de booleanos, donde cada elemento representa si un estado simple está o no en los estados de $P(A)$. El recorrido en anchura puede tomar como máximo $O(n \times 2^n)$ ya que tiene 2^n estados y 2×2^n transiciones, y en cada iteración se toma un $O(n)$ para construir los nuevos estados. Existen otras formas de representar a los estados, se pueden utilizar $sets$ o $unordered\_sets$ pero el hashing y los accesos logarítmicos hacen ineficiente el algoritmo. Otra forma desarrollada consiste en utilizar potencias de 2 como variables y utilizar operaciones en bit, pero solo podríamos utilizar hasta 32 o 64 estados de acuerdo al tipo de variable utilizado.

B. Problema DEC-SINC

El problema DEC-SINC detecta si un autómata es sincronizable o no. Este problema puede ser resuelto en tiempo polinomial utilizando las propiedades sobre autómatas sincronizables que Cerny demostró.

Teorema 1: Sea A un autómata sincronizable, entonces A tiene, por lo menos, un estado sumidero [1].

Un estado sumidero es un estado al cual todos los demás estados pueden alcanzarlo.

Teorema 2: Sea A un autómata, entonces A es sincronizable si y solo si para cada par de estados $u, v \in Q$ se cumple que existe una cadena w tal que $\delta(u, w) = \delta(v, w)$ [1].

Utilizaremos ambos teoremas para desarrollar un algoritmo que tenga una complejidad $O(n^2)$.

Primero, verificamos que tenga un estado sumidero, para lo cual debemos crear un nuevo autómata con los mismos estados pero con la función de transición invertida. Luego, realizamos un BFS por cada estado hasta encontrar alguno tal que alcance a todos los demás, esto demuestra que todos los estados lo alcanzarían en el AFD original. Esta ejecución toma $O(n)$ por cada BFS, y como máximo se aplicaría a cada estado, en total toma $O(n^2)$. Si no existe el estado sumidero, el autómata no es sincronizable.

Segundo, procesaremos el autómata $P^2(A)$, el cual se compone por estados que son subconjuntos de Q de tamaño 2, es decir, tiene estados unitarios y de tamaño 2. La función de transición δ evalúa a cada estado de los conjuntos. Para el autómata de Cerny de tamaño 4, el autómata $P^2(A)$ sería el siguiente.

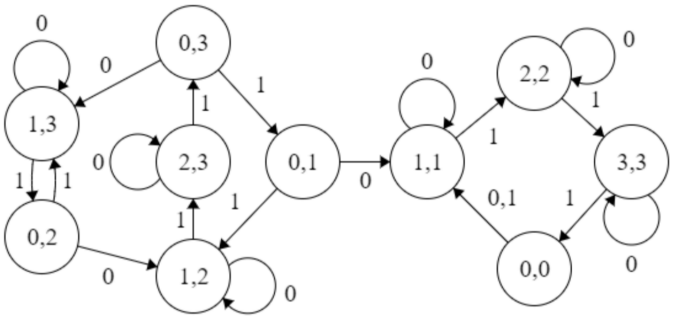

Fig 2.2: Autómata Potencia 2 del autómata de Cerny de tamaño 4

En este autómata, debemos verificar que cada estado de tamaño 2 alcanza a algún estado de tamaño 1. Esto se puede realizar con un BFS desde todos los estados de tamaño 2. Para evitar hacer un BFS por cada estado, se guarda el estado de visitado para cada uno. Sin embargo, realizaremos este algoritmo desde otro punto de vista pero misma complejidad, el cual nos permitirá entender fácilmente el algoritmo para el siguiente problema.

Para empezar, asignamos codificaciones a los estados para trabajar con ellos de forma más sencilla. Esto se hace en $O(n^2)$.

	0	1	2	3
0	0	4	5	6
1	4	1	7	8
2	5	7	2	9
3	6	8	9	3

Fig 2.3: Codificación en Autómata Potencia 2 del autómata de Cerny de tamaño 4

Luego, construimos el autómata inverso $P^2(A)$, evitando los bucles.

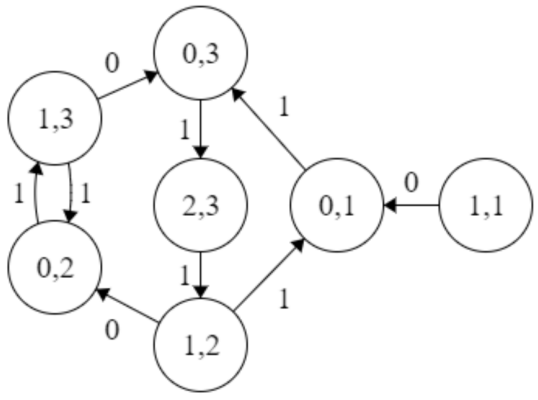

Fig 2.4: Autómata Potencia 2 inverso del autómata de Cerny de tamaño 4 sin bucles

Note que, si realizamos un BFS desde los estados unitarios en el autómata inverso $P^2(A)$, entonces alcanzaríamos a los estados de tamaño 2 que cumplen la condición de: el estado de tamaño 2 alcanza a un estado de tamaño 1 en el $P^2(A)$ original. Por lo tanto, si cubrimos a todos los estados de tamaño 2 entonces el autómata es sincronizable. Veámoslo en otro ejemplo.

