Universidad Nacional de la Patagonia San Juan Bosco
Facultad de Ingeniería
Licenciatura en Informática
Fundamentos teóricos de la informática

Trabajo Práctico Integrador

“Eliminación de no determinismo y minimización de autómatas finitos”

Alumno: Bonansea Camaño, Mariano Nicolás
marianobonanseapetrovial@gmail.com
Cátedra: Leonardo Moreno - Leonardo Morales
Fecha: 26/04/2025

Introducción

En el marco del estudio de los lenguajes formales, los **autómatas finitos** constituyen un modelo fundamental para describir el comportamiento de los **lenguajes regulares**. Su importancia radica en que permiten representar de manera formal sistemas capaces de procesar y reconocer cadenas de símbolos.

No obstante, un mismo lenguaje puede expresarse a través de diferentes autómatas, algunos de ellos con comportamientos **no deterministas** y otros con más estados de los estrictamente necesarios. Esto plantea la necesidad de aplicar dos procesos básicos:

1. La **conversión de un autómata finito no determinista (AFND) a un autómata finito determinista (AFD)**, de modo que el modelo pueda ser ejecutado de manera unívoca.

2. La **minimización** del AFD obtenido, con el fin de simplificar su estructura sin modificar el lenguaje aceptado.

El objetivo de este trabajo es implementar un programa que lleve a cabo dichos procesos, permita describir un autómata en un formato estandarizado, lo transforme en un AFD mínimo equivalente y ofrezca además la posibilidad de validar cadenas de entrada sobre las distintas representaciones generadas.

Fundamentos teóricos

- **Autómata Finito No Determinista (AFND):**
  Es un modelo matemático definido por una quíntupla $(Q, \Sigma, \delta, q_0, F)$ donde:
  - Q es el conjunto finito de estados.
  - Σ es el alfabeto de entrada.
  - δ es la función de transición, que a cada par (q, a) puede asociar un subconjunto de estados de Q.
  - q_0 es el estado inicial.
  - F es el conjunto de estados finales.

  En este tipo de autómatas, una misma configuración puede dar lugar a múltiples caminos de ejecución o incluso ninguno.

- **Autómata Finito Determinista (AFD):**
  También se expresa mediante una quíntupla $(Q, \Sigma, \delta, q_0, F)$, pero en este caso la función de transición δ es total y unívoca:
  - Cada par (q, a) conduce a un único estado. De esta manera, cada cadena de entrada corresponde exactamente a una computación posible.

- **Autómata Mínimo:**
  Es un AFD que reconoce el mismo lenguaje que otro AFD dado, pero con la menor cantidad de estados posible. Se obtiene a través de algoritmos de minimización que identifican y unifican estados equivalentes, es decir, aquellos que no pueden distinguirse en términos del lenguaje aceptado.

Algoritmo de Conversión AFND → AFD

Filtrado de estados útiles en el AFND:

- Se identifican y conservan únicamente los estados que pueden alcanzar algún estado final, eliminando desde el inicio los estados sumidero o inalcanzables.
- Se construye la tabla de transiciones del AFND solo con estos estados útiles, aplicando clausura epsilon para cada transición.

Construcción de la tabla del AFD (algoritmo de subconjuntos):

- El estado inicial del AFD es la clausura epsilon del estado inicial del AFND:

```
estado_inicial_conjunto = afnd.clausura_epsilon({afnd.estado_inicial})

estado_inicial_nombre = self._obtener_nombre_estado(estado_inicial_conjunto)

por_procesar = [estado_inicial_conjunto]
```

- Se utiliza una cola para iterar sobre conjuntos de estados AFND, generando nuevos estados AFD:

```
while por_procesar:

    conjunto_actual = por_procesar.pop(0)

    conjunto_key = frozenset(conjunto_actual)

    if conjunto_key in self.estados_procesados:

        continue

    ...

    self.estados_procesados.add(conjunto_key)

    nombre_estado = self._obtener_nombre_estado(conjunto_actual)

    self.tabla_afd[nombre_estado] = {}
```

- Para cada símbolo del alfabeto, se calcula el conjunto destino aplicando las transiciones y la clausura epsilon:

```
for simbolo in sorted(afnd.alfabeto):

    if simbolo == '':

        continue

    conjunto_destino = set()

    for estado_afnd in conjunto_actual:

        if estado_afnd in self.tabla_afnd and simbolo in self.tabla_afnd[estado_afnd]:

            conjunto_destino.update(self.tabla_afnd[estado_afnd][simbolo])
```

- Antes de agregar una transición, se verifica si el conjunto destino puede alcanzar estados finales:

```
if conjunto_destino:
    if self._puede_alcanzar_final(conjunto_destino, afnd):
        nombre_destino = self._obtener_nombre_estado(conjunto_destino)
        self.tabla_afd[nombre_estado][simbolo] = nombre_destino
        ...
        if (conjunto_destino_key not in self.estados_procesados and
                conjunto_destino not in por_procesar):
            por_procesar.append(conjunto_destino)
    else:
        self.tabla_afd[nombre_estado][simbolo] = None
```

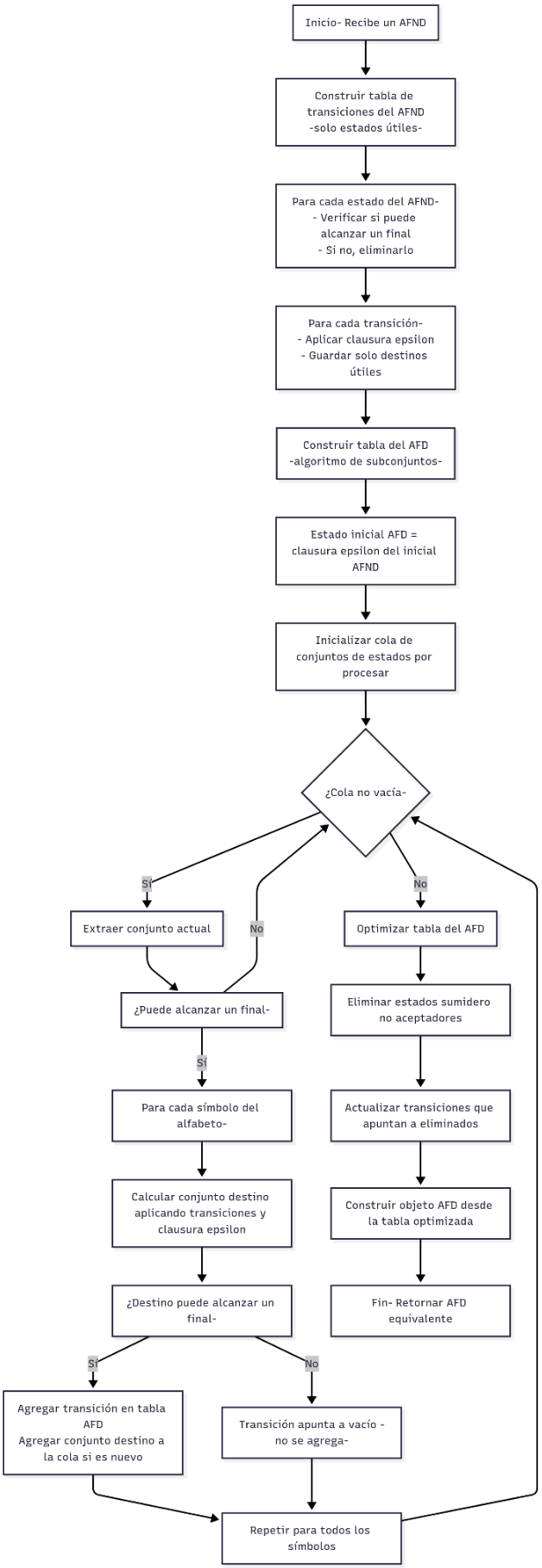
- Se eliminan estados sumidero no aceptadores y se actualizan las transiciones que apuntan a ellos, evitando transiciones innecesarias.

