ChatGPT

Resumen Unidad 3: Máquinas Secuenciales y Autómatas Finitos Deterministas

Conceptos Generales

En 1936 Alan M. **Turing** introdujo el modelo de máquina abstracta conocido como *Máquina de Turing*, y pocos años después (1938) **Claude Shannon** aplicó la lógica matemática al análisis de circuitos eléctricos **combinatorios y secuenciales**, sentando bases para una teoría formal de máquinas abstractas. Dentro de estos modelos, a mediados de los años 50 surgieron dos tipos de **máquinas secuenciales** simples propuestas por **George Mealy (1955)** y **Edward Moore (1956)**. Ambos modelos representan máquinas abstractas síncronas que funcionan en tiempo discreto y disponen de un número finito de estados internos. Estas máquinas secuenciales son esencialmente **traductoras**: a partir de una secuencia de símbolos de **entrada** producen una secuencia de símbolos de **salida**. Originalmente se asume que operan de forma permanente (cíclica), por lo que **no** incluían en su definición un estado de inicio ni de detención. Sin embargo, para analizar su comportamiento ante una secuencia dada, es práctico suponer un **estado inicial** (estado de arranque). Más adelante, extendiendo la formalización, se incorporó explícitamente un **estado inicial** único y uno o más **estados de finalización**, delimitando un conjunto finito de estados posibles: así se definió el **Autómata Finito**. En su forma más general, un autómata finito también puede considerarse una máquina traductora (genera salidas mientras lee la entrada y termina en algún estado final tras consumirla completamente, por lo que describe un procedimiento algorítmico finito). Con el tiempo, sin embargo, el interés se enfocó en el uso de autómatas finitos como **reconocedores de lenguajes**: se reemplazaron los estados de detención por **estados de aceptación**, y se relegó la capacidad traductora (inclusive definiendo máquinas sin salida). Es decir, se definió el autómata finito **reconocedor** puro, cuyo objetivo es **validar** cadenas de entrada más que traducirlas.

Como evolución adicional, se consideró permitir que el autómata moviera su cabezal de lectura **en ambos sentidos** sobre la cinta de entrada (no solo hacia adelante). A estos se les llama **autómatas finitos bidireccionales**. Esta capacidad de releer la entrada *no aumenta* el poder de cómputo (los lenguajes reconocibles siguen siendo los mismos), pero puede facilitar ciertos diseños de manera conceptual. También introduce el riesgo de **bucle infinito** si el control de movimiento no es cuidadoso, puesto que el cabezal podría oscilar indefinidamente. Los autómatas bidireccionales no son muy difundidos en la práctica, pero conceptualmente apuntan hacia modelos más potentes como la máquina de Turing (que extiende aún más las capacidades, incluyendo lectura/escritura y una memoria infinita).

Máquinas Secuenciales

Las **máquinas secuenciales** (o máquinas de estados finitos simples) se definen formalmente mediante una **quíntupla** de componentes y existen en dos variantes principales: la **Máquina de Mealy** y la **Máquina de Moore**. En ambos casos, la máquina opera en ciclos discretos leyendo símbolos de entrada, cambiando de estado y produciendo símbolos de salida según sus funciones de transición y de salida.

Máquina de Mealy

Una **Máquina de Mealy** se define como:

- **ME = (ΣE, ΣS, Q, f, g)**

Donde: - ΣE es el **alfabeto de entrada** (conjunto de símbolos que la máquina puede leer). - ΣS es el **alfabeto de salida** (símbolos que puede emitir). - Q es el **conjunto finito (no vacío) de estados** posibles de la máquina. - f: Q × ΣE → Q es la **función de transición**, que dado un estado actual y un símbolo de entrada determina el **próximo estado**. - g: Q × ΣE → ΣS es la **función de salida**, que dado un estado actual y un símbolo de entrada determina el **símbolo de salida** emitido.

En cada paso de tiempo discreto, estando la máquina en un estado `q_actual` y leyendo un símbolo de entrada `e_leído`, **Mealy** define que: - La máquina **emite** inmediatamente el símbolo de salida `s = g(q_actual, e_leído)`. - **Transita** a un nuevo estado `q_siguiente = f(q_actual, e_leído)`.

Tanto f como g dependen del **estado actual** *y* del símbolo de entrada leído simultáneamente. Por esta razón, es común representar ambas funciones mediante **tablas** (matrices) donde las filas corresponden a los estados y las columnas a cada símbolo de entrada posible, indicando en cada celda el siguiente estado (tabla de transición) o el símbolo de salida (tabla de salida). Otra representación equivalente es mediante un **grafo dirigido** (diagrama de estados): cada **nodo** representa un estado, y cada **arco** dirigido representa una transición etiquetada con el símbolo de entrada leído y el símbolo de salida producido (convención "entrada/salida" sobre la flecha).

Ejemplos prácticos (Máquinas de Mealy):
- *Ejemplo:* En el **Ejemplo 3.2** del material se modela un **dosificador de productos** usando una máquina de Mealy. Este dispositivo distribuye alternadamente elementos de entrada `a` y `b` hacia tres salidas (`c`, `d` y `e`) mediante dos válvulas que cambian de posición. La máquina secuencial correspondiente tiene 4 estados (codificando las posiciones de las válvulas A y B como `00`, `01`, `10`, `11`), con ΣE = {a, b} y ΣS = {c, d, e}. Sus funciones f y g se pueden tabular; por ejemplo, en cierto estado las transiciones con entrada `a` o `b` llevan a nuevos estados y producen salidas distribuyendo equitativamente los símbolos `a` y `b` entre `c`, `d` y `e`. Al procesar una secuencia de entrada alternada (por ejemplo `α = abababab` de largo 8), la máquina recorre un ciclo completo de estados y genera una secuencia de salida donde 4 símbolos `a` y 4 `b` de entrada quedan repartidos como 2 símbolos por la salida `c`, 3 por `d` y 3 por `e` (mostrando un desempeño balanceado del dosificador).
- *Ejercicio sugerido:* Analizar el comportamiento de otro dosificador similar (ver **Figura 3.23** en el material) mediante una máquina secuencial de Mealy, construyendo sus tablas de transición/salida y determinando la composición y cantidad de símbolos que salen por cada una de las tres salidas cuando se procesan secuencias alternadas de `a` y `b`. Comparar los resultados con el caso del Ejemplo 3.2 para verificar cómo se comporta este nuevo dosificador.

Máquina de Moore

Una **Máquina de Moore** comparte los mismos componentes que la de Mealy (ΣE, ΣS, Q, f, g) pero se **diferencia** en la definición de su función de salida. En una máquina de Moore, **g depende solo del estado actual**, no del símbolo de entrada del momento. Formalmente, para una máquina de Moore:

- **MO = (ΣE, ΣS, Q, f, g)**

Con: - f: Q × ΣE → Q (igual que en Mealy, determina el próximo estado según estado actual y símbolo de entrada). - g: Q → ΣS (la **función de salida** ahora asigna *a cada estado* un símbolo de salida fijo, independiente de la entrada actual).

Esto significa que **cada estado** de la máquina de Moore tiene asociado un símbolo de salida, el cual será emitido *cada vez que la máquina esté en ese estado*. Cuando llega un símbolo de entrada, la máquina cambia de estado según f, *y recién en el nuevo estado* se considera la salida correspondiente (es decir, la salida refleja el estado *previo* al próximo símbolo). **Moore introduce un retardo** entre la entrada y la salida: la salida en el instante `t` depende del estado alcanzado por la entrada del instante anterior (`t-1`). En notación, si en el instante `t` la máquina está en estado `q_t`, entonces la salida es `s_t = g(q_t)`. Y dado que `q_t` es resultado de la transición desde el estado anterior con la entrada anterior (`q_t = f(q_{t-1}, e_{t-1})`), se cumple `s_t = g(f(q_{t-1}, e_{t-1}))`. En otras palabras, la salida actual corresponde a la respuesta a la entrada previa.

Toda máquina de Moore puede convertirse en una de Mealy equivalente en comportamiento, y viceversa. Convertir de Mealy a Moore requiere un poco más de trabajo: básicamente, en la Mealy original se deben crear estados desglosados para cada combinación que producía salidas distintas con una misma configuración, asegurando que en Moore cada estado único produzca consistentemente un único símbolo de salida. En cambio, pasar de Moore a Mealy es más sencillo: se puede definir una Mealy cuyos estados coincidan con los de Moore y que, para cada transición, emita el símbolo de salida asociado al estado de partida (así Mealy replicará el "retardo" de Moore). En términos generales, **Mealy y Moore son equivalentes en poder de expresión**: cualquier secuencia de entradas/salidas producida por una puede ser generada por la otra mediante un diseño apropiado, aunque sus estructuras de estado pueden diferir.