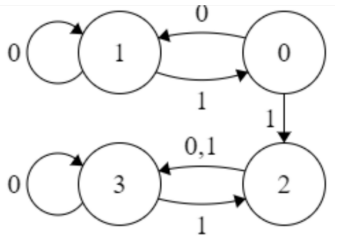

Fig 2.5: Autómata sincronizable

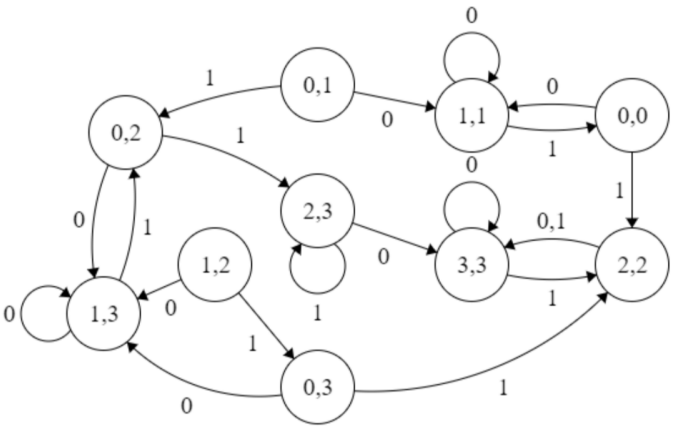

Fig 2.6: Autómata Potencia 2 de un autómata sincronizable

Si realizamos un BFS desde los estados de tamaño 1 en el autómata inverso $P^2(A)$ alcanzaríamos a todos los estados de tamaño 2 si el autómata fuera sincronizable. En un autómata no sincronizable, no se alcanzarían a todos los estados de tamaño 2, por ejemplo.

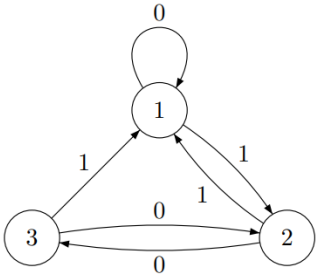

Fig 2.7: Autómata no sincronizable [1]

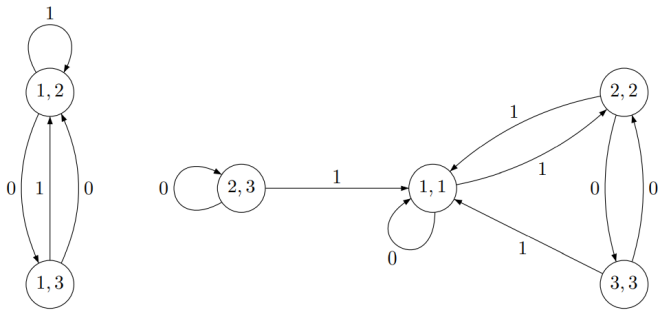

Fig 2.8: Autómata Potencia 2 de un autómata no sincronizable [1]

1) Algoritmo polinomial para DEC-SINC*:* La implementación de las funciones se detallan en su código.

function DEC-SINC(A, n)
  $A' \leftarrow Autómata - Inverso(A)$
  $q \leftarrow Sink - State(A)$
  if $q = NULL$ **then**
    return $False$
  end if
  $I^2 \leftarrow Autómata^2 - Inverso(A)$
  $Qu = \{q : q \in Q\}$ ▷ Inicializa la cola con estados unitarios
  $visited(q : q \in Q) = True$
  while $Q \neq 0$ **do**
    $u = DESENCOLAR(Qu)$
    for $uv \in I^2$ **do**
      if $visited(v) = False$ **then**
        $visited(v) = True$
        $ENCOLAR(Qu, v)$
      end if
    end for
  end while
  if $\exists q \in I^2 : visited(q) = False$ **then**
    return $False$
  else
    return $True$
  end if
end function

2) Implementación en C++ del algoritmo polinomial para DEC-SINC*:* El código implementado de este algoritmo se encuentra al final de este documento. La verificación de la existencia de un estado sumidero se realiza en $O(n^2)$, la función de codificación de pares toma $O(n^2)$, la que crea el autómata inverso $P^2(A)$ también toma $O(n^2)$, y el BFS a partir de los estados unitarios también toma $O(n^2)$, ya que recorre como máximo $O(n^2)$ estados y $O(n^2)$ transiciones. En total, el algoritmo es $O(n^2)$.

C. Problema CAD-SINC

El problema CAD-SINC consiste en hallar una cadena sincronizadora, no necesariamente de tamaño mínimo. Para este problema, ya se sabe que el autómata es sincronizable. Entonces, para resolverlo, aplicaremos el algoritmo de Eppstein [2].

Eppstein realizó una mejora al algoritmo de Natarajan para encontrar una cadena sincronizadora de un autómata sincronizable. El algoritmo de Natarajan se describe a continuación [2]:

function NATARAJAN-ALGORITHM(A, n)
  $X \leftarrow Q$
  $\tau \leftarrow$ ""
  while $|X| > 1$ **do**
    $pick\ q_i, q_j \in X : q_i \neq q_j$
    $find\ a\ secuence\ \sigma\ taking\ s_i\ and\ s_j\ to\ the\ same\ state$
    $X \leftarrow \delta(X, \sigma)$
    $\tau \leftarrow \tau\sigma$
  end while
end function

Este algoritmo tiene una complejidad $O(n^4)$. Los pasos de mayor complejidad son: encontrar σ y aplicarlo a X [2]. Eppstein propone un pre-procesamiento antes de realizar este algoritmo que permitirá encontrar σ y aplicarlo a X en menor tiempo. Este pre-procesamiento se realiza sobre el autómata $P^2(A)$, y disminuye la complejidad del algoritmo a $O(n^3)$.

El pre-procesamiento que realiza Eppstein puede realizarse también para el autómata inverso $P^2(A)$, solo necesitamos guardar la información de la transición para cada estado (0 o 1). Para explicarlo, lo desarrollaremos en el siguiente autómata.

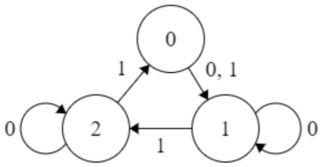

Fig 2.9: Pre-procesamiento 1

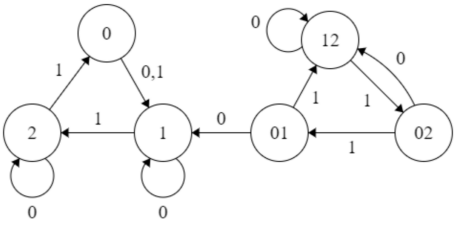

Fig 2.10: Pre-procesamiento 2

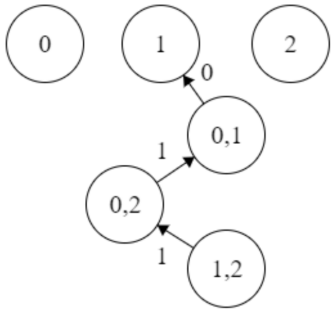

Fig 2.11: Pre-procesamiento 3

Note que pueden existir muchos más transiciones entre los estados en otros autómatas, sin embargo, al realizar el BFS inverso desde los estados de tamaño 1, se generará un bosque, cuyas raíces corresponderán con los estados de tamaño 1. Para cada estado debemos guardar la cadena que envía el estado de tamaño 2 a uno de tamaño 1, y la transición de cada estado unitario con dicha cadena. Esta información la guardamos en 2 tablas, una para las cadenas $O(n^2)$ y otra para las transiciones $O(n^3)$.

La tabla de cadenas se llena a partir del BFS inverso, donde los estados unitarios tienen cadena vacía, y cada estado visitado tendría su cadena igual a su transición más la cadena de su padre.

	0	1	2	0,1	0,2	1,2
w				0	10	110

Fig 2.12: Pre-procesamiento 4

La tabla de transiciones es una tabla que tiene $n \times (n+1)/2$ filas (una por cada estado en el autómata $P^2(A)$) y n columnas (para cada estado unitario). Las filas correspondientes a estados unitarios se llenan al inicio, y tienen el mismo valor pues no hay cadena para transicionar.

	0	1	2
0	0	1	2
1	0	1	2
2	0	1	2
01	1	1	2
02	1	2	1
12	2	1	1

Fig 2.13: Pre-procesamiento 5

Luego evalúa por nivel según el recorrido BFS del Pre-procesamiento 3. Entonces, para 01, evalúa la transición $\delta(\{0,1,2\},0)$, luego para 02 se evalúa la transición $\delta(\{0,1,2\},10)$, y para 12 se evalúa la transición $\delta(\{0,1,2\},110)$.

Para hacer esta transición eficiente, a partir del BFS, note que se cumple lo siguiente: si el par x ya está computado, entonces ya tiene una cadena w_x asignada y la transición $\delta(q_i, w_x) : i \in [0, n-1]$ establecida. Para computar el siguiente par y, hijo de x en el BFS inverso, cuya transición a x se da con el caracter c_y (0 o 1), debemos realizar lo siguiente: 1) calcular $z = \delta(q_i, c_y) : i \in [0, n-1]$, 2) buscar en x la transición $\delta(z, w_x)$. Esto se cumple a partir de $\delta(q_i, c_y w_x) = \delta(\delta(q_i, c_y), w_x)$. Gráficamente, para el estado 12, el cálculo se vería así.

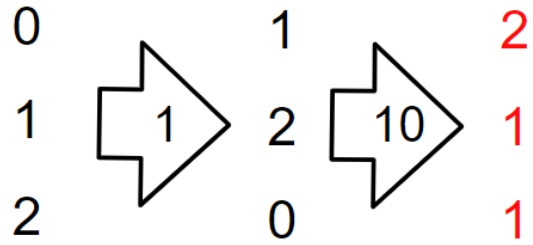

Fig 2.14: Pre-procesamiento 6

Este pre-procesamiento permite aplicar el algoritmo de Nataraján en tiempo $O(n^3)$, ya que las funciones de encontrar σ y aplicarlo a X se obtienen directamente de la tabla. El pre-procesamiento toma $O(n^3)$ y el algoritmo en sí toma $O(n^2)$, en total es $O(n^3)$.

1) Algoritmo polinomial para CAD-SINC*:*

function CAD-SINC(A, n)
  $I^2 \leftarrow Autómata^2 - Inverso(A)$
  $Qu = \{q : q \in Q\}$ ▷ Inicializa la cola con estados unitarios
  $visited(q : q \in Q) = True$
  $minString(q : q \in Q) = ""$
  $state(q_i, j) = j : q_i \in I^2, j \in [0, n-1]$
  while $Q \neq 0$ **do**
    $u = DESENCOLAR(Qu)$
    for $uv \in I^2$ **do**
      if $visited(v) = False$ **then**
        $visited(v) = True$
        $ENCOLAR(Qu, v)$
        $minString(v) = v.transition + minString(u)$
        $state(v, i) = state(u, \delta(i, v.transition))$
      end if
    end for
  end while
  $X \leftarrow Q$
  $\tau \leftarrow ""$
  while $|X| > 1$ **do**
    $pick\ q_i, q_j \in X : q_i \neq q_j$
    $\tau = \tau + minString(q_i q_j)$
    $X \leftarrow \delta(X, \sigma)$ ▷ De $state$
    $\tau \leftarrow \tau\sigma$
  end while

return τ
end function

2) Implementación en C++ del algoritmo polinomial para CAD-SINC*:* El código implementado de este algoritmo se encuentra al final de este documento. La función de codificación toma $O(n^2)$, la que crea el autómata inverso $P^2(A)$ también toma $O(n^2)$, el pre-procesamiento en el BFS a partir de los estados unitarios toma $O(n^3)$, ya que recorre debemos computar la transición de los estados en Q desde cada estado de tamaño 2, finalmente, aplicamos el algoritmo de Natarajan, la cual toma ahora $O(n^2)$. En total, el algoritmo es $O(n^3)$.

3. PROBLEMA MAGICAL STONES 2

El problema explica que existen dos tipos de hechizos, uno W y uno B, que se aplican a unas piedras mágicas, y cada uno envía a la piedra a un nuevo estado. El objetivo es enviar a todas las piedras mágicas a un mismo estado. Este problema se asemeja a encontrar una cadena que sincronice a todas las piedras mágicas, donde el alfabeto es W y B, y los estados de las piedras son los estados de un autómata. Para resolverlo, debemos juntar los códigos de DEC-SINC y CAD-SIN para detectar si es sincronizable, y, si lo es, devolver una cadena sincronizadora no necesariamente de tamaño mínimo. El código que resuelve este problema está al final de este documento, y tiene una complejidad de $O(n^3)$.

Source Summary Results Compilation messages

Test Number	CPU Usage	Memory Usage	Result
4	114 ms	5624 KB	OK
3	102 ms	16.7 MB	OK
2	78 ms	5544 KB	OK
1	5 ms	912 KB	OK
0	3 ms	892 KB	OK

Fig 3.1: Resultado del problema Magical Stones 2

4. EXPERIMENTACIÓN NUMÉRICA

A. Programa completo

Desarrollamos un programa que ejecute alguno de los algoritmos sobre algún autómata, el cual puede ser ingresado por consola en el formato adecuado, generado aleatoriamente de acuerdo a la cantidad de datos, o genera un autómata de Cerny de un tamaño dado.

```
--------------------Proyecto de Teoria de la Computacion--------------------
1. Problema Min-Sinc O(2^n).
2. Problema Dec-Sinc O(n^2).
3. Problema Cad-Sinc O(n^3).
4. Salir del programa.
Ingrese una opcion:
```

Fig 4.1: Menú de algoritmos

```
--------------------Eleccion de automata--------------------
1. Ingresar AFD por consola.
2. Generar AFD aleatorio.
3. Generar AFD sincronizable de Cerny.
4. Regresar al menu anterior.
5. Salir del programa.
Ingrese una opcion:
```

Fig 4.2: Menú de creación de autómatas

Para cada algoritmo, creamos 10 autómatas aleatorios de distintos tamaños para comparar su complejidad. El programa mide el tiempo que demora cada algoritmo. Los resultados se muestran en la parte final de este trabajo. Debemos tener en cuenta que, según Cerny, un autómata de alfabeto de tamaño 2 tiene un probabilidad de $1 - O(1/n)$ de ser sincronizable, por lo que, un autómata aleatorio para un n alto es sincronizable con alta probabilidad.

B. Resultados

Ejecutamos los algoritmos para tamaños de autómatas de 10 a 100 con paso de 10. Se realizan 15 pruebas en cada uno y se toma un promedio.

Si graficamos los tres algoritmos juntos, se observa el comportamiento exponencial de MIN-SINC.

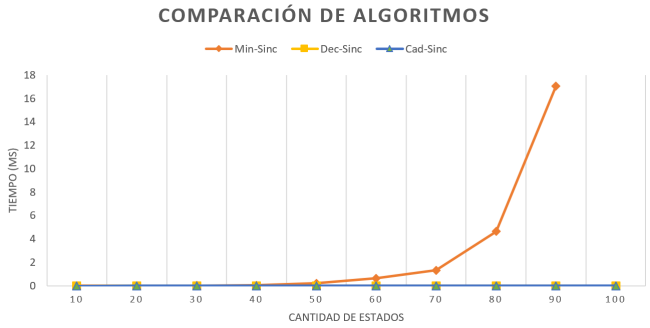

Fig 4.3: Gráfica 1

Quitamos algunos valores en MIN-SINC para observar de forma más clara los 3 algoritmos. Aquí podemos ver que CAD-SINC crece más rápido que DEC-SINC, ya que el primero es $O(n^3)$ y el segundo $O(n^2)$.

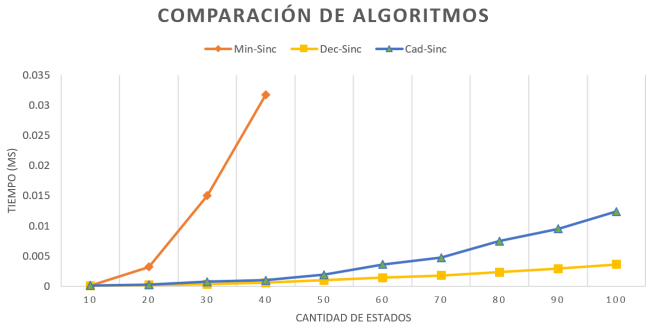

Fig 4.4: Gráfica 2

Por último, observamos el crecimiento polinomial entre DEC-SINC y MIN-SINC.

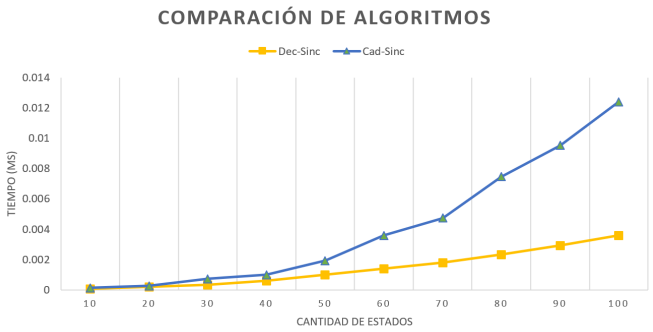

Fig 4.5: Gráfica 3

REFERENCIAS

[1] ČERNÝ, JAN, *Poznámka k homogénnym experimentom s konečnými automatmi*. Matematicko-fyzikálny Časopis, Slovenská Akadémia, Vied 14 (1964), No. 3, 208–216.

[2] EPPSTEIN, DAVID, *Reset sequences for monotonic automata*. SIAM J. Comput, 19 (1990), 500-510.

[3] VOLKOV, MIKHAIL, *Synchronizing Automata and the Černý Conjecture*. LATA 2008: Language and Automata Theory and Applications pp 11-27.

Repositorio en Github: https://github.com/jneirar/TeoriaComputacionProyecto.git

5. ANEXOS

A. *Código para* MIN-SINC

```
string Min_Sinc(vector<vector<ll>> transition, ll n){       //O(n*2^n)
    //vbool = vector<bool>
    vbool sInit(n, 1);                                      //n
    //BFS
    unordered_map<vbool, string> shortString;               //max 2^n
    unordered_set<vbool> visited;                           //max 2^n
    queue<vbool> qState;
    qState.push(sInit);                                     //O(1)
    visited.insert(sInit);                                  //O(1)
    shortString[sInit] = "";                                //O(1)

    while(!qState.empty()){                                 //O(V + E) = O(2^n * transicion)
        vbool vCur = qState.front();                        //max n
        vbool sNext0(n, 0), sNext1(n, 0);                   //max n
        for(ll i = 0; i < n; i++){                          //O(n)
            if(vCur[i]){                                    //O(1)
                sNext0[transition[i][0]] = 1;               //O(1)
                sNext1[transition[i][1]] = 1;               //O(1)
            }
        }
        if(!visited.count(sNext0)){                         //O(n)
            visited.insert(sNext0);                         //O(n)
            shortString[sNext0] = shortString[qState.front()] + '0';//O(n)
            qState.push(sNext0);                            //O(1)
        }
        if(!visited.count(sNext1)){
            visited.insert(sNext1);
            shortString[sNext1] = shortString[qState.front()] + '1';
            qState.push(sNext1);
        }
        qState.pop();                                       //O(1)
        int sum = 0;                                        //O(1)
        for(int i = 0; i < n; i++) sum += sNext0[i];        //O(n)
        if(sum == 1)                                        //O(1)
            return shortString[sNext0];                     //O(1)
        sum = 0;                                            //O(1)
        for(int i = 0; i < n; i++) sum += sNext1[i];        //O(n)
        if(sum == 1)                                        //O(1)
            return shortString[sNext1];                     //O(1)
    }
    return ".";
}
```

B. Funciones comunes

```
vector<vector<ll>> pairCodification(ll n){                  //O(n^2)
    vector<vector<ll>> pairToCode(n);
    for(ll i = 0; i < n; i++)   pairToCode[i].resize(n);    //O(n)
    ll code = n;                                            //O(1)
    for(ll i=0; i<n; i++){                                  //O(n^2)
        for(ll j=i; j<n; j++){                              //O(n)
            if(i == j)                                      //O(1)
                pairToCode[i][i] = i;                       //O(1)
            else{                                           //O(1)
                pairToCode[i][j] = code;                    //O(1)
                pairToCode[j][i] = code++;                  //O(1)
            }
        }
    }
    return pairToCode;
}

vector<vector<pair<ll, ll>>> pairInvertedTransition(ll n, vector<vector<ll>> transition, vector<vector<ll
    >> pairToCode){//O(n^2)
    vector<vector<pair<ll, ll>>> invertedTransitions(n*(n+1)/2);
    for(ll i=0; i<n; i++){                                                          //O(n * n) = O(n
        ^2)
        for(ll j=i+1; j<n; j++){                                                    //O(n)
            //Evito bucles, operaciones O(1)
            if(pairToCode[ transition[i][0] ][ transition[j][0] ] != pairToCode[i][j])
                invertedTransitions[pairToCode[ transition[i][0] ][ transition[j][0] ]].push_back(
                    make_pair(pairToCode[i][j], 0));
            if(pairToCode[ transition[i][1] ][ transition[j][1] ] != pairToCode[i][j])
                invertedTransitions[pairToCode[ transition[i][1] ][ transition[j][1] ]].push_back(
                    make_pair(pairToCode[i][j], 1));
        }
    }
    return invertedTransitions;
}
```

C. *Código para* DEC-SINC

```
bool Dec_Sinc(ll n, vector<vector<ll>> transition){          //O(n^2)
    vector<vector<ll>> invertedTransitions(n);
    for(ll i=0; i<n; i++){                                   //O(n)
        if(i != transition[i][0])   invertedTransitions[transition[i][0]].push_back(i);
        if(i != transition[i][1])   invertedTransitions[transition[i][1]].push_back(i);
    }
    bool sinkDetec = false;
    for(ll i=0; i<n; i++){                                   //O(n^2)
        vector<bool> visited(n, 0);
        queue<ll> q;
        visited[i] = 1;
        q.push(i);
        while(!q.empty()){                                   //BFS = O(n)
            for(ll e : invertedTransitions[q.front()]){
                if(!visited[e]){
                    visited[e] = 1;
                    q.push(e);
                }
            }
            q.pop();
        }
        for(ll i = 0; i < n; i++){                           //O(n)
            if(!visited[i]) break;
            if(i == n - 1 && visited[i]) sinkDetec = true;
        }
        if(sinkDetec)  break;
    }
    if(!sinkDetec) return false;

    ll n_2 = n*(n+1)/2;
    vector<vector<ll>> pairToCode = pairCodification(n);     //O(n^2)
    vector<vector<pair<ll, ll>>> invertedTransitions2 = pairInvertedTransition(n, transition, pairToCode);
          //O(n^2)
    queue<ll> q;
    vector<bool> visited(n_2, 0);
    for(ll i=0; i<n; i++){                                   //O(n)
        visited[i] = 1;
        q.push(i);
    }
    while(!q.empty()){                                       //BFS = O(n^2)
        for(pair<ll, ll> pcode : invertedTransitions2[q.front()]){
            //pcode.first = state codificado
            if(!visited[pcode.first]){
                visited[pcode.first] = 1;
                q.push(pcode.first);
            }
        }
        q.pop();
    }
    for(ll i=0; i<n_2; i++)                                  //BFS = O(n^2)
        if(!visited[i]) return false;
    return true;
}
```

D. *Código para* CAD-SINC

```
string Cad_Sinc(ll n, vector<vector<ll>> transition){    //O(n^3)
    ll n_2 = n*(n+1)/2; //total pares
    vector<vector<ll>> pairToCode = pairCodification(n); //O(n^2)
    vector<vector<pair<ll, ll>>> invertedTransitions2 = pairInvertedTransition(n, transition, pairToCode);
          //O(n^2)

    vector<vector<ll>> pairState(n_2);
    for(ll i = 0; i < n_2; i++) pairState[i].resize(n);  //O(n^2)
    vector<string> pairMinString(n_2, "");
    vector<ll> visited(n_2, 0);
    queue<ll> q;
    for(ll i = 0; i < n; i++){                           //O(n^2)
        q.push(i);
        for(ll j=0; j < n; j++){                         //O(n)
            pairState[i][j] = j;
        }
    }
    while(!q.empty()){                                   //BFS + O(pairState) = O(n^3)
        ll cur = q.front();
        q.pop();
        for(auto pcode : invertedTransitions2[cur]){
            if(visited[pcode.first])    continue;
            visited[pcode.first] = 1;
            q.push(pcode.first);
            if(pcode.second)
                pairMinString[pcode.first] = "1" + pairMinString[cur];
            else
                pairMinString[pcode.first] = "0" + pairMinString[cur];
            for(ll i = 0; i < n; i++)                    //O(n)
                pairState[pcode.first][i] = pairState[cur][ transition[i][pcode.second] ];
        }
    }
    unordered_set<ll> X;
    for(ll i=0; i<n; i++)   X.insert(i);            //O(n)
    string cad = "";
    ll s1, s2, codes1s2;
    while(X.size() > 1){                            //Se realiza n veces como maximo. O(n^2)
        auto it = X.begin();
        s1 = *(it++);
        s2 = *it;
        codes1s2 = pairToCode[s1][s2];              //O(1)
        cad += pairMinString[codes1s2];             //O(1)
        unordered_set<ll> Y;
        for(auto state : X)                         //O(n)
            Y.insert(pairState[codes1s2][state]);
        X = Y;
    }
    return cad;
}
```