Construcción final del AFD:

- Se crea el AFD con los estados, transiciones, estado inicial y estados finales obtenidos de la tabla optimizada:

```
afd = AFD(
    estados=estados_afd,
    alfabeto=afnd.alfabeto - {''},
    estado_inicial=estado_inicial_afd,
    estados_finales=estados_finales_afd,
    transiciones=transiciones_afd,
    descripcion=f"AFD convertido desde AFND - {getattr(afnd, 'descripcion', '')}"
)
```

Diagrama de flujo Algoritmo de Conversión AFND → AFD



Pseudocódigo de flujo Algoritmo de Conversión AFND → AFD

```
ALGORITMO Convertir_AFND_a_AFD(afnd):
  // Reiniciar estructuras internas
  tabla_afnd ← {}
  tabla_afd ← {}
  mapeo_estados ← {}
  estados_procesados ← ∅
  historial ← []
  // Paso 1: Construir tabla del AFND solo con estados útiles
  estados_utiles ← ∅
  PARA cada estado EN afnd.estados HACER
    SI PuedeAlcanzarFinal({estado}, afnd) ENTONCES
      estados_utiles ← estados_utiles ∪ {estado}
  FIN PARA
  PARA cada estado EN estados_utiles HACER
    PARA cada simbolo EN afnd.alfabeto SIN '' HACER
      destinos ← destinos de (estado, simbolo) en afnd.transiciones
      destinos_con_epsilon ← ∅
      PARA cada destino EN destinos HACER
        clausura ← ClausuraEpsilon({destino})
        destinos_con_epsilon ← destinos_con_epsilon ∪ (clausura ∩ estados_utiles)
      FIN PARA
      tabla_afnd[estado][simbolo] ← destinos_con_epsilon
    FIN PARA
  FIN PARA
  // Paso 2: Generar tabla del AFD (algoritmo de subconjuntos)
  estado_inicial_conjunto ← ClausuraEpsilon({afnd.estado_inicial})
  por_procesar ← [estado_inicial_conjunto]
  estados_procesados ← ∅
  MIENTRAS por_procesar NO esté vacío HACER
    conjunto_actual ← por_procesar.pop(0)
```

```
    SI conjunto_actual YA está en estados_procesados ENTONCES
      CONTINUAR
    FIN SI
    SI NO PuedeAlcanzarFinal(conjunto_actual, afnd) ENTONCES
      CONTINUAR
    FIN SI
    estados_procesados ← estados_procesados U {conjunto_actual}
    nombre_estado ← ObtenerNombreEstado(conjunto_actual)
    tabla_afd[nombre_estado] ← {}
    PARA cada simbolo EN afnd.alfabeto SIN '' HACER
      conjunto_destino ← ∅
      PARA cada estado_afnd EN conjunto_actual HACER
        conjunto_destino ← conjunto_destino U tabla_afnd[estado_afnd][simbolo]
      FIN PARA
      SI conjunto_destino ≠ ∅ Y PuedeAlcanzarFinal(conjunto_destino, afnd) ENTONCES
        nombre_destino ← ObtenerNombreEstado(conjunto_destino)
        tabla_afd[nombre_estado][simbolo] ← nombre_destino
        SI conjunto_destino NO está en estados_procesados NI en por_procesar ENTONCES
          por_procesar.append(conjunto_destino)
        FIN SI
      SINO
        tabla_afd[nombre_estado][simbolo] ← None
      FIN SI
    FIN PARA
  FIN MIENTRAS
  // Paso 3: Optimizar tabla del AFD eliminando estados sumidero no aceptadores
  estados_finales_afd ← CalcularEstadosFinalesAFD(tabla_afd, afnd)
  estados_a_eliminar ← ∅
  PARA cada estado EN tabla_afd HACER
    SI estado NO es final Y todas sus transiciones son None o a sí mismo ENTONCES
      estados_a_eliminar ← estados_a_eliminar U {estado}
```

```
        FIN SI
    FIN PARA


    PARA cada estado EN estados_a_eliminar HACER
      Eliminar estado de tabla_afd
    FIN PARA
    PARA cada estado EN tabla_afd HACER
      PARA cada simbolo EN tabla_afd[estado] HACER
        SI tabla_afd[estado][simbolo] EN estados_a_eliminar ENTONCES
          tabla_afd[estado][simbolo] ← None
        FIN SI
      FIN PARA
    FIN PARA
    // Paso 4: Construir y retornar el objeto AFD
    estados_afd ← claves de tabla_afd
    estado_inicial_afd ← ObtenerNombreEstado(ClausuraEpsilon({afnd.estado_inicial}))
        estados_finales_afd ← {estado | estado ∈ estados_afd Y conjunto_afnd(estado) ∩ afnd.estados_finales ≠ ∅}
    transiciones_afd ← {(estado, simbolo): destino | destino ≠ None en tabla_afd}
      retornar NuevoAFD(estados_afd, afnd.alfabeto - {''}, estado_inicial_afd, estados_finales_afd, transiciones_afd)
```

Algoritmo de Minimización de AFD

Eliminación de estados inalcanzables

- Se identifican los estados alcanzables desde el estado inicial y se eliminan los que no lo son:

```
estados_alcanzables = afd.obtener_estados_alcanzables()

afd_sin_inalcanzables = self._eliminar_estados_inalcanzables(afd, estados_alcanzables)
```

Partición inicial (finales vs no finales)

- Se agrupan los estados en dos conjuntos: finales y no finales:

```
estados_finales = set(afd_sin_inalcanzables.estados_finales)
estados_no_finales = afd_sin_inalcanzables.estados - estados_finales
particion_actual = []
if estados_no_finales:
    particion_actual.append(estados_no_finales)
if estados_finales:
    particion_actual.append(estados_finales)
```

Refinamiento de particiones

- Se repite el proceso de subdivisión de grupos hasta que no haya cambios, comparando las "firmas" de transiciones de cada estado:

```
while True:
    nueva_particion = self._refinar_particion(afd_sin_inalcanzables, particion_actual)
                        if len(nueva_particion) == len(particion_actual) and
self._particiones_equivalentes(particion_actual, nueva_particion):
        break
particion_actual = nueva_particion
```

- Cada grupo se divide según el destino de sus transiciones para cada símbolo del alfabeto:

```
for estado in grupo:
    firma = []
    for simbolo in sorted(afd.alfabeto):
        if (estado, simbolo) in afd.transiciones:
            destino = afd.transiciones[(estado, simbolo)]
            particion_destino = self._encontrar_grupo_de_estado(destino, particion_actual)
            firma.append(particion_destino)
        else:
            firma.append(-1)
    firmas.setdefault(tuple(firma), set()).add(estado)
```

Construcción del AFD minimizado

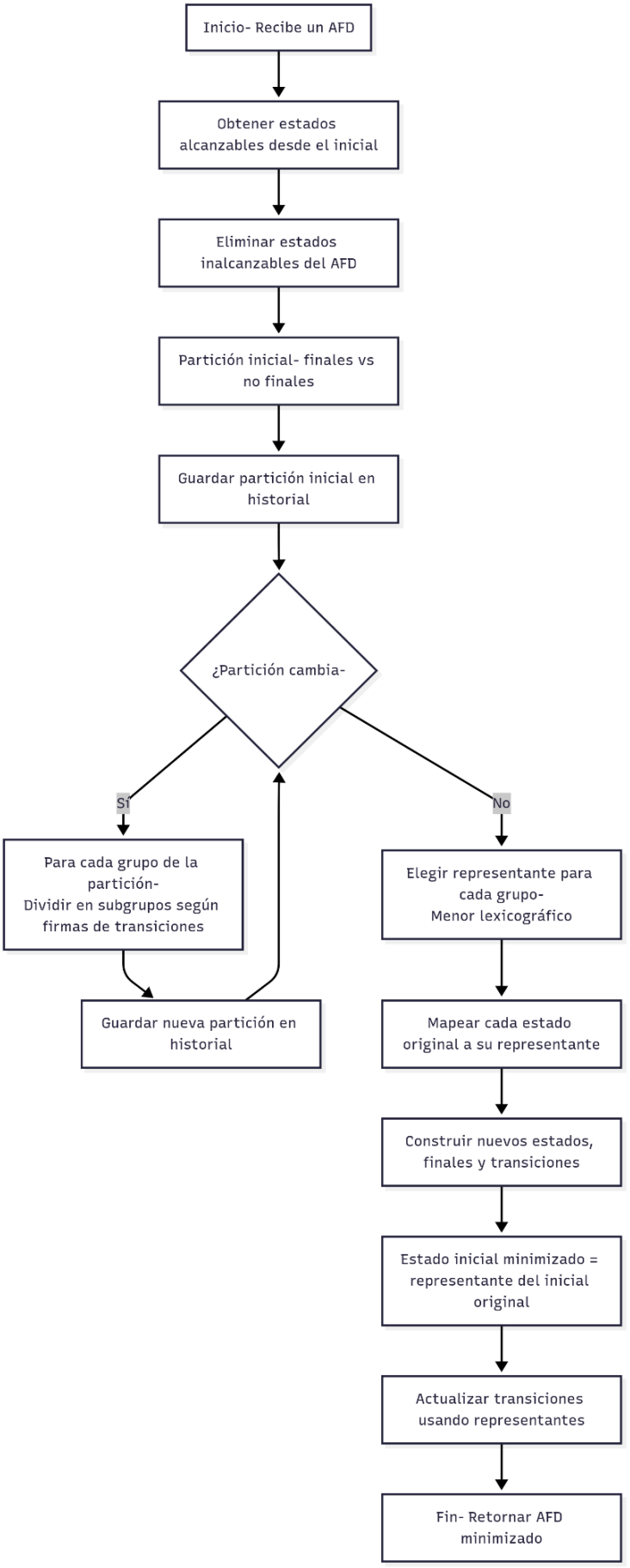
- Cada grupo final de la partición se convierte en un estado del nuevo AFD, eligiendo un representante para cada grupo:

```
for i, grupo in enumerate(particion_final):
    representante = min(grupo, key=str)
    for estado in grupo:
            mapeo_estados[estado] = representante
```

- Las transiciones y los estados finales se actualizan en función de los representantes:

```
for (origen, simbolo), destino in afd_original.transiciones.items():
    if origen in mapeo_estados and destino in mapeo_estados:
        origen_min = mapeo_estados[origen]
        destino_min = mapeo_estados[destino]
        transiciones_min[(origen_min, simbolo)] = destino_min
```

Diagrama de flujo de Algoritmo de Minimización de AFD



Pseudocódigo de Algoritmo de Minimización de AFD

```
ALGORITMO Minimizar_AFD(afd):
  // Paso 1: Eliminar estados inalcanzables
  estados_alcanzables ← obtener_estados_alcanzables(afd)
  afd ← eliminar_estados_inalcanzables(afd, estados_alcanzables)
  // Paso 2: Partición inicial (finales vs no finales)
  finales ← afd.estados_finales
  no_finales ← afd.estados - finales
  particion ← []
  SI no_finales ≠ ∅ ENTONCES particion.agregar(no_finales)
  SI finales ≠ ∅ ENTONCES particion.agregar(finales)
  // Paso 3: Refinar partición hasta que no cambie
  repetir
    nueva_particion ← []
    PARA cada grupo EN particion HACER
      subgrupos ← dividir_grupo_por_firma(afd, grupo, particion)
      nueva_particion.agregar_todos(subgrupos)
    FIN PARA
    SI particion_equivalente(particion, nueva_particion) ENTONCES
      salir del bucle
    FIN SI
    particion ← nueva_particion
  hasta que no haya cambios
  // Paso 4: Construir AFD minimizado
  mapeo_estados ← {}
  estados_min ← ∅
  PARA cada grupo EN particion HACER
    representante ← menor_lexicografico(grupo)
    estados_min.agregar(representante)
    PARA cada estado EN grupo HACER
```

```
            mapeo_estados[estado] ← representante
        FIN PARA
    FIN PARA
    estado_inicial_min ← mapeo_estados[afd.estado_inicial]
    finales_min ← {mapeo_estados[e] | e ∈ afd.estados_finales}
    transiciones_min ← {}
    PARA cada (origen, simbolo) → destino EN afd.transiciones HACER
        SI origen y destino en mapeo_estados ENTONCES
            origen_min ← mapeo_estados[origen]
            destino_min ← mapeo_estados[destino]
            transiciones_min[(origen_min, simbolo)] ← destino_min
        FIN SI
    FIN PARA
    retornar NuevoAFD(estados_min, afd.alfabeto, estado_inicial_min, finales_min, transiciones_min)
```

Validación de cadenas en autómatas

Funcionamiento en AFD (Determinista)

- El método recorre la cadena símbolo por símbolo, actualizando el estado actual según las transiciones definidas:

```
estado_actual = self.estado_inicial
for simbolo in cadena:
    if simbolo not in self.alfabeto:
        return False
    if (estado_actual, simbolo) in self.transiciones:
        estado_actual = self.transiciones[(estado_actual, simbolo)]
    else:
        return False
    return estado_actual in self.estados_finales
```

  - Si algún símbolo no está en el alfabeto, la cadena se rechaza.
  - Si falta una transición, la cadena se rechaza.
  - Se acepta si el estado final es aceptador.

Funcionamiento en AFND (No Determinista)

- El método mantiene un conjunto de estados actuales y explora todas las transiciones posibles para cada símbolo:

```
estados_actuales = {self.estado_inicial}
for simbolo in cadena:
    if simbolo not in self.alfabeto:
        return False
    nuevos_estados = set()
    for estado in estados_actuales:
        if (estado, simbolo) in self.transiciones:
            destinos = self.transiciones[(estado, simbolo)]
            if isinstance(destinos, set):
                nuevos_estados.update(destinos)
            else:
                nuevos_estados.add(destinos)
    estados_actuales = nuevos_estados
    if not estados_actuales:
        return False
return bool(estados_actuales.intersection(self.estados_finales))
```

  - Se exploran todos los caminos posibles simultáneamente.
  - Si en algún paso no hay estados alcanzables, la cadena se rechaza.
  - Se acepta si al menos un estado final es alcanzado.

