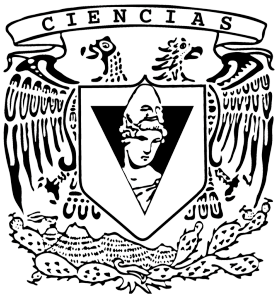

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
FACULTAD DE CIENCIAS

Programación lógica, aplicaciones y resolución de problema especifico

Alumnos:
Mauricio Comas
Benjamín Martinez
Medina Peralta Joaquín

Fecha de Entrega: 9 de junio de 2024

Abstract

En este proyecto, tenemos como objetivo el investigar acerca de la programación lógica y sus aplicación en la industria, como uno de los que se profundizara, como en la lingüística computacional con el uso de una plataforma de código abierto para la traducción automática.

A su vez, se hablara de la resolución de un problema propuesto, en el cual dado un laberinto y quesos en cada espacio, un ratón debe de buscar la salida, pero teniendo el problema que si come un queso envenenado muere. Este problema, se modelo por medio de la lógica de predicados y en especial, el uso de PROLOG como lenguaje de programación lógico. De esta manera, se muestran los resultados de la investigación, como de la implementación del código.

Preliminares

Programación Lógica

Conocida como uno de los paradigmas mas famosos de la programación declarativa o programación inferencial la programación lógica se basa en fragmentos de la lógica de predicados usando las clausulas de Horn como base del lenguaje de programación, nos fijamos en las clausulas de Horn ya que su semántica operacional permite que su implementación sea mas eficiente, pues nos permite crear una semántica declarativa por teoría de modelos que toma al modelo como la interpretación de un universo de discurso puramente sintáctico con el que empleando su sistema de resolución SLD permite probar por medio de refutación usando el algoritmo de la unificación, permitiendo obtener respuestas dada una base de conocimientos(Nuestro universo de discurso), el mecanismo de computo de este estilo de programación permite tener una búsqueda de resultados indeterminista, es decir dado el contexto tomar varios caminos que lleven a varias soluciones (Todas las posibles) dependiendo del contexto, además permite computar con datos parcialmente definidos pues el computo en este estilo de programación se centra en la inferencia de la lógica que formamos en nuestra base de conocimientos. Informacion recuperada de [Pascual Julían Irazno, 2007]

Aplicaciones de la Programación Lógica

Una de las aplicaciones que estaremos hablando en este trabajo, es la ayuda para la traducción automática la cual es usada en la lingüística computacional. Como se menciona en el articulo [ARMENTANO-OLLER, 2007], los sistemas de traducción automática son encargados de traducir un texto escrito a un idioma de origen a otro idioma destino, donde hay varias técnicas principalmente dos enfoques:

- Sistemas basados en reglas: son basados en reglas más habituales basados en tres fases: análisis que se produce en la frase de lengua de origen a una representación intermedia dependiente de la lengua de origen, transferencia a una nueva representación dependiente a la lengua destino, y por último una generación a una frase en lengua destino por medio del resultado anterior.

- Sistemas basados en corpus: uso de cadenas o patrones inferidos automáticamente a partir de las regularidades que contienen documentos en la lengua origen y correspondiente a la lengua destino.

En este caso, haremos más énfasis del primer sistema, debido a que, como se menciona en el artículo [ARMENTANO-OLLER, 2007], es el principal enfoque de Apertium, la cual es una plataforma de código abierto que incluye herramientas de para la traducción automática basada en reglas y en módulos estadísticos. La parte que nos interesa, es su motor de traducción automática, consiste en 8 subsistemas interconectados.

Como breve mención, según [ARMENTANO-OLLER, 2007], algunos de sus subsistemas son: desformateador, analizador morfologico, desambiguador léxico categorial, módulo de transferencia léxica, módulo de transferencia estructural, generador morfológico, postgenerador y deformateador.

Lo que nos interesa en este trabajo, son los módulos de analizador morfológico, transferencia léxica, transferencia estructural y el generador morfológico. Debido a que estos,

ocupan reglas de inferencia para obtener información relevante (la forma base para consulta de diccionarios y categoría léxica) de la cual el programa utiliza para obtener su traducción al lenguaje destino. Aunque no se ocupe directamente PROLOG, debido a que utilizamos una generación de una gramática, sintaxis y semántica con nuestras propias reglas de inferencia, es posible considerar el funcionamiento de Aparetium como programación lógica.

De esta manera, esta aplicación esta ayudando en uno de los problemas clásicos de la computación el cual es la traducción automática, usando reglas de inferencia y gramáticas dependiendo de la lengua. Y como se menciona en [ARMENTANO-OLLER, 2007], dado que Aparetium es de código libre, se busca que la comunidad ayude en la mejora del software, donde se puede utilizar PROLOG de manera directa para mejores inferencias al momento de traducir, así como darle soporte a más lenguas. A su vez, como igual se menciona en [ARMENTANO-OLLER, 2007], el margen de error de los pares de lenguas disponibles están entre el 5% y el 10%.

Implementación

Análisis y descripción del Problema

EL problema planteado consiste en lo siguiente: Considere una caja cuadriculada rectangular de dimensiones NxM, donde sus cuadriculas pueden estar vacías o tener un pedazo de queso (puede contener ron o veneno). Dicho esto, se coloca el ratón en alguna cuadricula y con cierta dirección, el cual debe de buscar la salida, pero al encontrar un pedazo de queso puede suceder los siguientes casos:

- Si el queso no tiene veneno ni ron, se lo come y se mueve en la misma dirección.
- Si el queso tiene ron se lo come y se emborracha, provocando que avance 7 pasos en dirección aleatoria.
- Si el queso esta envenenado, se lo come sólo si está borracho y muere.
- Si el ratón esta sobrio, no come el queso y avanza en la misma dirección.
- Si el ratón llega a la pared de la caja y no puede moverse más en la dirección actual, entonces: Si esta sobrio, gira a la izquierda y sigue caminando, en otro caso, si esta borracho, choca con la pared hasta que se le pase la borrachera.

Dicho lo anterior, pensemos en la solución del problema dado. Primero necesitamos definir el estado del ratón, con respecto a su posición, dirección y si esta borracho o no (esto debido a que no tenemos más efectos dentro del problema y dado que son 7 pasos que dura el estado, se utiliza un número desde 7 a 0). Así tenemos que:

$$raton(X, Y, D, B) \longleftarrow D = \{arriba, abajo, izquierda, derecha, muerto\}, B = 0 \leq 7$$

A su vez, tenemos las reglas direcciones, dadas por lo siguiente:

$$direccion(derecha, 1, 0), direccion(izquierda, -1, 0),$$
$$direccion(arriba, 0, -1), direccion(abajo, 0, 1).$$

De esta manera, empecemos nuestro código con la parte del movimiento del ratón, de este modo, chequemos el siguiente método:

```
% Direcciones
direccion(derecha, 1, 0).
direccion(izquierda, -1, 0).
direccion(arriba, 0, -1).
direccion(abajo, 0, 1).

% Avanzar al raton dependiendo de su direccion
avanza(X, Y, arriba, X, NY) :- NY is Y - 1.
avanza(X, Y, abajo, X, NY) :- NY is Y + 1.
avanza(X, Y, izquierda, NX, Y) :- NX is X - 1.
avanza(X, Y, derecha, NX, Y) :- NX is X + 1.

% Girar a la izquierda
girar_izquierda(derecha, arriba).
girar_izquierda(arriba, izquierda).
girar_izquierda(izquierda, abajo).
girar_izquierda(abajo, derecha).

% Movimiento en una direccion aleatoria
random_direction(D) :-
    random_between(1, 4, R),
    (R = 1 -> D = derecha;
     R = 2 -> D = izquierda;
     R = 3 -> D = arriba;
     R = 4 -> D = abajo).
```

Listing 1: Movimiento del ratón

Con esto podemos construir el movimiento del ratón por medio de la caja, en donde revisamos el avance de la coordenada del ratón, verificamos como actúa el ratón al momento de girar a la izquierda (esto debido a la vista del ratón) y una dirección aleatoria (se utiliza al momento de que el ratón este borracho).

Con estas funciones, podemos construir las siguientes tres funciones:

```
% Indica si el raton esta pegado a una pared y su direccion es hacia
   esa pared
viendo_pared(Raton) :-
    Raton = raton(X, Y, D, _),
    dimensiones(N, M),
    LN is N - 1,
    LM is M - 1,
    (
        D = derecha, X = LN;
        D = izquierda, X = 0;
        D = arriba, