INFORME DE LABORATORIO 2: SIMULACIÓN DE UN SISTEMA E INTERACCIÓN DE CHATBOTS EN PROLOG

Nombre: Reinaldo Pacheco Parra
Profesor: Gonzalo Martínez Ramírez
Asignatura: Paradigmas de programación (2/2023)

Índice

Introducción

En el siguiente informe, se presenta el desarrollo de un sistema de interacción de chatbots utilizando el Paradigma Lógico en el lenguaje Prolog a través del compilador SWI-Prolog. Este informe se estructura en varias secciones clave, que abarcan la descripción del problema y los objetivos, la explicación del paradigma lógico, aspectos de análisis, diseño e implementación de la solución.
Se discute la razón por la cual se optó por una implementación en particular en lugar de otras alternativas y se destacan las ventajas de esta elección. A continuación, se exploran algunos de los predicados específicos que se han implementado y los enfoques que se tuvieron que considerar durante el proceso de desarrollo. Posteriormente, se proporcionan las instrucciones necesarias para verificar el correcto funcionamiento del programa. Para concluir, se realiza un análisis de los resultados obtenidos a partir de la implementación y se lleva a cabo una autoevaluación detallada de cada uno de los predicados implementados en el sistema de interacción de chatbots.

Descripción breve del problema:

Se busca realizar una simulación de un sistema de chatbots de ITR (interacción respuesta a texto) los cuales responden a preguntas en base a opciones especificas ya definidas por algún número o palabra clave que permite diferenciarlos entre sí. Este simulador de chatbots debe permitir realizar diferentes operaciones convencionales tales como: crear un flujo de opciones, crear un chatbot, agregar un flujo dentro de un chatbot, crear un sistema de chatbots, registrar usuarios para poder iniciar y cerrar sesión en el sistema, simular una conversación con un chatbot, entre otros.

Descripción breve del paradigma:

La programación lógica es un paradigma de programación que considera la computación como un conjunto de hechos y reglas que forman una "base de conocimiento", la cual siempre se asume verdadera. En este paradigma, todos los elementos se denominan "términos" y pueden interconectarse para establecer relaciones. A diferencia de la ejecución de secuencias de comandos característica de la programación imperativa o la evaluación de expresiones propia de la programación funcional, la programación lógica se fundamenta en la lógica booleana, empleando valores de verdad (verdadero o falso) y conectivos lógicos (y, o).
Mediante consultas a la base de conocimiento, el sistema determina la veracidad de una afirmación devolviendo verdadero si se encuentra una solución a la consulta, o falso en caso contrario.

Otros elementos importantes de este paradigma son los siguientes:

Consulta: Corresponde a una petición a la base de conocimientos para obtener información de ella.

Variable: Corresponde a una incógnita a resolver, es un término que siempre comienza por una letra mayúscula.

Unificación: Corresponde al proceso por el cual se determina si una consulta es válida o verdadera.

Inferencia: Corresponde al proceso por el cual a través de una variable se pueden descubrir otras variables relacionadas.

Desarrollo

Análisis del problema

Para desarrollar un programa de simulación de chatbots, se han identificado seis Tipos de Datos Abstractos (TDA) esenciales: Sistema, Chatbot, Opción, Flujo, Usuario y ChatHistory. La estrategia para abordar el sistema de chatbots consiste en visualizarlo como una estructura de datos. En este contexto, se ha elegido la estructura de lista debido a su fácil representación y manejo en comparación con otras estructuras más complejas, como los grafos o árboles. El primer elemento de la estructura se basa en la "Opción", que representa las posibles elecciones que un Chatbot puede ofrecer al usuario. Seguidamente, el "Flujo" agrupa un conjunto de estas opciones, modelándose como una lista de opciones. De manera análoga, el "Chatbot" se compone de múltiples flujos, por lo que se representa como una lista de flujos, que, a su vez, son listas de opciones. La construcción del sistema sigue un enfoque similar, ya que se representa como una lista que contiene una lista de chatbots, cada uno con su propia estructura de flujos y opciones, así como otros elementos como "Usuario" y "ChatHistory", los cuales también se modelan como listas

La representación de cada uno de los elementos es la siguiente:

- **System:** Un sistema tiene una representación en forma de lista que contiene un nombre de sistema, un id de chatbot inicial y un conjunto de 0 o más chatbots.
  (name (string) X InitialChatbotCodeLink (int) X chatbot)*
- **Chatbot:** Un chatbot tiene una representación en forma de lista que contiene un id de chatbot, un nombre de chatbot, un mensaje de bienvenida, un id de inicio de flujo y un conjunto de 0 o más flujos.
  (chatbotID (int) x name (string) x welcomeMessage (string) x startFlowId(int) x flows)*
- **Option:** Una opción tiene una representación en forma de lista que contiene un código de opción, un mensaje, un código de chatbot, un código inicial y un conjunto de 0 o más palabras claves.
  (code (int) x message (string) x ChatbotCodeLink (int) x InitialFlowCodeLink (int) x keyword)*
- **Flow:** Un flujo tiene una representación en forma de lista que contiene un código de flujo, un nombre y un conjunto de 0 o más opciones.
  (id (int) X name-msg (string) X option)*
- **User:** Un usuario tiene una representación en forma de lista que contiene el nombre del usuario.
  (username (string))
- **Chat-history:** Un historial de chat tiene una representación en forma de lista que almacena el historial de un usuario.
  (history (string))

Además, se deben realizar las siguientes operaciones entre los elementos:
option: crea una opción
flow: crea un flujo (conjunto de opciones)
flow-add-option: permite agregar una opción dentro de un flujo
chatbot: crea un chatbot.
chatbot-add-flow: permite agregar un flujo dentro de un chatbot
system: crear un sistema
system-add-chatbot: permite agregar un chatbot al sistema
system-add-user: permite agregar un usuario al sistema
system-login: inicia una sesión de un usuario en el sistema (el usuario debe estar registrado para poder iniciar sesión)
system-logout: cierra una sesión de un usuario en el sistema.
system-talk: permite interactuar con un chatbot.
system-synthesis: ofrece una síntesis del chatbot a partir del chat-history.
system-simulate: simula un diálogo entre dos chatbots en el sistema.

Diseño de la solución

El diseño de la solución implementado, utilizando Programación Lógica, consiste en representar los elementos del sistema de chatbots como listas. Esto se hace dado que esta es una de las estructuras de datos más sencillas de implementar en Prolog, además de que esta representación permite agregar elementos dentro de otros. Por ejemplo, para la representación de un flujo de opciones, cada opción se debe pensar como una lista, los cuales estarán a su vez, contenidos dentro de otra lista y así sucesivamente con los demás elementos del sistema. Por lo que, para la solución se debe trabajar con listas dentro de listas.
Uno de los enfoques principales de la problemática es que algunos de los elementos están representados por un id único que puede ser un entero o un string, lo cual evita que, al momento de agregar un elemento, no se tenga el mismo elemento dentro de la lista, si no, que, para efectos específicos del laboratorio, el sistema debe entregar false. Los casos específicos en donde esto debe suceder son:

- El predicado flow contiene dos o más opciones que tienen el mismo Code.
- El predicado flowAddOption agrega una opción que ya se encuentra dentro del flujo.
- El predicado chatbot contienen dos o más flujos con el mismo Id.
- El predicado chatbotAddFlow agrega un flujo que ya se encuentra dentro del chatbot.
- El predicado system contiene dos o más chatbots con el mismo ChatbotID
- El predicado systemAddChatbot agrega un chatbot que ya se encuentra dentro del sistema.
- El predicado systemAddUser agrega un usuario que ya se encuentra registrado.
- Al llamar al predicado systemLogin con un usuario que no está registrado en el sistema.
- Al llamar al predicado systemLogin cuando ya se encuentra un usuario logeado en el sistema.
- Al llamar al predicado systemTalkRec, en el sistema no hay un usuario logeado.
- Al llamar al predicado systemTalkRec, el usuario escribe una opción que no se encuentra en las keywords.

Los predicados esenciales para la construcción de la solución, **“pertenece”** y **“noPertenece”** nos permiten verificar si un elemento se encuentra o no dentro de una lista, esto es muy importante debido a que gran parte de las consultas tiene que ver con el evitar crear una lista con 2 o más elementos iguales y evitar agregar un elemento a una lista que previamente ya lo contiene. Por ello, se construyen estos dos predicados de la capa de pertenencia. *(ver Figura 1 y Figura 2 en Anexos)*

Otro de los predicados importantes de la construcción de la solución es **“flowAddOption”** el cual permite agregar una opción a un flujo, el cual hace uso de los predicados anteriores. Para poder agregar una opción a un flujo, primero debemos comprobar de que la opción no exista previamente en el flujo, haciendo uso del predicado **“noPertenece”** *(ver figura 3 en Anexos)*

De igual forma, en el predicado auxiliar **“addUser”** hace uso de un predicado llamado **“noPertenece2”** que trabaja bajo la misma lógica, solo que con la diferencia de que, en este caso, se compara un string, siendo el nombre del usuario. Se comprueba de que el nombre del usuario no exista previamente en la lista de usuarios registrados para poder agregarlo. *(ver figura 4 en Anexos)*

Aspectos de implementación

Para la resolución del problema, se utiliza el Paradigma Lógico mediante el lenguaje de programación Prolog en el compilador SWI-Prolog de versión 9.0.4 sin hacer uso de bibliotecas externas. Además, para efectos de pruebas, se utilizó la versión online de Prolog llamada SWISH.

Para la implementación se tiene una representación con TDA para cada elemento ya que permite organizar y ordenar de mejor forma los elementos. Por último, la construcción de la mayoría de los predicados se distribuye en las siguientes capas

- **Constructor:** Crea la estructura o representación. Se representa con la palabra "make".
- **Pertenencia:** Valida operaciones o reglas, entrega un booleano. Se representa con "is?" o "exists?"
- **Selector:** Obtiene datos específicos de una representación. Se representa con la palabra "get".
- **Modificador:** Permite modificar datos de la representación. Se representa con "set".
- **Otros**

Los TDA utilizados en la implementación son:

- **TDA System**
- **TDA Chabot**
- **TDA Flow**
- **TDA Option**
- **TDA User**
- **TDA ChatHistory**

(ver Tablas en Anexo donde se adjuntan todos los predicados implementados por nombre, capa, recursión y descripción)

Instrucciones de uso

Ejemplos de uso

Para el correcto uso de este programa, se deben tener todos los archivos llamados "TDA" y el archivo de prueba dentro de la misma carpeta. Esto con el fin de que compartan la misma ubicación para evitar errores al momento de hacer llamadas a predicados entre los TDA.

El primer archivo llamado pruebas contiene el script de pruebas, el cual nos permite crear opciones, flujos, chatbots y un sistema, además de insertar opciones dentro de un flujo, flujos dentro de un chatbot e insertar chatbots dentro de un sistema. También permite registrar, iniciar sesión y cerrar sesión de un usuario.

Para verificar el correcto funcionamiento del programa, se debe abrir la carpeta "lab2_21133732_PachecoParra" y dirigirse al archivo de pruebas "pruebas_21133732_PachecoParra" el cual contiene 2 script. Se debe ejecutar el archivo, copiar el script de pruebas y pegarlo en el apartado de consultas, es necesario saber que los casos en que el programa entrega false se encuentran comentados, esto con el fin de evitar que Prolog no siga revisando los demás predicados. Es por esto, que el revisor debe borrar el carácter "%" para quitar el predicado comentado y ejecutarlo nuevamente, esto se debe hacer para cada uno de los predicados los cuales también están destacados y mencionados en el archivo.

Resultados Esperados

Se espera que el programa logre crear correctamente un sistema completo el cual contiene una lista de chatbots, a su vez, cada chatbot contiene un conjunto de flujos y cada flujo tiene las opciones correspondientes. Permitir registrar e iniciar sesión a usuarios y guardar el historial de chat de la conversación, además de realizar interacciones permitiendo al chatbot entregar respuesta dependiendo de las opciones solicitadas por el usuario.

Otro de los resultados esperados es que el programa maneje correctamente los casos borde en donde se debe entregar false para predicados específicos durante las pruebas.

Posibles errores

Para el predicado systemTalkRec se toma el supuesto de que la primera interacción inicia el primer chatbot, esto debido a que en el archivo de script de prueba se hicieron varios cambios durante la realización del laboratorio, esto se hace con el fin de que la interacción con el chatbot tenga linealidad y no se pierda el sentido de la conversación con el usuario.

Resultados y autoevaluación

Resultados

Los predicados implementados se probaron de múltiples formas para verificar su correcto funcionamiento. Se verificaron mediante los 2 script de prueba disponibles, funcionando correctamente para todos obteniendo las respuestas y resultados esperados.
Los predicados systemSymulate y systemSinthesis no se lograron implementar correctamente debido a la complejidad de estas.

Autoevaluación

Los predicados flow, flowAddOption, chatbot, chatbotAddFlow, system, systemAddChatbot, systemAddUser, systemLogin, systemLogout fueron implementadas en su totalidad funcionando correctamente para ambos script y pruebas realizadas. El predicado systemTalkRec se implementó parcialmente. Por último, los predicados systemSimulate y systemSinthesis fueron implementadas.

Conclusión

Después de llevar a cabo el programa de simulación de un sistema de chatbots utilizando el Paradigma Lógico en Prolog, es evidente que se lograron satisfacer la mayoría de los requisitos planteados en la problemática. Inicialmente, se enfrentó el desafío de comprender y adaptarse al paradigma lógico, dado que previamente se había trabajado con el paradigma funcional en el laboratorio anterior. Sin embargo, a medida que se avanzó en la implementación de los requisitos se observó que algunos de los predicados, se podían implementar con una perspectiva similar al paradigma funcional, pero utilizando la sintaxis del lenguaje Prolog.

En comparación al paradigma funcional, este paradigma ofreció ventajas como implementación de reglas y relaciones similares entre sí, permitiendo reutilizar predicados, principalmente el “pertenece” y “noPertenece”. En cuanto a desventajas, Prolog no permite utilizar funciones, por lo que, para realizar alguna operación se requiere de mucho más código que en otros paradigmas, además de que se debe crear una base de conocimiento antes de trabajar con él. Fue crucial abordar el problema de manera adecuada y trabajar con listas dentro de listas, ya que elegir otra representación, como la representación en forma de árbol, habría complicado significativamente la construcción de la solución. Además, el uso de la capa de Tipos de Datos Abstractos (TDA) permitió una mejor organización de la problemática al posibilitar la creación de predicados selectores, de pertenencia y constructoras que podían ser llamadas en múltiples predicados, reduciendo considerablemente la cantidad de código en algunas de ellas y permitiendo una mejor organización e implementación del proyecto.

Bibliografía y referencias

Prolog Manual (n.d) extraído de: https://www.swi-prolog.org/pldoc/doc_for?object=manual

Flores Sánchez, V. (2023). Proyecto Semestral de Laboratorio Paradigmas de Programación 2023/2
https://docs.google.com/document/d/1QCk3k-HCShNSByEZHoEIFbrxoOJSnPblyjiJDj3Q9Jk/edit

González Ibáñez, R. (2023). Proyecto Semestral de Laboratorio General 2023/2
https://docs.google.com/document/d/1L-B2b3J71Baqa_IuZt6EmRwDlxCoqzWn9YJAm6FIiJk/edit

Anexos

```
% nombre predicado: pertenece
% Descripcion: predicado que verifica si un elemento pertenece a una lista
% Dom: elemento X lista
% MetaPrimaria: pertenece/2
pertenece(Elem,[Elem|_]).
pertenece(Elem,[_|Resto]):-
    pertenece(Elem,Resto).
```

Figura 1: Predicado pertenece

```
% nombre predicado: noPertenece
% Descripcion: predicado que verifica si un elemento no pertenece a una lista
% Dom: elemento X lista
% MetaPrimaria: noPertenece/2
noPertenece(Elem,Lista):-
    \+pertenece(Elem,Lista).
```

Figura 2: Predicado noPertenece

```
% RF4:
% nombre predicado: flowAddOption
% Descripcion: predicado que agrega una opcion a un flujo
% Dom: flow X option X newFlow
% MetaPrimaria: flowAddOption/3
% MetaSecundaria: getFlowId/2, getFlowNameMessage/2, getFlowOptions/2, removeDuplicates/4, flow/4
flowAddOption(Flow, Option, NewFlow):-
    getFlowOptions(Flow, Options),
    noPertenece(Option, Options),
    getFlowId(Flow, Id),
    getFlowNameMessage(Flow, Name),
    flow(Id, Name, [Option|Options], NewFlow).
```

Figura 3: Predicado flowAddOption

```
% nombre predicado: addUser
% Descripcion: predicado que agrega un usuario a la
% Dom: userName X users X newUsers
% MetaPrimaria: addUser/3
% MetaSecundaria: noPertenece/2, append/3, user/2
addUser([],Name,[Name]).
addUser(UserRegistered, Users, NewUsers):-
    noPertenece2(Users, UserRegistered),
    add(Users, UserRegistered, NewUsers).
addUser(UserRegistered, _, UserRegistered).
```

Figura 4: Predicado AddUser

Tabla 1: TDA Option

Nombre	Capa	Recursión	Descripción
option	Constructor	No aplica	Crea una opción
getOptionCode	Selector	No aplica	Obtiene el código de la opción
getOptionMessage	Selector	No aplica	Obtiene el mensaje de la opción
getOptionChatbotCodeLink	Selector	No aplica	Obtiene el code de unión con el chatbot
getOptionInitialFlowCodeLink	Selector	No aplica	Obtiene el código de flujo

getOptionKeywords	Selector	No aplica	Obtiene las palabras claves de una opción

Tabla 2: TDA Flow

Nombre	Capa	Recursión	Descripción
flow	Constructor	No aplica	Construye un flujo
removeDuplicates	Otros	No aplica	Elimina las opciones duplicadas
getFlowId	Selector	No aplica	Obtiene el id de un flujo
getFlowNameMessage	Selector	No aplica	Obtiene el mensaje de un flujo
getFlowOptions	Selector	No aplica	Obtiene la lista de opciones de un flujo
noOpcionesDuplicadas	Otros	No aplica	Verifica si no hay opciones duplicadas en una lista de opciones
flowAddOption	Modificador	No aplica	Añade opciones a un flujo
pertenece	Pertenencia	No aplica	Verifica si un elemento pertenece a una lista
noPertenece	Pertenencia	No aplica	Verifica si un elemento NO pertenece a una lista

Tabla 3: TDA Chatbot

Nombre	Capa	Recursión	Descripción
chatbot	Constructor	No aplica	Crea un chatbot
newChatbot	Otros	No aplica	Crea un nuevo chatbot auxiliar
getChatbotID	Selector	No aplica	Obtiene el id del chatbot
getChatbotNameChatbot	Selector	No aplica	Obtiene el nombre del chatbot
getChatbotWelcomeMessage	Selector	No aplica	Obtiene el mensaje de bienvenida del chatbot
getChatbotStartFlowId	Selector	No aplica	Obtiene el id de flujo inicial del chatbot
getChatbotFlows	Selector	No aplica	Obtiene los flujos del chatbot
chatbotAddFlow	Modificador	No aplica	Agrega un flujo en un chatbot
chatbotAddFlowRec	Modificador	Si	Auxiliar que agrega un flujo en un chatbot
noFlowsDuplicados	Pertenencia	No aplica	Verifica si no hay flujos duplicados en una lista de flujos

Tabla 4: TDA System

Nombre	Capa	Recursión	Descripción
system	Constructor	No aplica	Crea un sistema
getSystemName	Selector	No aplica	Obtiene el nombre del sistema
getSystemInitialChatbotCodeLink	Selector	No aplica	Obtiene el código inicia de chatbot del sistema
getSystemChatbots	Selector	No aplica	Obtiene la hora de creación del sistema
getSystemChatHistory	Selector	No aplica	Obtiene la lista de chatbots del sistema
getUsuarioRegistrado	Selector	No aplica	Obtiene la lista de usuarios registrados
getUsuarioLogeado	Selector	No aplica	Obtiene una lista con el usuario logeado
auxSystem	Constructor	No aplica	Crea un sistema auxiliar
setUsuarioLogeado	Modificador	No aplica	Modifica el usuario logeado

removeDuplicatesChatbot	Otros	No aplica	Elimina los chatbots duplicados en una lista de chatbots
noChatbotsDuplicados	Pertenencia	No aplica	Verifica si NO hay un chatbot en una lista de chatbots
chatbotsDuplicados	Pertenencia	No aplica	Verifca si hay un chatbot en una lista de chatbots
tieneDuplicados	Pertenencia	No aplica	Verifica si hay elementos duplicados dentro de una lista
systemAddChatbot	Modificador	No aplica	Agrega un chatbot a un sistema
noPertenece2	Pertenencia	No aplica	Verifica si un elemento no pertenece a una lista
addUser	Modificador	No aplica	Agrega un usuario a la lista de usuarios
systemAddUser	Modificador	No aplica	Agrega un usuario al sistema
userRegistered	Pertenencia	No aplica	Verifica si un usuario está registrado en el sistema
userLogged	Pertenencia	No aplica	Verifica si un usuario está logeado en el sistema
systemLogin	Modificador	No aplica	Inicia sesión de un usuario en el sistema
systemLogout	Modificador	No aplica	Cierra sesión de un usuario en el sistema
systemTalkRec	Otros	No aplica	Permite realizar una interacción con el chatbot
findChatbot	Otros	No aplica	Encuentra el chabot en la lista de chatbots según su Id
processMessage	Otros	No aplica	Pasa el mensaje enviado y determina una respuesta
getCurrentFlow	Selectores	No aplica	Obtiene el flujo actual de un chatbot
matchMessageToOption	Otros	No aplica	Verifica si un mensaje es igual que una opción
updateSystemWithResponse	Modificador	No aplica	Actualiza el sistema con la respuesta del chatbot
setSystemChatHistory	Modificador	No aplica	Actualiza el historial del sistema

Tabla 5: TDA User

Nombre	Capa	Recursión	Descripción
user	Constructor	No aplica	Crea un usuario
getUsername	Selector	No aplica	Obtiene el nombre de un usuario

Tabla 6: TDA ChatHistory

Nombre	Capa	Recursión	Descripción
chatHistory	Constructor	No aplica	Crea el historial de chat
getChatHistory	Selector	No aplica	Obtiene el historial del chat