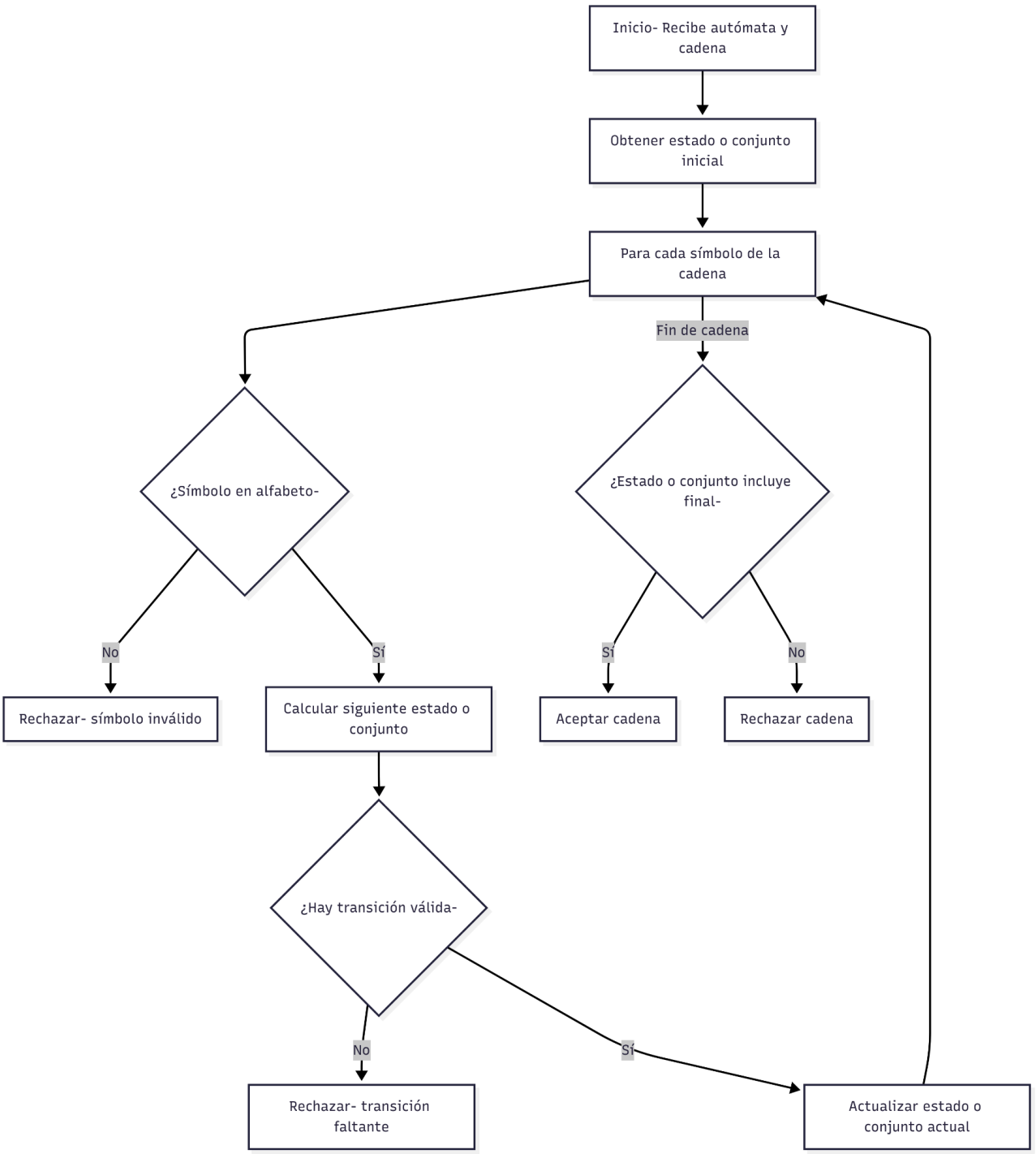
Manejo de Clausura Epsilon

- Para AFND, se calcula la clausura epsilon cuando es necesario, permitiendo alcanzar estados adicionales mediante transiciones vacías:

```
def clausura_epsilon(self, estados):
    clausura = set(estados)
    por_procesar = list(estados)
    while por_procesar:
        estado = por_procesar.pop()
        if (estado, '') in self.transiciones:
            destinos = self.transiciones[(estado, '')]
            if isinstance(destinos, set):
                for destino in destinos:
                    if destino not in clausura:
                        clausura.add(destino)
                        por_procesar.append(destino)
            else:
                if destinos not in clausura:
                    clausura.add(destinos)
                    por_procesar.append(destinos)
return clausura
```

- Se expande el conjunto de estados actuales con todos los alcanzables por transiciones epsilon.

Diagrama de flujo de Validación de cadenas en autómatas



Pseudocódigo de Validación de cadenas en autómatas

```
ALGORITMO Validar_Cadena(automata, cadena):

  SI automata es AFD ENTONCES
    estado_actual ← automata.estado_inicial

    PARA cada simbolo EN cadena HACER
      SI simbolo NO está en automata.alfabeto ENTONCES
        retornar False  // símbolo inválido
      FIN SI

      SI (estado_actual, simbolo) EN automata.transiciones ENTONCES
        estado_actual ← automata.transiciones[(estado_actual, simbolo)]
      SINO
        retornar False  // transición faltante
      FIN SI
    FIN PARA

    retornar estado_actual EN automata.estados_finales

  SINO SI automata es AFND ENTONCES
    estados_actuales ← {automata.estado_inicial}

    PARA cada simbolo EN cadena HACER
      SI simbolo NO está en automata.alfabeto ENTONCES
        retornar False  // símbolo inválido
      FIN SI

      nuevos_estados ← ∅
      PARA cada estado EN estados_actuales HACER
        SI (estado, simbolo) EN automata.transiciones ENTONCES
          destinos ← automata.transiciones[(estado, simbolo)]
          SI destinos es conjunto ENTONCES
            nuevos_estados ← nuevos_estados U destinos
          SINO
            nuevos_estados ← nuevos_estados U {destinos}
          FIN SI
        FIN SI
      FIN PARA

      estados_actuales ← nuevos_estados

      SI estados_actuales es vacío ENTONCES
        retornar False  // no hay caminos posibles
      FIN SI
    FIN PARA

    retornar estados_actuales ∩ automata.estados_finales ≠ ∅

  FIN SI
```

Decisiones de diseño

- **Modularidad y organización:** El programa se estructuró en módulos independientes para cada funcionalidad principal: conversión, minimización, validación, graficación y utilidades. Esto facilita el mantenimiento, la extensión y la reutilización del código.
- **Soporte de múltiples formatos:** Se eligió soportar tanto archivos JSON como texto plano para la entrada y salida de autómatas, permitiendo flexibilidad y compatibilidad con distintas herramientas.
- **Validación automática de equivalencia:** Tras cada conversión y minimización, el sistema valida automáticamente que el autómata resultante sea equivalente al original usando el método formal del autómata producto, garantizando la corrección de las transformaciones.
- **Interfaz interactiva y por lotes:** Se implementó una interfaz de usuario que permite validar cadenas de forma interactiva, individual o masiva, facilitando la comprobación rápida y flexible de resultados.
- **Reportes y visualización:** Se generan reportes detallados y gráficos visuales de los autómatas, ayudando a la interpretación y análisis de los resultados.
- **Gestión de archivos temporales:** El sistema maneja y elimina archivos temporales automáticamente para evitar residuos y mantener el entorno limpio.
- **Robustez y experiencia de usuario:** Se priorizó la captura de errores, mensajes claros y la posibilidad de volver al menú de selección sin reiniciar el programa, mejorando la experiencia de uso.

Casos de prueba

Dentro del contenido del trabajo se incluyo una carpeta de ejemplos, la cual contiene múltiples casos de autómatas para ser probados por la aplicación. A continuación se detalla el procedimiento de instalación y funcionamiento completo a partir de uno de los casos de ejemplo incluidos.

Instalación y configuraciones iniciales

1. Clonar el repositorio
2. Crear un entorno virtual con python: python -m venv .venv
3. Activar el entorno virtual:
    - Windows: .venv\Scripts\activate
    - Linux/Mac: source .venv/bin/activate
4. Instalar las dependencias: pip install -r requirements.txt
5. Instalar Graphviz (para las opciones de representación gráfica):
    - Windows: Descargar desde graphviz.org y agregar al PATH
    - Linux: sudo apt-get install graphviz (Ubuntu/Debian) o sudo yum install graphviz (CentOS/RHEL)
    - macOS: brew install graphviz

Uso

- **Comandos disponibles:**

  python main.py <archivo> [directorio_salida] [opciones]

- **Argumentos posicionales:**

  <archivo>: Archivo del autómata (JSON o texto)

  [directorio_salida]: Directorio de salida para los resultados (por defecto: resultados)

- **Opciones principales:**

  -o, --output DIR Directorio de salida para los resultados (sobrescribe el posicional)

- **Opciones de procesamiento (excluyentes):**

  -c, --convertir Solo convertir AFND a AFD (sin minimizar)

  -m, --minimizar Solo minimizar el AFD (el archivo debe ser un AFD)

  -v, --validar Modo interactivo para validar cadenas

  -s, --string CADENA Validar una cadena específica

--validar-archivo ARCHIVO_JSON Validar múltiples cadenas desde un archivo JSON con el autómata especificado

- **Opciones de graficación:**

  -g, --graficar Generar gráficos del autómata y del proceso

  --solo-graficar Solo grafica el autómata proporcionado, sin procesamiento adicional

  -f, --formatos FORMATO(S) Formatos de gráficos separados por comas (png,pdf,svg) (por defecto: png)

