Wazelog

Greivin Carrillo Rodríguez

Fabián Castillo Cerdas

Jorge Guillén Campos

Instituto Tecnológico de Costa Rica

Profesor Marco Rivera Meneses

Paradigmas de programación

6 de octubre de 2023

Índice

Descripción de los hechos y reglas implementadas

Hechos

Arcos

Los hechos principales corresponden al establecimiento de conexiones entre dos lugares diferentes, en su mayoría tienen la misma estructura, por ejemplo:

unions(“paraiso”,”orosi”, 8).

Este hecho se leería como una ruta desde Paraíso en dirección a Orosi y con una distancia de 8 km.

Elementos sintácticos

Existen varios hechos referentes a los elementos sintácticos que conforman una oración o que podrían dar respuesta a las interrogantes que establezca el sistema experto, los negativos y afirmativos presentan varias formas de representarlos, no obstante, presentan la misma estructura en sus hechos.

negativo(["no"|S], S).

afirmativo(["si"|S], S).

Ambos tipos de hecho presentan una relación con la lista que contiene las entradas, en donde será negativo si se comprueba que justamente el primer elemento de la lista es “sí” o “no”, o también cualquiera de los otros hechos establecidos para responder afirmativa o negativamente. Esta revisión con el primer elemento de la lista para reconocer la presencia de una instancia específica se aplica también con los anteriores y posteriores elementos que se explicaran, en donde por lo general poseen la misma estructura al nivel del estudio de la lista.

Los pronombres poseen una serie de hechos que a nivel interno presentan una clasificación restringida de acuerdo con cada uno de ellos. Estos pronombres se diferencian principalmente entre singulares y plurales, no obstante, a la vez cuentan con las siguientes

clasificaciones: personales (“yo”, “nosotras”, “nosotros”), reflexivos (“me”, “nos”), posesivos (“mi”, “nuestro”) e impersonales (“se”).

pronombre(personal, plural,["nosotros"|S],S).

pronombre(reflexivo, singular,["me"|S],S).

Para los artículos la clasificación únicamente será de género (masculino o femenino) y de cantidad (singular o plural), ejemplos de estos son:

articulo(masculino, plural, ["los"|S], S).

articulo(femenino, singular, ["la"|S], S).

Dentro de los sustantivos y adjetivos únicamente se establecieron dos para cada uno de ellos:

sustantivo(masculino, singular,["destino"|S], S).

sustantivo(masculino, singular,["lugar"|S], S).

y

adjetivo(masculino, singular,["destino"|S], S).

adjetivo(masculino, singular,["instermedio"|S], S).

Ambos presentan la misma clasificación otorgada a los artículos.

En los verbos existen gran variedad de tipos en los que podrían ser escritos de acuerdo con la gramática española y sus diferentes conjugaciones tanto en tiempo verbal como a nivel de la persona, es por esto que de acuerdo con los requerimientos se diferencian principalmente en masculinos o femeninos, y estos a su vez pueden presentarse como copulativos, infinitivos, intransitivos, reflexivos y transitivos. Un ejemplo para cada uno de estos es:

verbo(copulativo, _, ["es"|S], S).

verbo(infinitivo, _, ["ir"|S], S).

verbo(intransitivo, singular, ["viajo"|S], S).

verbo(reflexivo, plural, ["encontramos"|S], S).

verbo(transitivo, singular, ["necesito"|S], S)

.

Finalmente, las preposiciones, estas en su primera y única clasificación se establecen como de finalidad, lugar o conexión:

preposicion(finalidad, ["para"|S], S).

preposicion(lugar, ["alrededor"|S], S).

preposicion(conexion, ["de"|S], S).

Ciudades, lugares y establecimientos

Estos tipos de hechos únicamente se encargan de la diferenciación entre las ciudades, lugares que podrían estar en dichas ciudades y los establecimientos, nuevamente lo que permiten es la verificación de la presencia de alguno de estos en la lista suministrada para corroborar que las oraciones sí muestran un sentido.

ciudad(["cartago"|S], S).

local(["walmart"|S], S).

establecimiento(["piscinas"|S], S).

Reglas

Las reglas existentes tienen una gran variedad de acuerdo con el contexto, en donde muchas poseen diferentes reglas para generar la estructura de reglas requeridas para desarrollar una tarea específica, por ejemplo, la regla *miembro* está presente en conjunto dos reglas más que poseen la misma cabeza de la regla. Por lo tanto, a continuación, solo se mostrarán las reglas más importantes y se ignorarán aquellas que pese a tener la misma cabeza tienen un cuerpo diferente.