(Nota: Las máquinas secuenciales, al operar continuamente, no tenían condiciones de aceptación ni finalización en su definición original. Sin embargo, en aplicaciones prácticas a veces se define un estado o condición final cuando se modela un proceso finito. En este curso, el foco luego se trasladará a autómatas con estados de aceptación para reconocer lenguajes, como se detalla a continuación.*)*

Autómatas Finitos Deterministas (AFD)

Cuando a una máquina secuencial (por ejemplo de Mealy) se le agrega la noción de **estado inicial** y de **estados de aceptación**, obtenemos un **Autómata Finito Determinista**. En su forma más general, un AFD puede seguir considerándose traductor (produce salidas mientras procesa la entrada) y tiene un inicio y fin bien definidos. La definición formal completa de un **AFD traductor** es una **séptupla**:

- **AFDT = (ΣE, ΣS, Q, q_0, A, f, g)**

Donde: - ΣE: alfabeto de símbolos de **entrada** (conjunto de símbolos que el autómata reconoce). - ΣS: alfabeto de símbolos de **salida** (posibles símbolos que puede emitir).
- Q: conjunto finito (y no vacío) de **estados** posibles. - q_0: **estado inicial**, $q_0 \in Q$, desde el cual el autómata comienza su operación. - A: conjunto de **estados de aceptación** ($A \subseteq Q$). Son los estados considerados **aceptadores** al finalizar la lectura. - f: Q × ΣE → Q es la función de **transición** (igual que en la máquina secuencial, determina el siguiente estado dado el estado actual y el símbolo leído). - g: Q × ΣE → ΣS es la función de **salida** (en caso de un AFD traductor, puede definirse igual que en Mealy, generando una salida por cada transición).

Debido a que ahora hay un estado inicial y la cadena de entrada es **finita**, un AFDT siempre completa su operación en un número finito de pasos (tras leer todos los símbolos de la entrada). Por lo tanto, cada ejecución del autómata es finita y el proceso de cómputo que realiza es un algoritmo bien definido.

En muchos casos, estamos interesados únicamente en que el autómata **reconozca o valide** ciertas cadenas (lenguaje) y no en producir una salida específica. En esas situaciones, el alfabeto de salida ΣS y la función g resultan innecesarios. Podemos *eliminarlos* de la tupla, obteniendo el **Autómata Finito Determinista Reconocedor**, definido simplemente como una **quíntupla**:

- **AFDR = (ΣE, Q, q_0, A, f)**

Los componentes restantes tienen el mismo significado que antes. Comúnmente, en adelante, nos referiremos al AFDR simplemente como **AFD**, dado que nuestro foco estará en la capacidad de reconocimiento. Un AFD entonces inicia en el estado q_0 y, tras leer por completo una cadena de entrada, determinará si la termina en un estado de aceptación (lo que implica que la cadena es **aceptada**) o en un estado no aceptor (cadena **rechazada**).

Determinismo: La clave de un **autómata determinista** es que su función de transición **f es una función propiamente dicha**, no solo una relación. Esto significa que **para cada posible par (estado, símbolo)** de entrada, está definido *a lo sumo un* próximo estado. En otras palabras, dado un estado actual y un símbolo leído, el siguiente estado queda *unívocamente determinado*. No puede haber dos transiciones distintas activas con la misma combinación de estado y símbolo. Esta propiedad garantiza que, para una cadena de entrada dada, el camino de cómputo del autómata está completamente determinado (no hay bifurcaciones ni elecciones aleatorias en el recorrido).

- *Función completa vs parcial:* La función de transición f de un AFD puede ser **total (completa)** o **parcial**. Se dice que f es *total* cuando está definida para **todos** los pares (estado, símbolo) en Q × ΣE. En ese caso, el autómata tiene especificado un próximo estado para cualquier símbolo que pueda leer en cualquier estado en el que se encuentre. Si, por el contrario, existe al menos un par (estado, símbolo) para el cual f **no** está definido, la función es *parcial*. Esto significa que en ciertas condiciones el autómata **no tiene transición** definida (no sabe a qué estado ir). En la práctica, esas condiciones no previstas representan situaciones de **falla** o rechazo inmediato: si la máquina lee un símbolo para el cual no hay transición desde su estado actual, se interpreta que la máquina **se detiene** porque la cadena no es válida en ese contexto. Para efectos teóricos, a menudo se puede introducir un **estado sink (de error)** no aceptor para absorber cualquier transición no definida originalmente, convirtiendo así f en una función completa sin cambiar el lenguaje reconocido (todas las transiciones que antes eran "vacío" ahora llevan a este estado de error, que es no aceptor y con bucles a sí mismo). En adelante, consideraremos que los AFD pueden presentarse tanto con funciones completas como parciales; en cualquier caso, conceptualmente siempre es posible completarlos añadiendo un estado de error cuando haga falta.

Ejemplos prácticos (AFD reconocedores):
- *Ejemplo:* Un caso sencillo es construir AFD que reconozcan cadenas según la **paridad** de cierta condición. Por ejemplo, con ΣE = {1}:
- Un AFD puede reconocer las cadenas que contienen un **número impar** de símbolos 1. Este AFD tendría dos estados básicos: un estado inicial/aceptor que representa "cantidad de 1 leídos es par (0 al inicio, que consideramos par)", y otro estado (no aceptor inicial) que representa "cantidad de 1 es impar". Cada vez que lee un 1, el autómata conmute entre estos dos estados. De este modo, solo terminará en el estado aceptor si ha leído 1 una cantidad impar de veces.
- De forma similar, se diseña otro AFD que reconoce las cadenas con **número par** de 1. En este caso el

estado inicial sería aceptor (0 es par) y permanecería o alternaría de estado con cada nuevo `1` leído de manera acorde.
(Estos corresponden al Ejercicio 2 propuesto: incisos a y b para número impar y par de unos, respectivamente. Son AFD muy simples, con 2 estados cada uno.)
- *Ejemplo:* En el **Ejemplo 3.3** del apunte se presenta un AFD diseñado para reconocer cadenas de bits que **terminan en "1000"**. ΣE = {0,1}, y el lenguaje reconocido corresponde a la expresión regular `(0+1)*1000` (cadenas binarias que tienen *1000* como sufijo). El autómata construido posee 5 estados, incluyendo uno inicial y uno de aceptación (en este caso el estado que indica que se han leído los cuatro bits finales "1000"). A modo de prueba, se analizan dos cadenas: `1011000` (que **sí** termina en 1000) y `1011100` (que **no** cumple con el sufijo). El AFD recorre una secuencia de estados distinta para cada una: para `1011000` consigue llegar a su estado de aceptación al agotar la entrada, mientras que con `1011100` termina en un estado no aceptor. (Ver más abajo cómo se puede comprobar el recorrido con herramientas gráficas).
- *Ejercicio sugerido:* Diseñar un **AFD** que reconozca un cierto patrón sobre {a, b}. Por ejemplo (Ejercicio 5 propuesto), un AFD que **excluya** de su lenguaje todas las cadenas que contengan las subcadenas `aa` o `bb`. Esto implica que el autómata debe rechazar cualquier entrada donde aparezcan dos símbolos iguales consecutivos. Una estrategia para este diseño es crear estados que representen el último símbolo leído y vigilar si el siguiente es igual (en cuyo caso se va a un estado *trampa* de rechazo). No hay longitud máxima predefinida para las palabras aceptadas, por lo que el autómata debe manejar secuencias de cualquier tamaño cumpliendo la restricción.

Operación y Lenguaje Reconocido por un AFD

Una **configuración** de un AFD en un momento dado puede describirse como `(` estado actual `, cadena_pendiente )`. Inicialmente, la configuración es `(`q_0, α`)`, donde q_0 es el estado inicial y α es la cadena de entrada completa por leer. El proceso de cómputo consiste en una secuencia de **movimientos** o transiciones elementales: si el autómata está en configuración (q, xα) (estado q con una entrada pendiente que comienza con símbolo x seguido del resto α), entonces tras un paso de transición irá a (q', α), donde q' = f(q, x) es el nuevo estado luego de leer x, y se ha consumido ese símbolo de la cadena. Formalmente, escribimos (q, xα) ⟶ (q', α)` para denotar este movimiento. Al aplicar sucesivamente las transiciones para cada símbolo de la entrada, eventualmente el autómata consume toda la cadena.

Se define la **función de transición extendida** `f*` (a veces denotada `f^e` o similar) para operar sobre cadenas completas. Esta función extendida toma un estado y una cadena `α` en ΣE* (secuencia de símbolos de entrada, incluyendo la posibilidad de cadena vacía λ) y retorna el estado en el que termina el autómata tras procesar toda esa cadena desde dicho estado. Puede definirse inductivamente: - Base: `f*(q, λ) = q` para cualquier estado `q` (si la cadena de entrada está vacía, el autómata no se mueve y permanece en el mismo estado). - Recursión: si `α = xβ` (primer símbolo `x` seguido de subcadena `β`), entonces `f*(q, α) = f*( f(q, x), β )`. Es decir, primero aplicamos una transición con `x` para obtener un nuevo estado, luego procesamos recursivamente el resto de la cadena desde ese estado.

En notación de configuraciones, podemos decir `(`q_0`, α) ⟶* (q_f, λ)` para indicar que a partir de la configuración inicial con cadena completa `α` el autómata realizó una serie (cero o más) de movimientos y terminó en el estado `q_f` habiendo consumido toda la entrada (λ restante). Si ese estado final `q_f` pertenece al conjunto A (estados de aceptación), entonces decimos que **acepta** la cadena `α`. Formalmente: una cadena `α` es **aceptada** si `f*(`q_0`, α) ∈ A`. Equivalente,

$(q_0, \alpha) \longrightarrow^* (q_f, \lambda)$ con $q_f \in A$. Si el estado final no es de aceptación, la cadena es **rechazada**.

Es importante destacar que para que un AFD *reconozca* (acepte/rechace) una cadena debe cumplir **dos condiciones simultáneas**: i) la lectura de la cadena de entrada se completa (toda la cadena es consumida), y ii) el estado alcanzado al final es un estado de aceptación. Si la cadena no se consume completamente (lo cual en un modelo formal ocurre solo si f es parcial en algún punto y se "detuvo" antes de tiempo por falta de transición) o si termina en un estado no aceptor, entonces la palabra no es aceptada. Un caso particular es la **cadena vacía** λ: un AFD la acepta si y solo si *su estado inicial es aceptador* (es decir, sin leer nada ya estamos en un estado de aceptación, $q_0 \in A$).

El conjunto de todas las cadenas que un autómata AFD acepta constituye un **lenguaje**. Si denotamos L(M) al lenguaje aceptado por la máquina M, se puede definir:

- **$L(M) = \{ \alpha \in \Sigma E \mid f(q_0, \alpha) \in A \}$**.

En otras palabras, L(M) es el conjunto de palabras sobre el alfabeto de entrada tales que el autómata, comenzando en su estado inicial, termina en un estado de aceptación después de leerlas completas. Equivalentemente, podemos decir que dos autómatas son diferentes si **aceptan lenguajes diferentes**.

Para **verificar el funcionamiento** de un AFD o ilustrar su comportamiento, a veces se utilizan herramientas gráficas: - Un **árbol de configuraciones** (o *árbol de descripciones instantáneas*) representa todas las posibles secuencias de configuraciones que el autómata puede tomar. En un AFD determinista, dado que no hay ramificaciones (no hay elección múltiple en las transiciones), el árbol de configuraciones para una cadena de entrada específica degenerará en una única rama lineal que muestra el recorrido estado por estado sobre los símbolos de esa cadena. - Un **plano de estados-entradas** es otra forma de representar la evolución: típicamente es una tabla bidimensional donde en una columna se escribe la cadena de entrada símbolo por símbolo, y en la fila correspondiente a cada estado se marca en qué posiciones el autómata estuvo en ese estado durante la lectura.

En el **Ejemplo 3.3** mencionado, después de definir el AFD formalmente, se presentan dos árboles de configuraciones que corresponden a procesar la cadena $\beta = 1011000$ (que sí posee el sufijo *1000*) y $\delta = 1011100$ (que no lo posee). En dichos árboles se observa que la rama para β finaliza en un estado de aceptación, mientras que la rama para δ finaliza en un estado no aceptor. Dado que el autómata es determinista, estos "árboles" en realidad son caminos únicos (uno por palabra analizada). Asimismo, el ejemplo incluye un diagrama plano estado-entrada para la cadena aceptada β=1011000, donde se ve la secuencia de estados $p \to q \to r \to \ldots \to t$ alineada con cada posición de la cadena hasta terminar en el estado t (que era el único estado de aceptación A = {t} definido). Estas herramientas de análisis ayudan a entender y *probar* la corrección del autómata en cuanto a las palabras que debe aceptar o rechazar.

Propiedades de los AFD

Dado un AFD, podemos estudiar algunas propiedades importantes de su estructura y de su relación con el lenguaje que reconoce:

- **Accesibilidad de estados:** Un estado $p \in Q$ se dice **accesible** desde otro estado q si existe alguna cadena de entrada α que lleve al autómata desde q hasta p. Es decir, p es alcanzable en la estructura de transiciones partiendo de q. En símbolos: $p\ Aq \iff \exists\ \alpha \in \Sigma E^*$ tal que $f^*(q, \alpha) = p$. En particular, un estado es accesible desde el estado inicial q_0 si el

autómata puede llegar a ese estado leyendo alguna secuencia desde q_0. Si un estado no es accesible desde q_0, **no influye** en el comportamiento reconocedor del autómata (porque nunca será visitado durante la lectura de ninguna cadena válida). Por lo tanto, **cualquier estado inaccesible puede eliminarse** de la definición del AFD sin cambiar el lenguaje reconocido. Se dice que un AFD es **conexo** si *todos* sus estados son accesibles desde el inicial. En símbolos: AFD conexo $\Leftrightarrow \forall\ p \in Q,\ p\ A\ q_0$. Antes de analizar o minimizar un autómata, suele ser conveniente remover estados no accesibles para trabajar solo con la parte "útil" (conexa) del mismo.
Ejemplo: En el **Ejemplo 3.7** del material se da la función de transición de un AFD con 8 estados {p1,...,p8}. Al representarlo gráficamente, se observa que uno de ellos (p7) **no es accesible** desde el estado inicial p1. Esto significa que el autómata **no es conexo**. En la solución, se señalan con líneas punteadas el estado p7 y sus transiciones, indicando que pueden eliminarse sin afectar al lenguaje que el autómata reconoce (Figura 3.14 del apunte). Tras quitar ese estado aislado, el autómata resultante queda conexo.

- **Equivalencia de estados:** Dos estados $p, q \in Q$ de un AFD se definen **equivalentes** (denotado $p \equiv q$ o $p\ E\ q$) si, *a partir de ellos*, el autómata acepta exactamente las **mismas** cadenas de entrada y rechaza exactamente las mismas. Es decir, no existe ninguna cadena que llevando al autómata desde p termine en aceptación y desde q termine en rechazo, o viceversa. En términos formales:
$p \equiv q \iff \forall\ \alpha \in \Sigma E^*, [f^*(p, \alpha) \in A \Leftrightarrow f^*(q, \alpha) \in A]$.
Esta relación $\equiv$ cumple las propiedades de una **relación de equivalencia** sobre el conjunto de estados Q (es reflexiva, simétrica y transitiva), por lo que divide (o **particiona**) a Q en clases de estados indistinguibles entre sí respecto del lenguaje reconocido. Si dos estados son equivalentes, **no hay forma de distinguirlos mediante ninguna continuación de entrada**, pues desde ambos el autómata "responde" igual (acepta o rechaza las mismas extensiones de cadena). En consecuencia, para propósitos de reconocimiento, tales estados redundan en la representación de un mismo comportamiento. (Cabe notar que si el autómata tiene más de un estado de aceptación, la definición permite que dos estados equivalentes bien puedan ser distintos estados de aceptación o incluso uno de aceptación y otro de rechazo, siempre que *desde ellos* cualquier cadena eventual termine concordando en aceptación/rechazo; no es necesario que las cadenas terminen exactamente en el mismo estado, solo importa si es aceptor o no). Existe también la noción de **equivalencia de longitud *k***: se dice que p y q son equivalentes de longitud k (denotado $p \equiv_k q$) si ambas producen los mismos resultados de aceptación/rechazo para *todas* las cadenas de entrada de longitud hasta k. A medida que k crece, la equivalencia de longitud k va refinando esta distinción hasta k suficientemente grande donde alcanza la equivalencia total (∞). Estas ideas son útiles para algoritmos de minimización, donde se comienza asumiendo equivalentes aquellos estados solo diferenciables por mirar una cierta longitud de cadenas y luego incrementando ese límite.

- **Equivalencia de autómatas:** Dos AFD (digamos M1 y M2, con posiblemente distintos conjuntos de estados) se consideran **equivalentes** si **reconocen el mismo lenguaje**, es decir, si *aceptan exactamente las mismas palabras*. En otras palabras, L(M1) = L(M2). Equivalentemente, sus estados iniciales $q0\_M1$ y $q0\_M2$ son estados equivalentes en el sentido arriba definido (sus comportamientos futuros son indistinguibles). Si dos autómatas son equivalentes, uno puede sustituirse por el otro en cuanto a reconocimiento se refiere. Esta propiedad es importante al comparar soluciones o al optimizar un diseño de autómata.

Minimización de Autómatas Finitos

Dado un AFD, muchas veces ocurre que fue diseñado con más estados de los estrictamente necesarios para reconocer su lenguaje. Estados adicionales pueden haber sido útiles durante el *diseño* para representar claramente condiciones intermedias, pero una vez construido el autómata, podría haber estados redundantes (equivalentes entre sí). Un **autómata mínimo** para un lenguaje L es aquel que tiene la menor cantidad posible de estados entre todos los AFD que reconocen L. Todo lenguaje regular (reconocible por algún AFD) tiene un autómata finito determinista mínimo **único** (a excepción del renombrado de estados). Obtener este autómata mínimo es deseable para optimizar la eficiencia.

Un AFD **conexo** que tenga más estados que los necesarios *debe* tener al menos un par de estados equivalentes (indistinguibles) según la definición anterior. La **minimización** consiste en identificar esos estados redundantes y **unificarlos** (colapsarlos) en uno solo, obteniendo un nuevo autómata más pequeño pero que acepta el mismo lenguaje. El método sistemático para minimizar un AFD se basa en construir el **conjunto cociente** Q/≡ de estados, es decir, las clases de equivalencia de estados según la relación `≡` de indistinguibilidad.

Procedimiento general de minimización (por particiones sucesivas): Se parte de una **partición inicial** de Q separando los estados **aceptadores** de los **no aceptadores**, ya que claramente cualquier estado de aceptación no puede ser equivalente a uno de rechazo (difieren en la cadena vacía, por ejemplo). Esta partición inicial son las clases de equivalencia considerando cadenas de longitud k=0. Luego, iterativamente, se *refinan* estas clases comprobando si dentro de una misma clase hay estados que se comportan diferente frente a *algún símbolo de entrada*. En concreto, para cada clase y para cada símbolo `a ∈ ΣE`, se verifica a qué clase lleva la transición: si dos estados `p` y `q` estaban en la misma clase pero existe algún símbolo `a` tal que `f(p,a)` y `f(q,a)` caen en **clases distintas** en la partición actual, entonces `p` y `q` **no** son equivalentes (porque ante la entrada `a` sus futuros difieren) y por tanto esa clase debe subdividirse separando a `p` y `q` en distintas subclases. Este refinamiento se repite sucesivamente (clases cada vez más finas, equivalencia para longitudes k crecientes) hasta que no haya cambios, es decir, hasta obtener dos particiones consecutivas iguales. En ese punto, cada clase resultante representa un conjunto de estados originalmente equivalentes e indistinguibles.

Finalmente, se construye el **autómata mínimo** cuyos estados son precisamente esas clases de equivalencia. El estado inicial del nuevo autómata es la clase que contenía al antiguo estado inicial, y los estados de aceptación son aquellas clases que contienen al menos un estado de aceptación original. La función de transición nueva se define inducida por las transiciones originales: si desde un estado `p` en la clase X con símbolo `a` se iba a `f(p,a) = q` que pertenece a la clase Y, entonces en el autómata minimizado definimos que desde el estado-X con `a` se va al estado-Y.

Ejemplos:
- *Ejemplo:* En el **Ejemplo 3.5** del apunte se diseñó un AFD relativamente grande (7 estados) para reconocer el lenguaje de *"todas las cadenas sobre {a,b} que terminan en* `ba` *"*. Este AFD fue construido de manera explícita para contemplar *todas* las combinaciones posibles de sufijos durante la lectura (asegurando que reconoce correctamente cualquier string que termine en "ba"). Luego se plantea la pregunta: ¿es posible un AFD más pequeño que reconozca el mismo patrón? Aplicando el algoritmo de **minimización**, se descubren estados equivalentes que pueden fusionarse. En efecto, tras separar los estados de aceptación del resto e iterar refinamientos, se obtiene un conjunto cociente con menos clases (menos estados) que el original. El autómata mínimo equivalente tiene únicamente aquellos estados necesarios para recordar el sufijo de longitud 2 requerido. (En este ejemplo particular, el autómata mínimo termina teniendo 3 estados esenciales, correspondientes a "no ha leído nada o algo que no aporta al sufijo `ba` todavía", "ha leído una `b` al final parcial" y "ha logrado leer `ba` al final";

cualquier otro estado intermedio del diseño original resultó indistinguible de alguno de estos).
- *Ejercicio sugerido:* Practicar la minimización con algún autómata dado. Por ejemplo, en el **Ejercicio 10** se proporcionan dos AFD distintos y se pide **comprobar si son equivalentes**. Para ello, se debe minimizar cada uno (eliminando primero estados no conexos si los hubiera) y luego comparar los autómatas mínimos obtenidos. Si ambos mínimos resultan idénticos (isomorfos), significará que los AFD originales reconocían el mismo lenguaje. En caso contrario, alguna diferencia en los lenguajes aparecerá durante la minimización. Como otro ejercicio, se podría tomar un AFD previamente diseñado (por ejemplo, el del ejercicio 5 que filtraba subcadenas prohibidas, u otro de los ejercicios de diseño) y aplicar el algoritmo de particiones para reducirlo a su versión mínima, verificando que el lenguaje aceptado permanece igual.

Autómatas Finitos Bidireccionales

Un **Autómata Finito Determinista Bidireccional (AFDB)** es una variante del AFD en la cual el cabezal de lectura de la cinta de entrada puede moverse **hacia la derecha (→)** *o* **hacia la izquierda (←)** en cada transición, en lugar de avanzar únicamente hacia adelante. Formalmente, esto implica que la función de transición f podría definirse con un comportamiento adicional que indique el movimiento del cabezal en cada paso (por ejemplo `f: Q × ΣE → Q × {L,R}`), o equivalentemente, que algunas transiciones en el grafo del autómata van con dirección de lectura inversa. En la práctica, se suele asumir que el autómata tiene **marcas de fin** en los extremos de la cinta (para evitar salirse de los límites de la entrada) o alguna convención similar.

Los AFDB **no reconocen nuevos lenguajes** más allá de los que reconoce un AFD tradicional: su capacidad computacional es la misma (siguen aceptando exactamente los **lenguajes regulares**). Sin embargo, ofrecer movimiento bidireccional puede hacer más sencillo el diseño de autómatas para ciertos patrones, ya que el autómata puede "revisitar" partes de la entrada para verificar alguna condición global que de otra forma requeriría más memoria interna (más estados). **Importante:** Dado que el autómata puede moverse a voluntad en la cinta, es posible que entre en un **ciclo infinito** (por ejemplo, yendo hacia la izquierda y derecha alternativamente sin nunca alcanzar el final de la cadena). Por eso, al diseñar un AFDB se debe asegurar que *siempre terminará* la lectura de la cadena en un número finito de pasos (por ejemplo, evitando bucles de movimiento inútiles o detectando condiciones de repetición).

El modelo de AFDB es principalmente de interés teórico. En problemas prácticos, en general, cualquier AFDB puede simularse mediante algún AFD equivalente (aunque posiblemente con más estados). Aun así, entender la bidireccionalidad es útil para conectar con modelos más potentes: conceptualmente, si permitimos cinta infinita y capacidad de escritura además de movimiento bidireccional, pasamos al terreno de las máquinas de Turing.

Ejemplos típicos:
- *Ejercicio sugerido:* Construir un **AFD bidireccional** que reconozca los **números binarios pares** (Ejercicio 12 propuesto). Recordemos que un número binario es par si su último dígito es 0. Con un AFD normal (una dirección) esto es trivial de reconocer leyendo el último símbolo, pero imaginemos usar un AFDB: una posible estrategia sería que el autómata se desplace primero hacia la **derecha** a lo largo de la cinta hasta encontrar el final de la cadena, luego retroceda un paso hacia la **izquierda** para leer el último dígito, y finalmente decida la aceptación según si dicho dígito es `0` o `1`. Este recorrido bidireccional aprovecha la capacidad de releer la cinta para moverse directamente al final y comprobar la condición requerida. (Nótese que, aunque útil para ilustrar el concepto, en este caso la ventaja no es significativa ya que un AFD convencional puede resolverlo con solo 2 estados y sin necesidad de moverse hacia atrás).

- *Ejercicio sugerido:* Diseñar un AFDB que acepte cadenas en **código Morse** (alfabeto Σ = {·, –}) que **comienzan y terminan con el mismo símbolo** (Ejercicio 13 del material). Por ejemplo, `··–·` no cumpliría porque inicia con `·` y termina con `·`? (aquí ambos son `·`, de hecho este sí cumple), otro que no cumpla sería `·–·–` (empieza con `·` y termina con `–`, diferente). Para resolver esto con un autómata de una sola dirección sería complejo porque requeriría "recordar" el primer símbolo hasta llegar al final de la cadena para compararlo con el último. En cambio, un enfoque bidireccional es muy directo: el AFDB puede **leer el primer símbolo** de la cadena (por ejemplo, entra al estado `q_dot` si es `·` o a `q_dash` si es `–`), luego **moverse hacia la derecha** leyendo sin tomar acciones especiales hasta encontrar el final de la secuencia, retroceder un paso para situarse sobre el último símbolo y **compararlo** con el primero (esto podría implicar tener diferentes estados finales de aceptación: uno que acepte si estaba en modo "esperando último igual a ·" y encuentra ·, u otro si esperaba – y encuentra –). Si coincide, acepta; si no, rechaza. Este ejemplo ilustra cómo la lectura hacia atrás permite verificar una relación entre el primer y último carácter de la cadena de manera más sencilla conceptualmente. (De nuevo, cualquier lenguaje que un AFDB pueda reconocer también lo puede reconocer algún AFD normal; de hecho este mismo lenguaje "palabras que empiezan y terminan igual" sobre un alfabeto finito es regular y un AFD unidireccional puede construirse, aunque con más estados para recordar el símbolo inicial hasta el final).

En resumen, los **autómatas finitos deterministas**, tanto en su versión sencilla como bidireccional, constituyen un concepto fundamental en la teoría de lenguajes formales. Hemos repasado sus definiciones, variantes, la forma en que operan, cómo reconocer lenguajes con ellos, sus propiedades estructurales (accesibilidad, equivalencias) y el procedimiento de minimización para optimizarlos. Cada tema ha sido acompañado de ejemplos de aplicación y ejercicios sugeridos, que sirven como guía de estudio para afianzar estos conceptos de cara a los exámenes finales de la materia. Así, **Unidad 3** proporciona las bases para entender cómo modelar y analizar comportamientos secuenciales y lenguajes regulares mediante máquinas de estados finitos deterministas.