- **Opciones de utilidad:**

  -h, --help Muestra la ayuda y sintaxis de uso

  --verificar-graphviz Verificar la instalación de Graphviz y salir

  -r, --no-reportes No generar reportes detallados

  --verbose Mostrar información detallada del proceso

  --version Muestra el número de versión del programa y sale

- **Ejemplos de uso**
  - Procesamiento completo (conversión + minimización)

    python main.py ejemplos/TP1_Ej9a.json resultados/
  - Solo conversión AFND → AFD

    python main.py ejemplos/TP1_Ej9a.json -c
  - Solo minimización de AFD

    python main.py ejemplos/TP1_Ej9a.json -m
  - Generar gráficos en varios formatos

    python main.py ejemplos/TP1_Ej9a.json -g -f png,pdf,svg
  - Validación interactiva de cadenas

    python main.py ejemplos/TP1_Ej9a.json -v
  - Validar una cadena específica

    python main.py ejemplos/TP1_Ej9a.json -s "abba"
  - Validar múltiples cadenas desde archivo JSON

    python main.py ejemplos/TP1_Ej9a.json --validar-archivo ejemplos/cadenas_prueba.json
  - Solo generar gráficos, sin procesamiento

```
python main.py ejemplos/TP1_Ej9a.json --solo-graficar
```

Ejemplo concreto del proceso completo (conversión, minimización y validación):

Tomando como archivo de ejemplo a minimizar a TP1_Ej9a.json o TP1_Ej9a.txt que tienen los siguientes formatos necesarios para el funcionamiento:

TP1_Ej9a.json:

```
{
    "estados": ["S0", "S1", "S2", "S3", "S4", "S5"],
    "alfabeto": ["a", "b"],
    "estado_inicial": "S0",
    "estados_finales": ["S3", "S4"],
    "transiciones": [
        {"origen": "S0", "simbolo": "a", "destino": "S1"},
        {"origen": "S0", "simbolo": "a", "destino": "S2"},
        {"origen": "S0", "simbolo": "b", "destino": "S5"},
        {"origen": "S1", "simbolo": "a", "destino": "S3"},
        {"origen": "S1", "simbolo": "b", "destino": "S1"},
        {"origen": "S2", "simbolo": "a", "destino": "S2"},
        {"origen": "S2", "simbolo": "b", "destino": "S4"},
        {"origen": "S3", "simbolo": "a", "destino": "S5"},
        {"origen": "S3", "simbolo": "b", "destino": "S3"},
        {"origen": "S4", "simbolo": "a", "destino": "S5"},
        {"origen": "S4", "simbolo": "b", "destino": "S5"},
        {"origen": "S5", "simbolo": "a", "destino": "S5"},
        {"origen": "S5", "simbolo": "b", "destino": "S5"}
    ],
    "descripcion": "AFND del ejercicio 9a del TP1"
}
```

TP1_Ej9a.txt:

```
ESTADOS: S0,S1,S2,S3,S4,S5
ALFABETO: a,b
ESTADO_INICIAL: S0
ESTADOS_FINALES: S3,S4
DESCRIPCION: AFND del ejercicio 9a del TP1
```

TRANSICIONES:

S0,a,S1

S0,a,S2

S0,b,S5

S1,a,S3

S1,b,S1

S2,a,S2

S2,b,S4

S3,a,S5

S3,b,S3

S4,a,S5

S4,b,S5

S5,a,S5

S5,b,S5

- **Nota:** el campo descripcion/DESCRIPCION es opcional y aporta una descripción del lenguaje que acepta el autómata o descripción del mismo autómata para facilitar su entendimiento.

Luego se procede a realizar el proceso completo (conversión y minimización graficada con validación interactiva de cadenas) mediante el siguiente comando (desde la raíz del proyecto):

```
python main.py ./ejemplos/TP1_Ej9a.json -g
```

Posteriormente se realizaran todos los procesos necesarios para la conversión, minimización y validación de los resultados mostrando los siguientes mensajes de log:

```
$ python main.py ./ejemplos/TP1_Ej9a.json -g

===== PROCESADOR DE AUTÓMATAS FINITOS =====

📂Cargando autómata desde: ./ejemplos/TP1_Ej9a.json
ℹ️Formato detectado: json
🔀Autómata cargado: AFND
🔵Estados: 6
➡️Transiciones: 12
🎨Graficador inicializado correctamente
🔄Convirtiendo AFND a AFD...
✅ Conversión completada: 9 estados
🎨Gráfico de conversión: resultados\conversion_afnd_afd.png
⚡ Minimizando AFD...
🎯Minimización completada: 8 estados
🎨Gráfico de minimización: resultados\minimizacion_afd.png
📉Reducción total: -2 estados (-33.3%)
💾Guardando resultados...
📄Guardado: resultados\automata_afd.json
📄Guardado: resultados\automata_minimizado.json
📋Generando reportes detallados...
✅ Reportes generados: 2 archivos
📋   resultados\reporte_conversion.txt
📋   resultados\reporte_minimizacion.txt
✅ Los autómatas son equivalentes
```

Finalmente, se le preguntará al usuario si desea validar cadenas interactivamente para el autómata en cualquiera de sus 3 estados (original/convertido/minimizado):

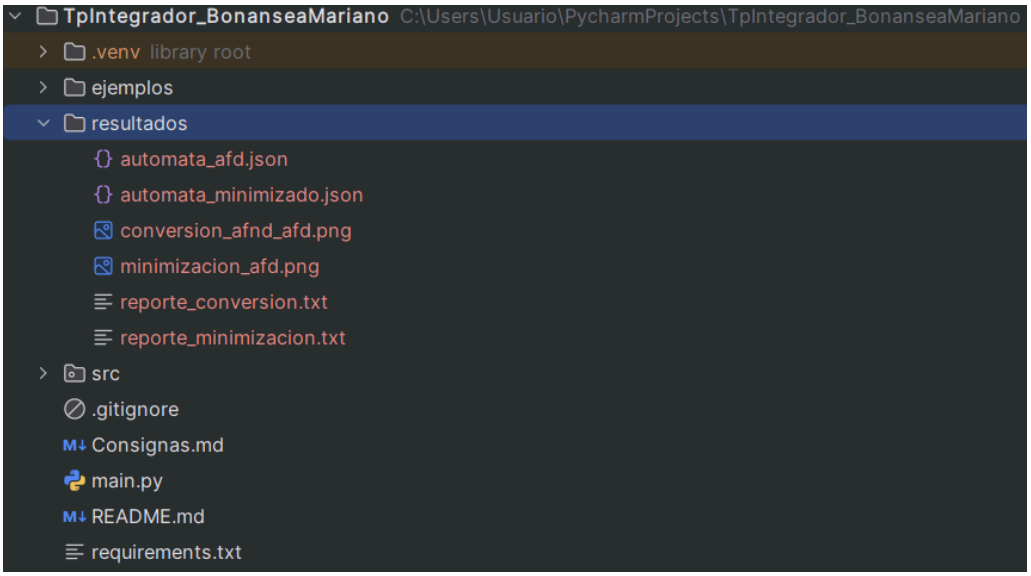
```
===== PROCESAMIENTO COMPLETADO =====


¿Deseas validar cadenas interactivamente? (s/N): s
Elige qué autómata usar:
1. Original
2. AFD (después de conversión)
3. Minimizado
Opción (1-3): 3
ℹ️ Formato detectado: json

===== VALIDADOR DE CADENAS =====

⚙️ Autómata: AFD
🔵 Estados: 8 (S0, S1, S2, S3, S4, {S1,S2}, {S1,S4}, {S2,S3})
🔤 Alfabeto: {'b', 'a'}
⚫ Estado inicial: S0
🔴 Estados finales: S3, S4, {S1,S4}, {S2,S3}
ℹ️ Descripción: AFD convertido desde AFND - AFND del ejercicio 9a del TP1
ℹ️
🔗 Ingresa cadenas para validar (o 'salir' para volver al menú):

▶ Cadena: ab
✅ 'ab' -> ACEPTADA

▶ Cadena: ba
❌ 'ba' -> RECHAZADA

▶ Cadena: salir
ℹ️ Volviendo al menú de selección de autómata...
Elige qué autómata usar:
1. Original
2. AFD (después de conversión)
3. Minimizado
```

El proceso de validación interactiva puede interrumpirse colocando la cadena “salir” para volver al menú de selección de autómata o con la combinación de teclas “ctrl+c” para terminar la ejecución del programa.

```
🔗 Ingresa cadenas para validar (o 'salir' para volver al menú):

▶ Cadena: ab
✅ 'ab' -> ACEPTADA

▶ Cadena: ba
❌ 'ba' -> RECHAZADA

▶ Cadena: ℹ️
✅ Saliendo...

Operación cancelada
```

Nota: Solo se detalla el proceso principal, las distintos procesos pueden realizarse individualmente o con ciertos parámetros como se detallo anteriormente en la sección de uso.

Una vez finalizada la validación interactiva de cadenas se podrá observar que se generó un directorio “resultados” en la raíz (ya que no se especificó una ruta de salida) que contiene los productos resultantes de los procedimientos:



Detalle de los archivos resultantes:

- **automata_afd.json:** contiene el autómata resultante de la eliminacion de no determinismo.


```
{
 "estados": [
  "{S1,S2}",
  "S1",
  "{S3,S4}",
  "{S1,S4}",
  "S0",
  "S4",
  "S3",
  "S2",
  "{S2,S3}"
 ],
 "alfabeto": [
  "b",
  "a"
 ],
 "estado_inicial": "S0",
```

```
"estados_finales": [
  "S4",
  "{S3,S4}",
  "S3",
  "{S2,S3}",
  "{S1,S4}"
],
"transiciones": [
  {
    "origen": "S0",
    "simbolo": "a",
    "destino": "{S1,S2}"
  },
  {
    "origen": "{S1,S2}",
    "simbolo": "a",
    "destino": "{S2,S3}"
  },
  {
    "origen": "{S1,S2}",
    "simbolo": "b",
    "destino": "{S1,S4}"
  },
  {
    "origen": "{S2,S3}",
    "simbolo": "a",
    "destino": "S2"
  },
  {
    "origen": "{S2,S3}",
    "simbolo": "b",
    "destino": "{S3,S4}"
  },
```

```
{
  "origen": "{S1,S4}",
  "simbolo": "a",
  "destino": "S3"
},
{
  "origen": "{S1,S4}",
  "simbolo": "b",
  "destino": "S1"
},
{
  "origen": "S2",
  "simbolo": "a",
  "destino": "S2"
},
{
  "origen": "S2",
  "simbolo": "b",
  "destino": "S4"
},
{
  "origen": "{S3,S4}",
  "simbolo": "b",
  "destino": "S3"
},
{
  "origen": "S3",
  "simbolo": "b",
  "destino": "S3"
},
{
  "origen": "S1",
  "simbolo": "a",
```

```
      "destino": "S3"
    },
    {
      "origen": "S1",
      "simbolo": "b",
      "destino": "S1"
    }
  ],
  "descripcion": "AFD convertido desde AFND - AFND del ejercicio 9a del TP1",
  "num_estados": 9,
  "num_transiciones": 13
}
```


- **automata_minimizado.json:** contiene el autómata resultante de la minimización del AFD construido previamente.


```
{
  "estados": [
    "{S1,S2}",
    "S1",
    "{S1,S4}",
    "S0",
    "S4",
    "S3",
    "S2",
    "{S2,S3}"
  ],
  "alfabeto": [
    "b",
    "a"
  ],
  "estado_inicial": "S0",
  "estados_finales": [
    "S4",
    "{S2,S3}",
```

```
    "{S1,S4}",
    "S3"
  ],
  "transiciones": [
    {
      "origen": "S0",
      "simbolo": "a",
      "destino": "{S1,S2}"
    },
    {
      "origen": "{S1,S2}",
      "simbolo": "a",
      "destino": "{S2,S3}"
    },
    {
      "origen": "{S1,S2}",
      "simbolo": "b",
      "destino": "{S1,S4}"
    },
    {
      "origen": "{S2,S3}",
      "simbolo": "a",
      "destino": "S2"
    },
    {
      "origen": "{S2,S3}",
      "simbolo": "b",
      "destino": "S3"
    },
    {
      "origen": "{S1,S4}",
      "simbolo": "a",
      "destino": "S3"
```

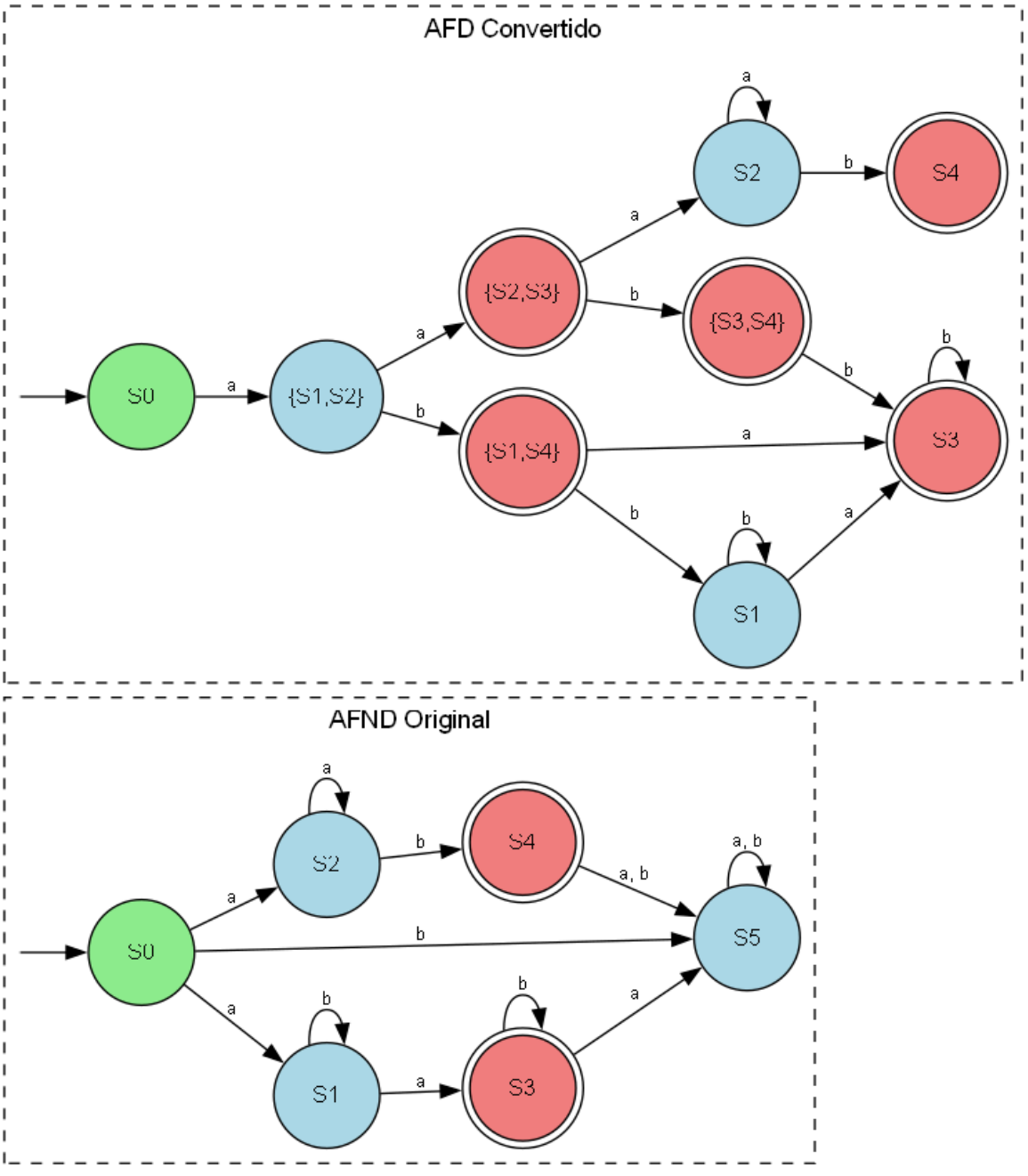
```
    },
    {
      "origen": "{S1,S4}",
      "simbolo": "b",
      "destino": "S1"
    },
    {
      "origen": "S2",
      "simbolo": "a",
      "destino": "S2"
    },
    {
      "origen": "S2",
      "simbolo": "b",
      "destino": "S4"
    },
    {
      "origen": "S3",
      "simbolo": "b",
      "destino": "S3"
    },
    {
      "origen": "S1",
      "simbolo": "a",
      "destino": "S3"
    },
    {
      "origen": "S1",
      "simbolo": "b",
      "destino": "S1"
    }
  ],
  "descripcion": "AFD convertido desde AFND - AFND del ejercicio 9a del TP1",
```

"num_estados": 8,

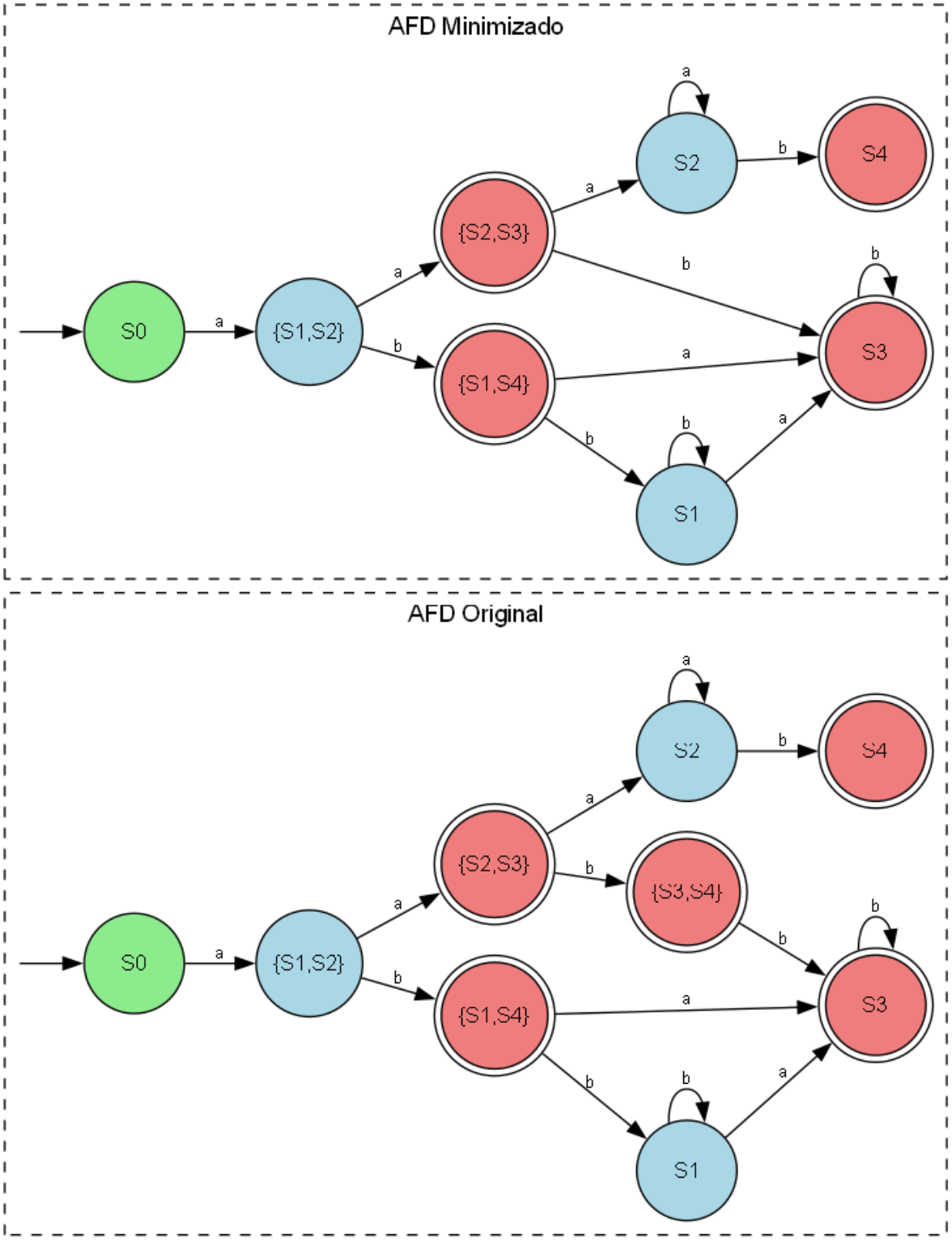
"num_transiciones": 12

}

- **conversion_afnd_afd.png:** contiene la representación gráfica del autómata original y la de la eliminacion del no determinismo (ya que se indico el parámetro -g pero sin formato en específico).



- **minimizacion_afd.png:** contiene la representación gráfica del afd anterior y la del autómata minimizado (ya que se indico el parámetro -g pero sin formato en específico).



- **reporte_conversion.txt:** contiene el reporte del proceso de eliminación del no determinismo incluyendo una tabla de transiciones del autómata original y otra del AFD construido.

```
======================================================================
REPORTE DE ELIMINACIÓN DE NO DETERMINISMO
======================================================================
RESUMEN:
 • Estados AFND: 6
 • Estados AFD: 9
PROCESO DETALLADO:
--------------------------------------------------
=== TABLA DE TRANSICIONES ORIGINAL SIN ESTADOS SUMIDERO ===
+----+---------+------+---+
| δ  |    a    |  b   | F |
+----+---------+------+---+
| S0 | {S1,S2} |  {}  | 0 |
| S1 |  {S3}   | {S1} | 0 |
| S2 |  {S2}   | {S4} | 0 |
| S3 |   {}    | {S3} | 1 |
| S4 |   {}    |  {}  | 1 |
+----+---------+------+---+
Estados sumidero eliminados desde el AFND: ['S5']
=== GENERACIÓN DE TABLA AFD ===
Estado inicial AFD: S0 = ['S0']
Procesando estado S0 = ['S0']
 δ(S0, a) = {S1,S2}
Procesando estado {S1,S2} = ['S1', 'S2']
 δ({S1,S2}, a) = {S2,S3}
 δ({S1,S2}, b) = {S1,S4}
Procesando estado {S2,S3} = ['S2', 'S3']
 δ({S2,S3}, a) = S2
 δ({S2,S3}, b) = {S3,S4}
Procesando estado {S1,S4} = ['S1', 'S4']
 δ({S1,S4}, a) = S3
 δ({S1,S4}, b) = S1
```

```
Procesando estado S2 = ['S2']
 δ(S2, a) = S2
 δ(S2, b) = S4
Procesando estado {S3,S4} = ['S3', 'S4']
 δ({S3,S4}, b) = S3
Procesando estado S3 = ['S3']
 δ(S3, b) = S3
Procesando estado S1 = ['S1']
 δ(S1, a) = S3
 δ(S1, b) = S1
Procesando estado S4 = ['S4']
=== TABLA DE TRANSICIONES DEL AFD CONSTRUIDO ===
+---------+---------+---------+---+
|    δ    |    a    |    b    | F |
+---------+---------+---------+---+
|   S0    | {S1,S2} |    -    | 0 |
|   S1    |   S3    |   S1    | 0 |
|   S2    |   S2    |   S4    | 0 |
|   S3    |    -    |   S3    | 1 |
|   S4    |    -    |    -    | 1 |
| {S1,S2} | {S2,S3} | {S1,S4} | 0 |
| {S1,S4} |   S3    |   S1    | 1 |
| {S2,S3} |   S2    | {S3,S4} | 1 |
| {S3,S4} |    -    |   S3    | 1 |
+---------+---------+---------+---+
```

- **reporte_minimizacion.txt:** contiene las tablas de transiciones del AFD anterior y la del AFD minimizado incluyendo un apartado de estados equivalentes y bajo que nombre figuran en la tabla del último.

```
======================================================================

REPORTE DE MINIMIZACIÓN

======================================================================

AUTÓMATA ORIGINAL:

+---------+---------+---------+---+

|    δ    |    a    |    b    | F |

+---------+---------+---------+---+

|   S0    | {S1,S2} |    -    | 0 |

|   S1    |   S3    |   S1    | 0 |

|   S2    |   S2    |   S4    | 0 |

|   S3    |    -    |   S3    | 1 |

|   S4    |    -    |    -    | 1 |

| {S1,S2} | {S2,S3} | {S1,S4} | 0 |

| {S1,S4} |   S3    |   S1    | 1 |

| {S2,S3} |   S2    | {S3,S4} | 1 |

| {S3,S4} |    -    |   S3    | 1 |

+---------+---------+---------+---+

AUTÓMATA MINIMIZADO:

+---------+---------+---------+---+

|    δ    |    a    |    b    | F |

+---------+---------+---------+---+

|   S0    | {S1,S2} |    -    | 0 |

|   S1    |   S3    |   S1    | 0 |

|   S2    |   S2    |   S4    | 0 |

|   S3    |    -    |   S3    | 1 |

|   S4    |    -    |    -    | 1 |

| {S1,S2} | {S2,S3} | {S1,S4} | 0 |

| {S1,S4} |   S3    |   S1    | 1 |

| {S2,S3} |   S2    |   S3    | 1 |

+---------+---------+---------+---+

ESTADOS EQUIVALENTES:

 {S3, {S3,S4}} -> S3
```

Si se desea, también es posible validar un conjunto de cadenas de un archivo json a cualquiera de los autómatas generados y/o originales. El formato del archivo debe ser como el del archivo cadenas_prueba.json:

```
{
  "cadenas": [
    "",
    "a",
    "b",
    "aa",
    "ab",
    "ba",
    "bb",
    "aaa",
    "aab",
    "aba",
    "abb",
    "baa",
    "bab",
    "bba",
    "bbb",
    "aaaa",
    "aabb",
    "abab",
    "baba",
    "bbbb"
  ]
}
```

La validación de cadenas se puede realizar imitando el siguiente comando de ejemplo:

```
python main.py ./resultados/automata_minimizado.json --validar-archivo ./ejemplos/cadenas_prueba.json
```

```
$ python main.py ./resultados/automata_minimizado.json --validar-archivo ./ejemplos/cadenas_prueba.json
📂Cargando autómata desde: ./resultados/automata_minimizado.json
ℹ️Formato detectado: json

===== VALIDADOR DE CADENAS =====

⚙️Autómata: AFD
🔵Estados: 8 (S0, S1, S2, S3, S4, {S1,S2}, {S1,S4}, {S2,S3})
🔤Alfabeto: {'b', 'a'}
⚫Estado inicial: S0
🔴Estados finales: S3, S4, {S1,S4}, {S2,S3}
ℹ️Descripción: AFD convertido desde AFND - AFND del ejercicio 9a del TP1

===== VALIDACIÓN DE MÚLTIPLES CADENAS =====

📄Validando 20 cadenas desde: ./ejemplos/cadenas_prueba.json
ℹ️ 1. λ               → ❌ RECHAZADA
ℹ️ 2. 'a'             → ❌ RECHAZADA
ℹ️ 3. 'b'             → ❌ RECHAZADA
ℹ️ 4. 'aa'            → ✅ ACEPTADA
ℹ️ 5. 'ab'            → ✅ ACEPTADA
ℹ️ 6. 'ba'            → ❌ RECHAZADA
ℹ️ 7. 'bb'            → ❌ RECHAZADA
ℹ️ 8. 'aaa'           → ❌ RECHAZADA
ℹ️ 9. 'aab'           → ✅ ACEPTADA
ℹ️10. 'aba'           → ✅ ACEPTADA
ℹ️11. 'abb'           → ❌ RECHAZADA
ℹ️12. 'baa'           → ❌ RECHAZADA
ℹ️13. 'bab'           → ❌ RECHAZADA
ℹ️14. 'bba'           → ❌ RECHAZADA
ℹ️15. 'bbb'           → ❌ RECHAZADA
ℹ️16. 'aaaa'          → ❌ RECHAZADA
ℹ️17. 'aabb'          → ✅ ACEPTADA
ℹ️18. 'abab'          → ✅ ACEPTADA
ℹ️19. 'baba'          → ❌ RECHAZADA
ℹ️20. 'bbbb'          → ❌ RECHAZADA
```

Finalmente el sistema solicita si desea guardar el reporte de validación donde se le deberá indicar la ruta de destino:

```
¿Deseas guardar un reporte detallado? (s/N): s
Ingresa la ruta y nombre del archivo (sin extensión): ./resultados/validaciones
📄Reporte guardado: resultados\validaciones.txt
```

Verificación de equivalencia entre autómatas

Después de cada conversión (AFND → AFD) y minimización, el sistema valida automáticamente que el autómata resultante sea equivalente al original. Esta verificación se realiza de forma interna y se informa en el log del proceso.

Para realizar la validación:

- Se utiliza el método del autómata producto (desarrollado en src/utils/equivalencia.py).
- El procedimiento recorre en paralelo los estados de ambos AFD, comenzando desde sus estados iniciales.
- Para cada par de estados, se verifica si uno es final y el otro no. Si esto ocurre, se detecta una cadena que distingue ambos lenguajes y se concluye que no son equivalentes.
- Se exploran todas las transiciones posibles para cada símbolo del alfabeto, asegurando que no exista ninguna cadena que sea aceptada por uno y rechazada por el otro.
- Si no se encuentra ninguna diferencia, los autómatas se consideran equivalentes y el sistema lo informa automáticamente.

Esta validación automática garantiza que las operaciones de conversión y minimización no alteran el lenguaje aceptado por el autómata, brindando seguridad y robustez al proceso.

Conclusión

El trabajo realizado permitió profundizar en los conceptos fundamentales de la teoría de autómatas, abordando de manera práctica la conversión de autómatas no deterministas a deterministas, la minimización de autómatas y la validación de lenguajes reconocidos. A través de la implementación y experimentación, se evidenció la importancia de los algoritmos formales para garantizar que las transformaciones preserven el lenguaje aceptado, así como el valor de la verificación automática de equivalencia como herramienta para sustentar la corrección teórica de los procesos.

Eficiencia

- Los algoritmos implementados son correctos y adecuados para autómatas de tamaño pequeño a mediano, como los que suelen encontrarse en el ámbito académico.
- La validación de equivalencia garantiza resultados formales, pero su complejidad puede crecer rápidamente con el tamaño de los autómatas.
- El manejo de archivos y la generación de reportes y gráficos se realiza de forma eficiente y automatizada, permitiendo un flujo de trabajo ágil.

Limitaciones

- El rendimiento puede verse afectado con autómatas de gran tamaño, especialmente en la validación de equivalencia y en la conversión de AFND con muchos estados y transiciones epsilon.
- El sistema no incluye una interfaz gráfica avanzada, limitándose a la línea de comandos y a la generación de archivos de salida.
- La validación de cadenas se basa en la definición formal del autómata, pero no se realiza una generación automática de contraejemplos explícitos en caso de no equivalencia.
- El soporte de formatos de entrada está limitado a JSON y texto plano; no se incluye XML ni otros estándares.

Posibles mejoras

- Optimizar los algoritmos de equivalencia y minimización para manejar autómatas de mayor tamaño.
- Incorporar una interfaz gráfica de usuario (GUI) para facilitar la interacción y visualización de los resultados.
- Permitir la exportación e importación en otros formatos estándar (por ejemplo, JFLAP, XML, DOT).
- Implementar la generación automática de contraejemplos (cadenas que distinguen dos autómatas no equivalentes).
- Añadir más validaciones y sugerencias automáticas para la corrección de archivos de entrada.

El programa cumple con los objetivos propuestos y sienta una base sólida para futuras extensiones y mejoras, tanto en eficiencia como en usabilidad y robustez.

Bibliografía

- Material de cátedra de Fundamentos teóricos de la informática 2025 - UNPSJB.
- Fundamentos de ciencia de la computación - Juan Carlos Augusto.