E. Código para resolver el problema Magical Stones 2 Parte 1

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <utility>
#include <set>
#include <unordered_set>
#include <map>
#include <unordered_map>
#include <queue>
#include <algorithm>
#include <array>

using namespace std;
#define fastio ios_base::sync_with_stdio(false); cin.tie(NULL)
#define ll long long

const ll MOD = 1e9+7;

void solve(){
    ll n, num;   cin >> n;
    vector<vector<ll>> transition(n);
    for(ll i=0; i<n; i++)   transition[i].resize(2);
    for(ll i=0; i<n; i++){
        cin >> num;
        transition[i][0] = num-1;
    }
    for(ll i=0; i<n; i++){
        cin >> num;
        transition[i][1] = num-1;
    }

    vector<vector<ll>> invertedTransitions(n);
    for(ll i=0; i<n; i++){
        if(i != transition[i][0])   invertedTransitions[transition[i][0]].push_back(i);
        if(i != transition[i][1])   invertedTransitions[transition[i][1]].push_back(i);
    }

    bool sinkDetec = false;
    for(ll i=0; i<n; i++){
        vector<bool> visited(n, 0);
        queue<ll> q;
        visited[i] = 1;
        q.push(i);
        while(!q.empty()){
            for(ll e : invertedTransitions[q.front()]){
                if(!visited[e]){
                    visited[e] = 1;
                    q.push(e);
                }
            }
            q.pop();
        }
        for(ll i = 0; i < n; i++){
            if(!visited[i]) break;
            if(i == n - 1 && visited[i]) sinkDetec = true;
        }
        if(sinkDetec)  break;
    }
    if(!sinkDetec){
        cout << "impossible\n";
        return;
    }
    ll n_2 = n*(n+1)/2;
    vector<vector<ll>> pairToCode(n);
    for(ll i = 0; i < n; i++)   pairToCode[i].resize(n);
    ll code = n;
    for(ll i=0; i<n; i++){
        for(ll j=i; j<n; j++){
            if(i == j)
                pairToCode[i][i] = i;
            else{
                pairToCode[i][j] = code;
                pairToCode[j][i] = code++;
            }
        }
    }
```

F. Código para resolver el problema Magical Stones 2 Parte 2

```
    vector<vector<pair<ll, ll>>> invertedTransitions2(n_2);
    for(ll i=0; i<n; i++){
        for(ll j=i+1; j<n; j++){
            if(pairToCode[ transition[i][0] ][ transition[j][0] ] != pairToCode[i][j])
                invertedTransitions2[pairToCode[ transition[i][0] ][ transition[j][0] ]].push_back(
                    make_pair(pairToCode[i][j], 0));
            if(pairToCode[ transition[i][1] ][ transition[j][1] ] != pairToCode[i][j])
                invertedTransitions2[pairToCode[ transition[i][1] ][ transition[j][1] ]].push_back(
                    make_pair(pairToCode[i][j], 1));
        }
    }
    queue<ll> qu;
    vector<bool> visited(n_2, 0);
    for(ll i=0; i<n; i++){
        visited[i] = 1;
        qu.push(i);
    }
    while(!qu.empty()){
        for(pair<ll, ll> pcode : invertedTransitions2[qu.front()]){
            if(!visited[pcode.first]){
                visited[pcode.first] = 1;
                qu.push(pcode.first);
            }
        }
        qu.pop();
    }
    for(ll i=0; i<n_2; i++){
        if(!visited[i]){
            cout << "impossible\n";
            return;
        }
    }
    vector<vector<ll>> pairState(n_2);
    for(ll i = 0; i < n_2; i++) pairState[i].resize(n);
    vector<string> pairMinString(n_2, "");
    vector<bool> visited2(n_2, 0);
    queue<ll> q;
    for(ll i = 0; i < n; i++){
        q.push(i);
        for(ll j=0; j < n; j++){
            pairState[i][j] = j;
        }
    }
    while(!q.empty()){
        ll cur = q.front();
        q.pop();
        for(auto pcode : invertedTransitions2[cur]){
            if(visited2[pcode.first])    continue;
            visited2[pcode.first] = 1;
            q.push(pcode.first);
            if(pcode.second)
                pairMinString[pcode.first] = "B" + pairMinString[cur];
            else
                pairMinString[pcode.first] = "W" + pairMinString[cur];
            for(ll i = 0; i < n; i++)
                pairState[pcode.first][i] = pairState[cur][ transition[i][pcode.second] ];
        }
    }
    unordered_set<ll> X;
    for(ll i=0; i<n; i++)   X.insert(i);
    string cad = "";
    ll s1, s2, codes1s2;
    while(X.size() > 1){
        auto it = X.begin();
        s1 = *(it++);
        s2 = *it;
        codes1s2 = pairToCode[s1][s2];
        cad += pairMinString[codes1s2];
        unordered_set<ll> Y;
        for(auto state : X)
            Y.insert(pairState[codes1s2][state]);
        X = Y;
    }
    cout << cad << endl;
}
 int main(){
    fastio;    ll q;
    cin >> q; while(q--)    solve();
    return 0;
}
```

G. Datos obtenidos en la etapa de experimentación

	10	20	30	40	50	60	70	80	90
1	0	0.001	0.003	0.025	0.151	3.539	0.128	4.202	12.341
2	0	0	0.003	0.018	0.326	0.452	3.84	3.295	1.964
3	0	0.001	0.013	0.037	0.328	1.159	0.14	14.336	3.237
4	0	0.007	0.079	0.064	0.042	0.657	0.72	2.612	15.201
5	0	0.003	0.037	0.013	0.113	0.126	0.444	1.06	46.662
6	0.001	0.004	0.006	0.031	0.039	0.444	0.015	1.026	1.515
7	0	0.006	0.02	0.009	0.386	0.27	1.037	4.357	18.101
8	0	0	0.017	0.009	0.206	0.293	0.989	3.642	4.673
9	0	0.006	0.008	0.004	0.104	0.504	2.801	4.715	49.719
10	0.001	0.009	0.003	0.018	0.031	0.369	1.702	7.597	37.881
11	0	0	0.018	0.105	0.359	0.661	2.384	1.23	0.871
12	0	0.001	0.005	0.076	0.308	0.075	1.307	5.473	17.625
13	0	0.005	0.004	0.007	0.027	0.957	0.578	7.714	39.024
14	0	0.002	0.002	0.012	0.562	0.061	1.758	3.298	0.564
15	0	0.003	0.008	0.048	0.118	0.041	1.989	4.958	6.679
Avg	0.00013	0.0032	0.01507	0.03173	0.20667	0.64053	1.32213	4.63433	17.07047

Table 5.1: Resultados de MIN-SINC

	10	20	30	40	50	60	70	80	90	100
1	0	0	0	0	0.001	0.001	0.001	0.002	0.002	0.004
2	0	0	0	0	0.001	0.002	0.002	0.002	0.003	0.004
3	0	0	0.001	0.001	0.001	0.001	0.002	0.002	0.003	0.005
4	0	0.001	0	0.001	0.001	0.002	0.002	0.002	0.003	0.004
5	0	0	0	0	0.001	0.003	0.002	0.002	0.002	0.003
6	0.001	0	0	0.001	0.001	0.002	0.002	0.002	0.004	0.003
7	0	0	0	0.001	0.001	0.001	0.002	0.002	0.003	0.003
8	0	0.001	0.001	0.001	0.001	0.001	0.002	0.002	0.003	0.003
9	0	0	0.001	0	0.001	0.001	0.001	0.004	0.003	0.002
10	0	0	0.001	0	0.001	0.001	0.002	0.002	0.002	0.004
11	0	0	0.001	0	0.001	0.001	0.002	0.002	0.003	0.005
12	0	0	0	0.001	0.001	0.001	0.001	0.002	0.002	0.004
13	0	0	0	0.001	0.001	0.001	0.002	0.003	0.003	0.003
14	0	0.001	0	0.001	0.001	0.002	0.002	0.002	0.003	0.004
15	0	0	0	0.001	0.001	0.001	0.002	0.004	0.005	0.003
Avg	0.00007	0.00020	0.00033	0.00060	0.00100	0.00140	0.00180	0.00233	0.00293	0.00360

Table 5.2: Resultados de DEC-SINC

	10	20	30	40	50	60	70	80	90	100
1	0	0	0	0.001	0.001	0.003	0.005	0.008	0.009	0.012
2	0	0.001	0	0.001	0.003	0.003	0.004	0.006	0.01	0.012
3	0.001	0	0.001	0.001	0.002	0.003	0.005	0.007	0.01	0.012
4	0	0.001	0.001	0.001	0.002	0.005	0.004	0.009	0.01	0.012
5	0	0	0.001	0.001	0.002	0.003	0.004	0.007	0.009	0.013
6	0	0	0.001	0.001	0.002	0.003	0.005	0.006	0.009	0.012
7	0	0	0	0.001	0.001	0.003	0.004	0.01	0.01	0.012
8	0.001	0	0.001	0.001	0.002	0.004	0.005	0.007	0.008	0.013
9	0	0.001	0.001	0.001	0.002	0.003	0.005	0.007	0.01	0.013
10	0	0	0.001	0.001	0.002	0.003	0.005	0.006	0.009	0.014
11	0	0	0.001	0.001	0.002	0.006	0.004	0.007	0.008	0.012
12	0	0.001	0.001	0.001	0.002	0.003	0.005	0.007	0.012	0.012
13	0	0	0	0.001	0.002	0.003	0.005	0.006	0.009	0.012
14	0	0	0.001	0.001	0.002	0.004	0.005	0.008	0.009	0.012
15	0	0	0.001	0.001	0.002	0.005	0.006	0.011	0.011	0.013
Avg	0.00013	0.00027	0.00073	0.00100	0.00193	0.00360	0.00473	0.00747	0.00953	0.01240

Table 5.3: Resultados de CAD-SINC