Materia: Sistemas Expertos

Profesor: Mauricio Alejandro Cabrera Arellano

Grado y grupo: 7F

Alumno: Daniel Alejandro Flores Sepúlveda 21310203

Fecha de entrega: 01/09/2024

1. La Componente Humana

- **¿Qué?**: La componente humana en un sistema experto se refiere a las personas involucradas en el desarrollo, mantenimiento y uso del sistema. Incluye a los **expertos en la materia**, quienes poseen el conocimiento profundo y especializado en un dominio específico; los **ingenieros del conocimiento**, quienes se encargan de recopilar, codificar y estructurar el conocimiento de los expertos en la base de conocimiento del sistema; y los **usuarios finales**, quienes interactúan con el sistema para obtener soluciones o respuestas a problemas específicos.
- **¿Para qué?**: Esta componente es esencial para capturar y representar el conocimiento especializado que no se puede programar de manera simple o automatizar fácilmente. Los expertos aportan el contenido crítico que debe ser traducido en reglas o modelos que el sistema utilizará para hacer inferencias. Los ingenieros del conocimiento son responsables de transformar el conocimiento de los expertos en un formato que el sistema pueda utilizar, mientras que los usuarios finales aprovechan el sistema para obtener soporte en la toma de decisiones.
- **¿Cómo?**: Los expertos en la materia colaboran con los ingenieros del conocimiento a través de entrevistas, talleres, y sesiones de observación para transferir su experiencia y habilidades al sistema. Los ingenieros del conocimiento luego organizan y codifican esta información en la base de conocimiento utilizando técnicas como reglas de producción, árboles de decisión, marcos, o redes semánticas. Finalmente, los usuarios finales interactúan con el sistema a través de interfaces de usuario diseñadas para facilitar la entrada de datos, la formulación de consultas y la interpretación de las recomendaciones o soluciones proporcionadas por el sistema experto.

2. La Base de Conocimiento

- **¿Qué?**: La base de conocimiento es el componente central de un sistema experto que almacena toda la información relevante necesaria para la toma de decisiones o la resolución de problemas dentro de un dominio específico. Esta información puede incluir hechos, reglas, procedimientos, relaciones entre entidades, heurísticas, y modelos.
- **¿Para qué?**: Su propósito es proporcionar el "saber" del sistema, actuando como una fuente de información y lógica para el motor de inferencia, el cual utiliza esta base para generar conclusiones, realizar diagnósticos, o sugerir soluciones. Es crucial para que el sistema pueda replicar el proceso de razonamiento humano en situaciones complejas.
- **¿Cómo?**: La base de conocimiento se construye utilizando diversas técnicas de representación del conocimiento:
    - **Reglas de producción (si-entonces)**: Para capturar y representar conocimientos heurísticos o empíricos en forma de reglas lógicas.
    - **Marcos o "frames"**: Para estructurar el conocimiento en términos de objetos, atributos y valores asociados, permitiendo una representación más rica y contextual.

- **Redes semánticas**: Para representar relaciones entre conceptos o entidades en forma de grafos, facilitando la visualización de interdependencias y conexiones.
- **Ontologías**: Para estructurar el conocimiento de manera formal y estandarizada, facilitando la interoperabilidad con otros sistemas y bases de datos.
- **Hechos**: Datos específicos o instancias concretas del dominio, como síntomas de una enfermedad en un sistema de diagnóstico médico.

La base de conocimiento debe ser mantenida y actualizada regularmente para asegurar que el sistema experto refleje con precisión el estado actual del conocimiento en el dominio específico.

```
# Definimos los hechos (estado inicial del conocimiento)
hechos = {
    "esta_lloviendo": False,
    "nubes_grises": True,
    "lleva_paraguas": False
}


# Definimos las reglas (condiciones y acciones)
reglas = [
    {
        "condiciones": ["esta_lloviendo == True"],
        "accion": lambda hechos: "Deberías llevar un paraguas."
    },
    {
        "condiciones": ["nubes_grises == True", "esta_lloviendo == False"],
       "accion": lambda hechos: "Parece que podría llover. Mejor lleva un paraguas por si acaso."
    },
    {
        "condiciones": ["nubes_grises == False", "esta_lloviendo == False"],
        "accion": lambda hechos: "No necesitas un paraguas."
    }
]
def motor_inferencia(hechos, reglas):
```

```
    for regla in reglas:
        # Evaluar todas las condiciones en la regla
            condiciones_cumplidas = all(eval(condicion, {}, hechos) for condicion in regla["condiciones"])

        if condiciones_cumplidas:
            # Si todas las condiciones son ciertas, ejecuta la acción de la regla
            resultado = regla["accion"](hechos)
            return resultado

    return "No se encontró una regla aplicable."

# Ejecutar el sistema experto
resultado = motor_inferencia(hechos, reglas)
print(resultado)
```

3. Subsistema de Adquisición de Conocimiento

- **¿Qué?**: El subsistema de adquisición de conocimiento es el componente de un sistema experto encargado de recopilar, organizar, y actualizar el conocimiento que se almacenará en la base de conocimiento. Este subsistema facilita la transferencia del conocimiento de los expertos humanos o fuentes externas hacia el sistema experto.
- **¿Para qué?**: Su propósito es asegurar que la base de conocimiento esté siempre actualizada, precisa y completa, reflejando el conocimiento más reciente y relevante del dominio. Este subsistema permite que el sistema experto evolucione con el tiempo, incorporando nuevas reglas, hechos, procedimientos, y datos a medida que cambian las circunstancias o se descubren nuevos conocimientos.
- **¿Cómo?**: El subsistema de adquisición de conocimiento emplea varias técnicas para capturar y estructurar el conocimiento:
    - **Entrevistas y talleres con expertos**: Interacciones directas con expertos en la materia para extraer su conocimiento a través de preguntas específicas y ejercicios prácticos.
    - **Observación y documentación**: Análisis de las actividades diarias de los expertos, revisión de documentación técnica, manuales, casos de estudio, y registros históricos.

- **Minería de datos**: Uso de algoritmos y técnicas de análisis de datos para extraer patrones, tendencias y relaciones útiles de grandes conjuntos de datos.
- **Aprendizaje automático (Machine Learning)**: Empleo de modelos de aprendizaje supervisado, no supervisado o por refuerzo para que el sistema aprenda de datos históricos y mejore su rendimiento de forma autónoma.
- **Algoritmos de extracción de información**: Uso de técnicas de procesamiento de lenguaje natural (PLN) para extraer conocimiento de textos no estructurados como artículos, investigaciones o documentos.

Este subsistema puede estar soportado por herramientas de software específicas que facilitan la integración y actualización del conocimiento en la base de conocimiento.

```
# Hechos iniciales
hechos = {
    "esta_lloviendo": False,
    "nubes_grises": True,
    "lleva_paraguas": False
}


# Reglas iniciales
reglas = [
    {
        "condiciones": ["esta_lloviendo == True"],
        "accion": lambda hechos: "Deberías llevar un paraguas."
    },
    {
        "condiciones": ["nubes_grises == True", "esta_lloviendo == False"],
       "accion": lambda hechos: "Parece que podría llover. Mejor lleva un paraguas por si acaso."
    },
    {
        "condiciones": ["nubes_grises == False", "esta_lloviendo == False"],
        "accion": lambda hechos: "No necesitas un paraguas."
    }
]
```

```
# Funciones para gestionar la adquisición de conocimiento

def agregar_hecho(hechos, nombre, valor):
    """Añadir un nuevo hecho al sistema."""
    hechos[nombre] = valor
    print(f"Hecho añadido: {nombre} = {valor}")

def eliminar_hecho(hechos, nombre):
    """Eliminar un hecho del sistema."""
    if nombre in hechos:
        del hechos[nombre]
        print(f"Hecho eliminado: {nombre}")
    else:
        print(f"Hecho no encontrado: {nombre}")

def agregar_regla(reglas, condiciones, accion):
    """Añadir una nueva regla al sistema."""
    regla = {
        "condiciones": condiciones,
        "accion": accion
    }
    reglas.append(regla)
    print(f"Regla añadida: {condiciones} -> {accion}")

def eliminar_regla(reglas, indice):
    """Eliminar una regla del sistema por su índice."""
    if 0 <= indice < len(reglas):
        reglas.pop(indice)
        print(f"Regla eliminada en el índice: {indice}")
    else:
```

```
        print("Índice de regla no válido.")


def modificar_hecho(hechos, nombre, nuevo_valor):
    """Modificar un hecho existente en el sistema."""
    if nombre in hechos:
        hechos[nombre] = nuevo_valor
        print(f"Hecho modificado: {nombre} = {nuevo_valor}")
    else:
        print(f"Hecho no encontrado: {nombre}")


# Motor de inferencia (como antes)
def motor_inferencia(hechos, reglas):
    for regla in reglas:
            condiciones_cumplidas = all(eval(condicion, {}, hechos) for condicion in regla["condiciones"])
        if condiciones_cumplidas:
            resultado = regla["accion"](hechos)
            return resultado
    return "No se encontró una regla aplicable."


# Ejemplo de uso del subsistema de adquisición de conocimiento
agregar_hecho(hechos, "temperatura", "baja")
agregar_regla(reglas, ["temperatura == 'baja'", "nubes_grises == True"], lambda hechos: "Lleva abrigo y paraguas.")
modificar_hecho(hechos, "esta_lloviendo", True)


# Ejecutar el sistema experto con los nuevos hechos y reglas
resultado = motor_inferencia(hechos, reglas)
print(resultado)
```

4. Control de la Coherencia

- **¿Qué?**: El control de la coherencia es el proceso mediante el cual se verifica que la base de conocimiento de un sistema experto sea internamente consistente y libre de conflictos. Esto implica asegurarse de que no existan contradicciones entre las reglas, hechos o datos almacenados en la base de conocimiento.
- **¿Para qué?**: Su propósito es garantizar la fiabilidad y precisión del sistema experto. Un sistema con incoherencias podría generar conclusiones incorrectas, lo que comprometería su utilidad y la confianza del usuario en las decisiones o recomendaciones proporcionadas.
- **¿Cómo?**: El control de la coherencia se lleva a cabo mediante varios métodos y técnicas:
    - **Validación de reglas**: Revisar las reglas "si-entonces" para asegurar que no existan contradicciones o solapamientos que puedan conducir a conclusiones conflictivas.
    - **Detección de inconsistencias**: Utilizar algoritmos que identifiquen y señalen posibles inconsistencias lógicas en la base de conocimiento, como reglas que se contradicen entre sí o datos que no se corresponden con las reglas establecidas.
    - **Pruebas de simulación**: Ejecutar escenarios de prueba que simulen la ejecución del sistema para observar si se generan resultados coherentes bajo diferentes condiciones.
    - **Mantenimiento y actualización**: Regularmente revisar y actualizar la base de conocimiento para corregir posibles incoherencias que puedan surgir con la adición de nuevo conocimiento.
    - **Mecanismos de retroalimentación**: Permitir que los usuarios o expertos retroalimenten el sistema cuando detectan inconsistencias en los resultados, lo que permite ajustar la base de conocimiento y mejorar su coherencia.

El control de la coherencia es una actividad continua que debe llevarse a cabo tanto durante el desarrollo inicial del sistema como a lo largo de su vida útil, a medida que se agregan nuevos conocimientos y se realizan actualizaciones.

5. El Motor de Inferencia

- **¿Qué?**: El motor de inferencia es el componente del sistema experto responsable de procesar la base de conocimiento para deducir nueva información o tomar decisiones. Actúa como el "cerebro" del sistema experto, aplicando reglas lógicas, heurísticas y algoritmos para realizar razonamientos similares a los de un ser humano experto.
- **¿Para qué?**: Su objetivo es **simular el proceso de razonamiento humano** para resolver problemas complejos en un dominio específico. El motor de inferencia permite al sistema analizar los datos de entrada, aplicar el conocimiento almacenado en la base de conocimiento y generar conclusiones, recomendaciones, diagnósticos, o predicciones basadas en esa información.

- **¿Cómo?**: El motor de inferencia opera a través de varios métodos y técnicas de razonamiento:
    - **Encadenamiento hacia adelante (Forward Chaining)**: Parte de los datos iniciales o hechos conocidos y aplica las reglas de producción de la base de conocimiento para inferir nuevas conclusiones, siguiendo un enfoque de "si-entonces" hasta que se alcanza un objetivo o se agoten las posibilidades.
    - **Encadenamiento hacia atrás (Backward Chaining)**: Comienza con una hipótesis o conclusión objetivo y trabaja "hacia atrás", buscando hechos o premisas que la justifiquen, aplicando las reglas de producción de manera inversa.
    - **Métodos heurísticos**: Utiliza estrategias de búsqueda basadas en el conocimiento específico del dominio (heurísticas) para encontrar soluciones de manera más eficiente que los métodos de búsqueda exhaustivos.
    - **Algoritmos de optimización**: Emplea técnicas como el análisis de costos y beneficios, o métodos de programación lineal para encontrar la mejor solución en problemas con múltiples variables y restricciones.
    - **Razonamiento probabilístico**: Utiliza métodos basados en la probabilidad, como redes bayesianas, para manejar incertidumbre en los datos y generar conclusiones probabilísticas.

El motor de inferencia puede también ser diseñado para explicar sus decisiones o recomendaciones a los usuarios, mostrando el razonamiento y las reglas aplicadas, lo que aumenta la transparencia y la confianza en el sistema.

```
# Hechos iniciales
hechos = {
    "esta_lloviendo": False,
    "nubes_grises": True,
    "lleva_paraguas": False
}


# Reglas iniciales
reglas = [
    {
        "condiciones": ["esta_lloviendo == True"],
        "accion": lambda hechos: "Deberías llevar un paraguas."
    },
```

```
    {
        "condiciones": ["nubes_grises == True", "esta_lloviendo == False"],
       "accion": lambda hechos: "Parece que podría llover. Mejor lleva un paraguas por si acaso."
    },
    {
        "condiciones": ["nubes_grises == False", "esta_lloviendo == False"],
        "accion": lambda hechos: "No necesitas un paraguas."
    }
]
def es_coherente_hecho(hechos, nuevo_hecho, nuevo_valor):
    """Verifica si el nuevo hecho es coherente con los hechos existentes."""
    if nuevo_hecho in hechos:
        if hechos[nuevo_hecho] != nuevo_valor:
          print(f"Conflicto detectado: El hecho '{nuevo_hecho}' ya existe con un valor diferente.")
            return False
    return True

def es_coherente_regla(reglas, nueva_regla):
    """Verifica si la nueva regla es coherente con las reglas existentes."""
    for regla in reglas:
        if set(regla["condiciones"]) == set(nueva_regla["condiciones"]):
            print("Conflicto detectado: Una regla con las mismas condiciones ya existe.")
            return False
    return True

def agregar_hecho(hechos, nombre, valor):
    """Añadir un nuevo hecho al sistema con control de coherencia."""
    if es_coherente_hecho(hechos, nombre, valor):
        hechos[nombre] = valor
        print(f"Hecho añadido: {nombre} = {valor}")
```

```
    else:
        print(f"No se pudo añadir el hecho: {nombre} = {valor}")


def agregar_regla(reglas, condiciones, accion):
    """Añadir una nueva regla al sistema con control de coherencia."""
    nueva_regla = {"condiciones": condiciones, "accion": accion}
    if es_coherente_regla(reglas, nueva_regla):
        reglas.append(nueva_regla)
        print(f"Regla añadida: {condiciones} -> {accion}")
    else:
        print("No se pudo añadir la regla debido a un conflicto de coherencia.")


# Motor de inferencia (sin cambios)
def motor_inferencia(hechos, reglas):
    for regla in reglas:
            condiciones_cumplidas = all(eval(condicion, {}, hechos) for condicion in regla["condiciones"])
        if condiciones_cumplidas:
            resultado = regla["accion"](hechos)
            return resultado
    return "No se encontró una regla aplicable."
# Ejemplo de uso del subsistema de adquisición de conocimiento con control de coherencia


agregar_hecho(hechos, "esta_lloviendo", True)  # Este hecho es coherente
agregar_hecho(hechos, "esta_lloviendo", False)  # Este hecho genera un conflicto


agregar_regla(reglas, ["esta_lloviendo == True"], lambda hechos: "Deberías llevar un paraguas.")  # Esta regla genera un conflicto
agregar_regla(reglas, ["temperatura == 'baja'", "nubes_grises == True"], lambda hechos: "Lleva abrigo y paraguas.")  # Esta regla es coherente
```

```
# Ejecutar el sistema experto con los hechos y reglas actuales
resultado = motor_inferencia(hechos, reglas)
print(resultado)
```

6. El Subsistema de Adquisición de Conocimiento

- **¿Qué?**: El subsistema de adquisición de conocimiento es el componente del sistema experto encargado de capturar, organizar y actualizar el conocimiento necesario para la base de conocimiento del sistema. Es un proceso que se enfoca en obtener la información relevante de diversas fuentes, principalmente de expertos en el dominio, para que el sistema pueda utilizarla en su razonamiento.
- **¿Para qué?**: Su finalidad es **mantener la base de conocimiento actualizada** con información precisa, completa y relevante, garantizando que el sistema experto pueda realizar inferencias correctas y adaptarse a cambios en el conocimiento del dominio. Este subsistema asegura que el sistema esté en continua evolución y aprendizaje, incorporando nuevas reglas, hechos o cambios de paradigmas.
- **¿Cómo?**: El subsistema de adquisición de conocimiento emplea diferentes técnicas para capturar y estructurar el conocimiento:
    - **Entrevistas y cuestionarios con expertos**: Realización de entrevistas estructuradas, semi-estructuradas o no estructuradas con expertos en la materia para extraer conocimientos específicos y detalles del dominio.
    - **Talleres colaborativos**: Uso de sesiones de trabajo en grupo con múltiples expertos para explorar temas complejos y consensuar el conocimiento a ser capturado.
    - **Observación directa**: Monitoreo y análisis de cómo los expertos toman decisiones en situaciones reales, para capturar conocimientos tácitos que pueden ser difíciles de articular verbalmente.
    - **Documentación y análisis de textos**: Revisión de manuales, libros, estudios de caso, artículos académicos y otros documentos relevantes para extraer información formalizada.
    - **Minería de datos y análisis automatizado**: Utilización de técnicas de minería de datos y algoritmos de aprendizaje automático para identificar patrones, tendencias y relaciones en grandes volúmenes de datos.
    - **Modelos de conocimiento**: Creación de modelos formales, como ontologías y diagramas de flujo de trabajo, para representar la estructura y dinámica del conocimiento dentro del dominio.

Este subsistema puede incluir herramientas de software especializadas que ayudan a capturar, validar, y estructurar el conocimiento, facilitando su integración en la base de conocimiento del sistema experto.

7. Interfase de Usuario

- **¿Qué?**: La interfase de usuario (IU) es el componente del sistema experto que permite la interacción entre el usuario final y el sistema. Es el medio a través del cual el usuario introduce datos, realiza consultas y recibe las respuestas, recomendaciones o explicaciones generadas por el sistema experto.
- **¿Para qué?**: Su propósito es **facilitar la comunicación y el uso eficiente del sistema** por parte del usuario. La interfaz debe ser intuitiva, amigable y comprensible, permitiendo a los usuarios aprovechar al máximo las capacidades del sistema experto sin necesidad de conocer los detalles técnicos de su funcionamiento interno.
- **¿Cómo?**: La interfaz de usuario se diseña utilizando varios elementos y técnicas:
    - **Elementos gráficos**: Uso de menús, botones, cuadros de texto, gráficos, y ventanas de diálogo para facilitar la entrada de datos y la navegación por el sistema.
    - **Lenguaje natural**: Implementación de asistentes conversacionales o chatbots que permiten a los usuarios interactuar con el sistema utilizando lenguaje natural, facilitando su uso por personas no técnicas.
    - **Sistemas de ayuda y tutoriales**: Integración de herramientas de asistencia, como mensajes emergentes, tutoriales guiados, y documentación en línea para orientar a los usuarios en la utilización del sistema.
    - **Retroalimentación visual y auditiva**: Provisión de mensajes de error claros, confirmaciones de acciones, y respuestas inmediatas a las entradas del usuario, mejorando la usabilidad.
    - **Personalización**: Ajuste de la interfaz según las preferencias del usuario, su nivel de experiencia, o su rol, para ofrecer una experiencia más relevante y efectiva.
    - **Accesibilidad**: Adaptación de la interfaz para usuarios con discapacidades mediante el uso de lectores de pantalla, control por voz, alto contraste, y otros métodos de accesibilidad.

La interfaz de usuario es crucial para asegurar que el sistema experto sea adoptado y utilizado de manera efectiva, maximizando su impacto en la resolución de problemas del dominio específico.

```
# Hechos iniciales

hechos = {

    "temperatura": "moderada",

    "esta_lloviendo": False,
```

```
    "viento_fuerte": False
}

# Reglas iniciales
reglas = [
    {
        "condiciones": ["temperatura == 'alta'", "esta_lloviendo == False"],
        "accion": lambda hechos: "Recomendamos nadar."
    },
    {
        "condiciones": ["temperatura == 'baja'", "esta_lloviendo == True"],
        "accion": lambda hechos: "Recomendamos quedarse en casa y leer un libro."
    },
    {
        "condiciones": ["temperatura == 'moderada'", "viento_fuerte == False"],
        "accion": lambda hechos: "Recomendamos ir a caminar."
    }
]
def agregar_hecho(hechos, nombre, valor):
    """Añadir un nuevo hecho al sistema."""
    if nombre in hechos:
        print(f"Hecho ya existente: {nombre} = {hechos[nombre]}. Modifícalo si es necesario.")
    else:
        hechos[nombre] = valor
        print(f"Hecho añadido: {nombre} = {valor}")

def modificar_hecho(hechos, nombre, nuevo_valor):
    """Modificar un hecho existente en el sistema."""
    if nombre in hechos:
        hechos[nombre] = nuevo_valor
```

```
        print(f"Hecho modificado: {nombre} = {nuevo_valor}")
    else:
        print(f"Hecho no encontrado: {nombre}. Añádelo primero.")

def eliminar_hecho(hechos, nombre):
    """Eliminar un hecho del sistema."""
    if nombre in hechos:
        del hechos[nombre]
        print(f"Hecho eliminado: {nombre}")
    else:
        print(f"Hecho no encontrado: {nombre}.")

def agregar_regla(reglas, condiciones, accion):
    """Añadir una nueva regla al sistema."""
    nueva_regla = {"condiciones": condiciones, "accion": accion}
    if not es_coherente_regla(reglas, nueva_regla):
        print("Conflicto detectado: Una regla con condiciones similares ya existe.")
    else:
        reglas.append(nueva_regla)
        print(f"Regla añadida: {condiciones} -> {accion}")

def modificar_regla(reglas, indice, nuevas_condiciones, nueva_accion):
    """Modificar una regla existente en el sistema."""
    if 0 <= indice < len(reglas):
        reglas[indice]["condiciones"] = nuevas_condiciones
        reglas[indice]["accion"] = nueva_accion
        print(f"Regla modificada en el índice {indice}.")
    else:
        print("Índice de regla no válido.")
```

```
def eliminar_regla(reglas, indice):
    """Eliminar una regla del sistema por su índice."""
    if 0 <= indice < len(reglas):
        reglas.pop(indice)
        print(f"Regla eliminada en el índice: {indice}")
    else:
        print("Índice de regla no válido.")


def es_coherente_regla(reglas, nueva_regla):
    """Verifica si la nueva regla es coherente con las reglas existentes."""
    for regla in reglas:
        if set(regla["condiciones"]) == set(nueva_regla["condiciones"]):
            return False
    return True


# Motor de inferencia
def motor_inferencia(hechos, reglas):
    for regla in reglas:
            condiciones_cumplidas = all(eval(condicion, {}, hechos) for condicion in
regla["condiciones"])
        if condiciones_cumplidas:
            resultado = regla["accion"](hechos)
            return resultado
    return "No se encontró una regla aplicable."
# Agregar un nuevo hecho
agregar_hecho(hechos, "humedad_alta", True)


# Modificar un hecho existente
modificar_hecho(hechos, "temperatura", "alta")
```

```
# Eliminar un hecho
eliminar_hecho(hechos, "viento_fuerte")


# Agregar una nueva regla
agregar_regla(reglas, ["temperatura == 'alta'", "humedad_alta == True"], lambda hechos: "Recomendamos beber mucha agua y descansar en sombra.")


# Modificar una regla existente
modificar_regla(reglas, 1, ["temperatura == 'baja'", "esta_lloviendo == True"], lambda hechos: "Recomendamos preparar una bebida caliente y ver una película.")


# Ejecutar el sistema experto con los hechos y reglas actuales
resultado = motor_inferencia(hechos, reglas)
print(resultado)
```

8. El Subsistema de Ejecución de Órdenes

- **¿Qué?**: El subsistema de ejecución de órdenes es el componente de un sistema experto que se encarga de ejecutar las acciones o comandos resultantes de las inferencias y recomendaciones del motor de inferencia. Su función es llevar a cabo tareas prácticas o aplicar decisiones en el entorno real o digital.
- **¿Para qué?**: Su propósito es **implementar las soluciones** o recomendaciones generadas por el sistema experto en un contexto práctico. Esto puede incluir la ejecución de comandos, la modificación de sistemas, la actualización de bases de datos, o la integración con otros sistemas para aplicar las decisiones tomadas por el sistema experto.
- **¿Cómo?**: El subsistema de ejecución de órdenes opera de varias maneras:
    - **Automatización de procesos**: Ejecuta comandos y acciones automáticamente basados en las conclusiones del sistema, como en sistemas de control industrial donde las decisiones del sistema pueden ajustar configuraciones de maquinaria o procesos.
    - **Interacción con otros sistemas**: Se integra con sistemas externos, como bases de datos, sistemas de gestión empresarial, o plataformas de control, para realizar tareas como la actualización de registros, envío de notificaciones, o ejecución de transacciones.

- **Generación de informes**: Produce informes y resúmenes sobre las decisiones o acciones tomadas, que pueden ser utilizados por los usuarios para seguimiento o auditoría.
- **Aplicación de cambios**: Implementa cambios en el entorno en función de las recomendaciones del sistema, como en sistemas de diagnóstico médico que pueden sugerir tratamientos o ajustes en el plan de atención.
- **Interacción con dispositivos físicos**: En sistemas de automatización y control, envía comandos a dispositivos físicos para realizar acciones específicas, como ajustar temperaturas, activar alarmas, o manejar equipos.

El diseño de este subsistema debe considerar aspectos de seguridad, precisión y control, ya que las acciones ejecutadas pueden tener un impacto significativo en el entorno o en los procesos en los que el sistema experto está involucrado.

```
# Definimos las acciones del sistema
def encender_dispositivo(nombre_dispositivo):
    print(f"{nombre_dispositivo} ha sido encendido.")


def apagar_dispositivo(nombre_dispositivo):
    print(f"{nombre_dispositivo} ha sido apagado.")


def enviar_notificacion(mensaje):
    print(f"Notificación enviada: {mensaje}")


# Acciones específicas para diferentes dispositivos
acciones = {
    "encender_luz": lambda: encender_dispositivo("Luz del salón"),
    "apagar_luz": lambda: apagar_dispositivo("Luz del salón"),
    "encender_calefaccion": lambda: encender_dispositivo("Calefacción"),
    "apagar_calefaccion": lambda: apagar_dispositivo("Calefacción"),
    "enviar_alerta_clima": lambda: enviar_notificacion("Alerta de clima adverso detectado."),
}
def ejecutar_orden(orden):
    """Ejecuta la orden proporcionada por el motor de inferencia."""
    if orden in acciones:
```

```
        acciones[orden]()
        print(f"Orden '{orden}' ejecutada con éxito.")
    else:
        print(f"Orden '{orden}' no reconocida o no disponible.")


# Motor de inferencia actualizado
def motor_inferencia(hechos, reglas):
    for regla in reglas:
            condiciones_cumplidas = all(eval(condicion, {}, hechos) for condicion in regla["condiciones"])
        if condiciones_cumplidas:
            resultado = regla["accion"](hechos)
            ejecutar_orden(resultado)  # Integración con el subsistema de ejecución
            return resultado
    return "No se encontró una regla aplicable."
def ejecutar_orden(orden):
    """Ejecuta la orden proporcionada por el motor de inferencia."""
    if orden in acciones:
        acciones[orden]()
        print(f"Orden '{orden}' ejecutada con éxito.")
    else:
        print(f"Orden '{orden}' no reconocida o no disponible.")


# Motor de inferencia actualizado
def motor_inferencia(hechos, reglas):
    for regla in reglas:
            condiciones_cumplidas = all(eval(condicion, {}, hechos) for condicion in regla["condiciones"])
        if condiciones_cumplidas:
            resultado = regla["accion"](hechos)
            ejecutar_orden(resultado)  # Integración con el subsistema de ejecución
```

```
            return resultado
    return "No se encontró una regla aplicable."
# Hechos iniciales
hechos = {
    "temperatura": "baja",
    "esta_lloviendo": True,
    "viento_fuerte": False
}

# Reglas actualizadas con órdenes
reglas = [
    {
        "condiciones": ["temperatura == 'baja'", "esta_lloviendo == True"],
        "accion": lambda hechos: "encender_calefaccion"  # Orden a ejecutar
    },
    {
        "condiciones": ["temperatura == 'alta'", "esta_lloviendo == False"],
        "accion": lambda hechos: "apagar_calefaccion"
    },
    {
        "condiciones": ["viento_fuerte == True"],
        "accion": lambda hechos: "enviar_alerta_clima"
    }
]
# Ejecutar el sistema experto con los hechos y reglas actuales
resultado = motor_inferencia(hechos, reglas)
print(resultado)
```

9. El Subsistema de Explicación

- **¿Qué?**: El subsistema de explicación es el componente de un sistema experto encargado de proporcionar al usuario una justificación clara y comprensible sobre las decisiones, recomendaciones o conclusiones alcanzadas por el sistema. Este subsistema facilita la transparencia y ayuda al usuario a entender el razonamiento detrás de las respuestas del sistema.
- **¿Para qué?**: Su propósito es **aumentar la confianza y la aceptación** del sistema experto, permitiendo a los usuarios comprender cómo y por qué se llegó a ciertas conclusiones o decisiones. Esto es esencial para la validación de las recomendaciones del sistema, especialmente en aplicaciones críticas como la medicina o la ingeniería.
- **¿Cómo?**: El subsistema de explicación utiliza varias técnicas para comunicar el razonamiento del sistema:
    - **Rastreo de inferencia**: Muestra el proceso lógico seguido por el motor de inferencia, incluyendo las reglas aplicadas, los hechos utilizados, y el flujo de razonamiento desde los datos de entrada hasta la conclusión final.
    - **Desglose de decisiones**: Proporciona una explicación detallada sobre cada paso del proceso de toma de decisiones, incluyendo cómo se evaluaron las opciones y se llegó a la recomendación final.
    - **Visualización de datos**: Utiliza gráficos, diagramas o tablas para representar la información y el proceso de manera visual, facilitando la comprensión de las relaciones y el razonamiento.
    - **Lenguaje natural**: Genera descripciones en lenguaje natural que expliquen las conclusiones de manera clara y accesible para los usuarios no técnicos.
    - **Comparaciones y ejemplos**: Ofrece comparaciones con situaciones similares o ejemplos específicos para ilustrar cómo se aplica el conocimiento y por qué se eligió una determinada solución.

El diseño de este subsistema debe ser flexible y adaptarse a las necesidades y niveles de experiencia de los usuarios, para asegurar que las explicaciones sean útiles y comprensibles.

```
# Definimos las acciones del sistema
def encender_dispositivo(nombre_dispositivo):
    print(f"{nombre_dispositivo} ha sido encendido.")

def apagar_dispositivo(nombre_dispositivo):
    print(f"{nombre_dispositivo} ha sido apagado.")
```

```
def enviar_notificacion(mensaje):
    print(f"Notificación enviada: {mensaje}")

# Acciones específicas para diferentes dispositivos
acciones = {
    "encender_luz": lambda: encender_dispositivo("Luz del salón"),
    "apagar_luz": lambda: apagar_dispositivo("Luz del salón"),
    "encender_calefaccion": lambda: encender_dispositivo("Calefacción"),
    "apagar_calefaccion": lambda: apagar_dispositivo("Calefacción"),
    "enviar_alerta_clima": lambda: enviar_notificacion("Alerta de clima adverso detectado."),
}

# Definimos la función ejecutar_orden
def ejecutar_orden(orden):
    """Ejecuta la orden proporcionada por el motor de inferencia."""
    if orden in acciones:
        acciones[orden]()
        print(f"Orden '{orden}' ejecutada con éxito.")
    else:
        print(f"Orden '{orden}' no reconocida o no disponible.")

# Subsistema de explicación
explicaciones = []

def agregar_explicacion(explicacion):
    """Añade una explicación a la lista de explicaciones."""
    explicaciones.append(explicacion)

def mostrar_explicaciones():
    """Muestra todas las explicaciones de las decisiones tomadas."""
```

```
    if explicaciones:
        print("Explicaciones de las decisiones:")
        for explicacion in explicaciones:
            print(f"- {explicacion}")
    else:
        print("No hay explicaciones disponibles. No se tomaron decisiones.")


# Motor de inferencia actualizado con explicación
def motor_inferencia(hechos, reglas):
    for regla in reglas:
            condiciones_cumplidas = all(eval(condicion, {}, hechos) for condicion in regla["condiciones"])
        if condiciones_cumplidas:
            resultado = regla["accion"](hechos)
            ejecutar_orden(resultado)  # Ejecuta la orden
            agregar_explicacion(regla["explicacion"])  # Agrega la explicación
            return resultado
    return "No se encontró una regla aplicable."


# Hechos iniciales
hechos = {
    "temperatura": "baja",
    "esta_lloviendo": True,
    "viento_fuerte": False
}


# Reglas con explicaciones
reglas = [
    {
        "condiciones": ["temperatura == 'baja'", "esta_lloviendo == True"],
```

```
        "accion": lambda hechos: "encender_calefaccion",
        "explicacion": "La temperatura es baja y está lloviendo, por lo que es recomendable encender la calefacción."
    },
    {
        "condiciones": ["temperatura == 'alta'", "esta_lloviendo == False"],
        "accion": lambda hechos: "apagar_calefaccion",
        "explicacion": "La temperatura es alta y no está lloviendo, por lo que no se necesita la calefacción."
    },
    {
        "condiciones": ["viento_fuerte == True"],
        "accion": lambda hechos: "enviar_alerta_clima",
        "explicacion": "El viento es fuerte, se envía una alerta de clima adverso."
    }
]

# Ejecutar el sistema experto con los hechos y reglas actuales
resultado = motor_inferencia(hechos, reglas)
print(resultado)

# Mostrar explicaciones de las decisiones tomadas
mostrar_explicaciones()
```

10. El Subsistema de Aprendizaje

- **¿Qué?**: El subsistema de aprendizaje es el componente de un sistema experto que permite al sistema mejorar y adaptarse continuamente a partir de nuevos datos, experiencias, o retroalimentación. Este subsistema está diseñado para actualizar y refinar la base de conocimiento y mejorar el rendimiento del sistema con el tiempo.
- **¿Para qué?**: Su propósito es **mantener la relevancia y precisión del sistema** al permitir que el sistema experto se adapte a cambios en el dominio, descubra nuevas tendencias o patrones, y mejore su capacidad para resolver problemas y tomar

decisiones. Esto es crucial para sistemas que operan en entornos dinámicos o que requieren adaptación a nuevas situaciones.

- **¿Cómo?**: El subsistema de aprendizaje puede implementar varias técnicas y métodos:
    - **Aprendizaje automático (Machine Learning)**: Utiliza algoritmos de aprendizaje supervisado, no supervisado o por refuerzo para ajustar modelos y reglas en función de datos históricos y nuevos. Los modelos de ML pueden identificar patrones, hacer predicciones y ajustar las recomendaciones del sistema.
    - **Actualización de reglas**: Modifica o añade nuevas reglas a la base de conocimiento en función de la retroalimentación recibida y de nuevos datos. Esto puede incluir el ajuste de los parámetros de las reglas existentes o la incorporación de nuevas reglas basadas en descubrimientos recientes.
    - **Retroalimentación del usuario**: Utiliza comentarios y evaluaciones de los usuarios para ajustar las respuestas y mejorar el sistema. La retroalimentación puede ser utilizada para identificar áreas de mejora y actualizar el conocimiento del sistema.
    - **Análisis de resultados**: Evalúa el rendimiento del sistema mediante la comparación de las recomendaciones o decisiones con resultados reales. Los errores o desvíos identificados pueden ser utilizados para ajustar y mejorar el sistema.
    - **Adaptación a cambios en el dominio**: Incorpora nuevos conocimientos o cambios en el dominio, como la evolución de tecnologías, regulaciones, o mejores prácticas, para mantener el sistema actualizado y relevante.

El subsistema de aprendizaje asegura que el sistema experto no solo se mantiene al día con la información más reciente, sino que también mejora su capacidad de resolución de problemas y toma de decisiones a medida que aprende de la experiencia y los datos nuevos.

```
# Definición del subsistema de aprendizaje
def ajustar_reglas(hechos, resultado_esperado, reglas):
    """
    Ajusta las reglas del sistema experto en función del resultado esperado proporcionado por el usuario.
    """
    for regla in reglas:
        # Verificar si la regla se aplica a los hechos actuales
            condiciones_cumplidas = all(eval(condicion, {}, hechos) for condicion in regla["condiciones"])
```

```
        if condiciones_cumplidas:
            # Si la acción de la regla no coincide con el resultado esperado, ajustar la regla
            if regla["accion"](hechos) != resultado_esperado:
                print(f"Ajustando regla para {regla['accion'](hechos)} a {resultado_esperado}.")
                regla["accion"] = lambda hechos: resultado_esperado
                regla["explicacion"] = f"La regla fue ajustada automáticamente para mejorar la precisión basada en datos recientes."
                return f"Regla ajustada a {resultado_esperado}."
    return "No se encontró una regla para ajustar."

# Definición de hechos y reglas para la prueba
hechos = {'temperatura': 20, 'humedad': 50}  # Ejemplo de hechos
reglas = [
    {
        'condiciones': ['temperatura > 22'],
        'accion': lambda hechos: 'encender_calefaccion',
        'explicacion': 'Se enciende la calefacción si la temperatura es menor o igual a 22°C.'
    },
    {
        'condiciones': ['temperatura <= 22'],
        'accion': lambda hechos: 'apagar_calefaccion',
        'explicacion': 'Se apaga la calefacción si la temperatura es mayor a 22°C.'
    }
]

# Ejemplo de retroalimentación del usuario
resultado_esperado = 'apagar_calefaccion'  # Por ejemplo, el usuario esperaba que la calefacción se apague.

# Motor de inferencia con integración de aprendizaje
def motor_inferencia_aprendizaje(hechos, reglas, resultado_esperado=None):
```

```
    for regla in reglas:
            condiciones_cumplidas = all(eval(condicion, {}, hechos) for condicion in regla["condiciones"])
        if condiciones_cumplidas:
            resultado = regla["accion"](hechos)
            ejecutar_orden(resultado)  # Ejecuta la orden
            agregar_explicacion(regla["explicacion"])  # Agrega la explicación
            if resultado_esperado:
                # Ajustar las reglas basadas en la retroalimentación del usuario
                ajuste = ajustar_reglas(hechos, resultado_esperado, reglas)
                print(ajuste)
            return resultado
    return "No se encontró una regla aplicable."

# Funciones adicionales necesarias
def ejecutar_orden(orden):
    print(f"Orden ejecutada: {orden}")

def agregar_explicacion(explicacion):
    print(f"Explicación: {explicacion}")

def mostrar_explicaciones():
    print("Mostrando todas las explicaciones.")

# Ejecutar el sistema experto con aprendizaje
resultado = motor_inferencia_aprendizaje(hechos, reglas, resultado_esperado)
print(resultado)

# Mostrar explicaciones de las decisiones tomadas
mostrar_explicaciones()
```