Listas

En el caso de las listas se tienen dos reglas esenciales, *miembro* e *invertir*, la primera tiene como finalidad verificar que un elemento es miembro de una lista, mientras que invertir se encarga tal como lo indica su nombre de invertir una lista.

miembro(X, [X|_]):- !.

invertir(X,Y):- invertir(X,Y,[]).

Sintagmas y oraciones

Entre las reglas más importantes se encuentran los sintagmas, a través de estos se pueden determinar las partes de una oración y finalmente construirlas. Para la construcción de la oración los sintagmas aprovechan los hechos establecidos referentes a los elementos sintácticos, de esta forma se generan en primera instancia los sintagmas y finalmente las oraciones. Entre los principales sintagmas están:

sintagma_nominal(N, S0, S):- pronombre(posesivo, N, S0,S1), sustantivo(G, N, S1,S2), adjetivo(G, N, S2,S), S1\=S2.

sintagma_verbal(N,S0,S):- verbo(reflexivo, N, S0, S1), preposicion(finalidad, S1, S2), verbo(infinitivo, N, S2, S3), sintagma_preposicional(N,S3,S).

sintagma_preposicional(N, S0, S):- preposicion(lugar,S0,S1), preposicion(conexion,S1,S2), sintagma_nominal(N,S2,S).

Por otra parte, están directamente las oraciones:

oracion(S0,S):- sintagma_nominal(N,S0,S1), sintagma_preposicional(N,S1,S).

oracion(S0,S):- sintagma_nominal(_,S0,S1), sintagma_verbal(_,S1,S2), sintagma_preposicional(_,S2,S).

Verificación de elementos sintácticos y procesamiento de entradas y salidas

Estas reglas son las encargadas de verificar que una entrada sea una oración, esto implica que se deben crear reglas que permitan la comprobación de afirmaciones, negaciones, ciudades, entre otros. Ejemplos representativos de estas reglas son:

es_oracion(S):- oracion(S, []), !.

es_negativo(N):- negativo(N,[]), !.

es_ciudad(Ciudad):- miembro(Ciudad, C), ciudad(C, []), !.

Para verificar que una entrada completa está construida con la gramática establecida se construye una regla que se basa a su vez en la verificación de oraciones para corroborar que la entrada sí es una oración con sentido, en caso de que no el sistema experto indicaría su no entendimiento.

validacion_entrada(Input):- validacion_entrada_aux(Input),!.

De la misma forma y prácticamente con la misma estructura se construyen reglas que verifican locales, ciudades, o lugares en general, los cuales a su vez cada uno cuenta con su mensaje de advertencia respectivo cuando la entrada ingresada no corresponde a algún hecho reconocido.

error_local:- writeln('\n WazeLog: Disculpe, aun no conozco ese local.\nÂ¡Por favor ingrese uno valido!').

Una vez que se desea dirigir al usuario una salida de texto se toma en consideración que los datos no son strings, sino átomos, y que cada uno de ellos debe convertirse y ser colocado en una nueva lista,

atomo_a_string([X|L1], L2, L3):-

downcase_atom(X, Y),

text_to_string(Y, String),

atomo_a_string(L1, [String|L2], L3).

Además de esto también deben eliminarse los elementos de puntuación que se encuentren en la lista de tal modo que al brindar la salida el mensaje solo contenga lo justamente necesario, para esto se requiere una regla que procese la lista eliminando estos signos una vez que ya se haya convertido a una lista de strings.

eliminar_puntuacion(X, S5):-

delete(X, ",", S1),

delete(S1, ".", S2),

delete(S2, "!", S3),

delete(S3, ";", S4),

delete(S4, "?", S5).

parseToList(X,W):-

atomo_a_string(X, Y),

eliminar_puntuacion(Y, Z),

invertir(Z,W).

Búsqueda de rutas

El algoritmo de búsqueda será explicado con mayor detalle en una próxima sección, no obstante, se mostrarán las reglas más significativas.

f*indminpath_t(X, W, T, P) :- findminpath_interm(X, W, P), T is W * 2.*

Esta es la principal que se encarga de determinar el camino más corto entre el origen y el destino, como es observador esta depende de *findminpath_interm* y esta a su vez de otras, pero el conjunto es el encargado de establecer todas las rutas y con estas calcular la más corta.

Representación de elementos en pantalla

La regla principal se denomina start(), esta despliega todos los mensajes para atender al usuario a medida que este va respondiendo las preguntas, finalmente le muestra al usuario cuál es la ruta que debería de tomar según todos los destinos intermedios así como la distancia total. Start nunca para, sino que incluso tras finalizar una ruta volverá a preguntar al usuario su destino y origen.

```
start():-
writeln('\nWazeLog: Bienvenido a WazeLog la mejor logica de llegar a su destino!'),
writeln('        Por favor indiqueme donde se encuentra.'),
ubicacion(Inicio),
 writeln('\nWazeLog: Muy bien, ¿Cual es su destino?'),
ubicacion(Destino),
writeln('\nWazeLog: Exelente, ¿Tiene algun destino intermedio? (si / no)'),
intermedio([], List),
nl,
%Realizar una sola lista.
append([Inicio],List,L1),
append(L1,[Destino],L2),
getPath(L2,P,W,T),
writeln("------------------------------------------------------------"),
writeln(P - W - T),
writeln("------------------------------------------------------------"),
start.
```

Descripción de las estructuras de datos desarrolladas

La estructura de datos desarrollada es un grafo construido a partir de listas, cada sublista posee una representación al nivel del recorrido. En el siguiente ejemplo se estudiará más.

[[[paraiso, [orosi, 8]], [orosi, [cachi, 12]]]]

Cada sublista representa un recorrido, el primer elemento indica el origen, como “paraíso”, mientras que la sublista de la sublista [orosi, 8] indica tanto el destino como la distancia en kilómetros, de esta forma la segunda sublista inicia en el último destino recorrido y se dirigirá a otro destino en caso de que se indique, en este caso hacia “cachi” con una distancia de 12 km, si en este punto finalizara el recorrido se brindaría toda la secuencia desde el primer origen hasta los otros destinos, omitiendo la repetición dado que el destino anterior corresponde al nuevo origen, además, se suman las distancias, para este caso un total de 20 km.

Descripción de los algoritmos desarrollados

El algoritmo de búsqueda diseñado para encontrar la ruta más corta parte primeramente de encontrar los diferentes caminos entre dos lugares. La regla con la que se calculan las distancias entre dos nodos es findapath, la cual toma el nodo de origen y destino y corrobora con los hechos prestablecidos que sí exista una ruta entre esos lugares. A su vez existe una regla con la misma cabeza de regla que se encarga de realizar la misma tarea, pero con algún nodo intermedio, por lo que obtendrá como resultado el peso total, así como una lista de los lugares que fueron recorridos.

La regla findminpath se comporta como una variación del algoritmo de Dijkstra, para este caso utilizado para la búsqueda del camino más corto entre dos lugares dentro del grafo Utilizando findapath se realiza una búsqueda de todos los caminos posibles de un nodo a otro, luego estos son almacenados temporalmente en solution. A diferencia del algoritmo Dijkstra, findminpath evalúa cada una de las posibles rutas, sacrificando eficiencia, sin embargo, fue necesario realizarlo de esta manera para asegurarse que la ruta calculada sea en definitiva la más corta, pues findapath no muestra las rutas ordenadas y así como la ruta más corta podría ser la primera solución, también podría tratarse de la última. Se dice que son almacenados temporalmente en solution ya que se debe mantener de forma dinámica la ruta más corta encontrada hasta el momento.

La regla findminpath_t es la que de forma general aprovechará las otras reglas para establecer todas las rutas, seleccionar la más corta y establecer la distancia total recorrida. Con la distancia total se calcula el tiempo, el cual en este caso será el doble de la distancia, pero en minutos.

Problemas sin Solución

1. Interfaz de Usuario Limitada: El proyecto carece de una interfaz de usuario amigable. Esto hace que la interacción con la aplicación sea complicada para usuarios no familiarizados con Prolog y la programación lógica.
2. Dependencia de Swi-Prolog: La aplicación está fuertemente vinculada a SWI-Prolog, lo que limita su portabilidad a otros entornos o lenguajes de programación. Esto podría dificultar la adopción por parte de un público más amplio.
3. Manejo de Errores Deficiente: La aplicación carece de un manejo de errores sólido. Cuando los usuarios ingresan entradas incorrectas o se producen errores en tiempo de ejecución, la aplicación puede bloquearse o proporcionar mensajes de error confusos.
4. Optimización de Búsqueda Ineficiente: El algoritmo de búsqueda de rutas más cortas podría mejorarse en términos de eficiencia y velocidad, especialmente para redes de ciudades más grandes.
5. Falta de Pruebas Unitarias y Automatizadas: La falta de pruebas unitarias y pruebas automatizadas dificulta la identificación temprana de errores y la realización de cambios en el código sin introducir nuevos problemas.
6. Complejidad de Gramática: La gramática utilizada para analizar las consultas de los usuarios es complicada y limita la comprensión y extensión de la funcionalidad del lenguaje natural.

Problemas solucionados

1. Desarrollo del Paradigma de Programación Lógica: Se logro desarrollar una aplicación que demuestra un buen entendimiento del paradigma de programación lógica, aplicando conceptos y técnicas de Prolog de manera efectiva.

2. Interacción con Prolog: La aplicación es capaz de interactuar con los usuarios a través de consultas escritas en lenguaje natural y responder adecuadamente, lo que implica una implementación exitosa del motor de Prolog.

3. Estructura de Datos: Se implemento una estructura de datos basada en listas de Prolog para representar las ciudades, localizaciones y establecimientos, lo que demuestra la habilidad para crear y manipular estructuras de datos en Prolog.

4. Cálculo de Rutas Más Cortas: Se logro calcular rutas más cortas entre ciudades, lo que es una característica esencial del proyecto. Esto demuestra la correcta implementación del algoritmo de búsqueda de rutas.

5. Gramática y Análisis de Lenguaje Natural: Se implemento una gramática y un analizador para comprender las consultas de los usuarios escritas en lenguaje natural. Esta es una característica clave del proyecto y demuestra tu capacidad para trabajar con análisis de lenguaje natural en Prolog.

Actividades a realizar por estudiante

Actividad	Encargado			Fecha
	Greivin	Fabian	Jorge	
Dividir el proyecto	x	x	x	18-sep
Definir estructuras de datos	x	x	x	
Investigacion sobre BNF		x		20-sep
Investigacion sobre SE			x	
Creacion de base de datos		x	x	22-sep
Creacion de grafo	x			25-sep
Implementacion parser, BNF, SE.	x	x	x	26-sep
Correccion de errores		x	x	2-oct
Implementar solucion final		x		
Hacer manual de usuario		x		4-oct
Hacer la documentacion	x		x	
Revision final	x	x	x	6-oct

Conclusiones

WazeLog se ha desarrollado con el objetivo de reforzar el conocimiento del paradigma de programación lógica. A través de la creación de una aplicación que actúa como un Sistema Experto utilizando Prolog.

Uno de los principales logros de este proyecto es la aplicación de los conceptos de programación lógica en un entorno práctico. WazeLog utiliza Prolog de manera efectiva para procesar consultas del usuario, buscar información en una base de datos de ubicaciones y proporcionar respuestas precisas. Esto ha demostrado la viabilidad y utilidad de la programación lógica en aplicaciones del mundo real.

La creación y manipulación de listas como estructuras de datos es fundamental en la funcionalidad de WazeLog. El proyecto ha demostrado la versatilidad de las listas en Prolog y cómo pueden ser utilizadas para representar datos complejos y realizar búsquedas eficientes.

El desarrollo de WazeLog ha sido un ejercicio exitoso para consolidar el conocimiento del paradigma de programación lógica. La aplicación no solo cumple con su propósito de proporcionar respuestas relacionadas con ubicaciones, ciudades y rutas, sino que también sirve como un ejemplo valioso de cómo aplicar conceptos de programación lógica en el mundo real. Este proyecto demuestra que la programación lógica es una herramienta poderosa y versátil que puede ser utilizada para crear aplicaciones prácticas y eficientes.

Recomendaciones

Durante el desarrollo de WazeLog, se han identificado algunas recomendaciones y áreas para futuras actualizaciones que pueden mejorar la aplicación y la experiencia del usuario:

1. Interfaz de Usuario Mejorada: Actualmente, WazeLog se ejecuta en una interfaz de línea de comandos. Una futura actualización podría incluir una interfaz gráfica de usuario (GUI) más amigable y visualmente atractiva. Esto facilitaría la interacción de los usuarios con la aplicación y haría que sea más accesible.

2. Mejora de la Base de Datos: Continuar expandiendo la base de datos de ciudades, lugares y establecimientos es fundamental para garantizar que WazeLog sea relevante y útil en múltiples contextos. Esto puede incluir la incorporación de datos geoespaciales más precisos y completos.

3. Optimización del Rendimiento: A medida que la base de datos crezca, será importante optimizar el rendimiento de las consultas y búsquedas para que la aplicación sea más rápida y eficiente. Esto podría incluir la implementación de índices o técnicas de búsqueda más avanzadas.

4. Mejora de la Validación de Entradas: Actualmente, WazeLog realiza validaciones de entrada básicas. En futuras actualizaciones, se podría implementar una validación más sofisticada para manejar una variedad más amplia de consultas y preguntas de los usuarios.

5. Documentación Mejorada: Continuar mejorando la documentación del proyecto, incluyendo manuales de usuario y técnicos, para facilitar su uso y desarrollo futuro.

6. Pruebas Extensivas: Realizar pruebas extensivas y depuración para garantizar que la aplicación funcione de manera confiable en una variedad de situaciones.

7. Colaboración y Contribuciones: Abrir el proyecto a la colaboración de la comunidad y considerar la contribución de otros desarrolladores para enriquecer la aplicación y expandir su conjunto de características.

Bibliografía

1. Bratko, I. (2012). Prolog Programming for Artificial Intelligence. Pearson.
2. Clocksin, W. F., & Mellish, C. S. (2003). Programming in Prolog. Springer.
3. Pereira, F. C. N., & Shieber, S. M. (1987). Prolog and Natural Language Analysis. Center for the Study of Language and Information.
4. Jurafsky, D., & Martin, J. H. (2020). Speech and Language Processing (3rd ed.). Pearson.
5. Allen, J. (1995). Natural Language Understanding. Pearson.
6. Manning, C. D., Raghavan, P., & Schütze, H. (2008). Introduction to Information Retrieval. Cambridge University Press.
7. Russel, S. J., & Norvig, P. (2009). Artificial Intelligence: A Modern Approach. Pearson.
8. Prolog SWI (2021). SWI-Prolog Documentation. [https://www.swi-prolog.org/]
9. SWI-Prolog (2021). SWI-Prolog FAQ. [https://www.swi-prolog.org/FAQ/]
10. Stack Overflow (Various Dates). Stack Overflow Prolog Questions. [https://stackoverflow.com/questions/tagged/prolog]

Bitácora en digital

Greivin:

- Se hizo una reunión para dividir las partes del trabajo y para declarar como ir trabajando. (18 septiembre)
- Cree un plan de actividades a realizar para el proyecto. (20 agosto)
- Investigue sobre algoritmos de búsqueda implementados en ProLog y sus diferencias con el implementado en Racket. (20-21 septiembre)
- Cree un grafo base para probar el pathfinder. (25 septiembre)
- Cree las reglas para encontrar una ruta entre 2 nodos dados. (26 septiembre)
- Optimice las reglas del pathfinder para que ahora muestre la ruta más corta. (27 septiembre)
- Actualice el grafo con el dado por el profesor. (30 septiembre)
- Trabaje en la documentación. (4 octubre)
- Se hizo una reunión para finalizar detalles y revisar el código final. (6 octubre)

Fabian:

- Se realizo una reunión inicial para definir la metodología del trabajo. (18 septiembre)
- Investigue sobre BNF y su implementación en ProLog. (19-21 septiembre)
- Cree una base datos general con las palabras que conforman las oraciones. (22 septiembre)
- Cree las reglas para comprobar la validez de una oración. (22 septiembre)
- Trabaje en la parte de la interfaz que utilizara el usuario para comunicarse con el programa. (24 septiembre)
- Implemente el parser que divide las oraciones en átomos para posteriormente validar. (25 septiembre)
- Solucione algunos bugs menores con respecto a respuestas del programa. (29 septiembre)
- Realice la sección de recomendaciones y conclusiones para la documentación. (4 octubre
- Realice el manual de usuario. (4 octubre)
- Se realizo una reunión para finalizar el código y concluir con la solución del proyecto. (6 octubre)

Jorge:

- Nos reunimos para dividir el trabajo y como trabajarlo. (18 septiembre)
- Investigue sobre los sistemas expertos. (21 septiembre)
- Hice una base de datos con lugares de interés que podría mencionar el usuario a la interfaz. (22 septiembre)
- Trabaje en la validación de palabras claves del sistema experto. (23-25 septiembre)
- Implemente las reglas para buscar palabras claves en la base de datos. (23-25 septiembre)
- Hice la parte de los algoritmos y estructuras de bases de la documentación. (4 octubre)
- Nos reunimos para revisar el código. (6 octubre)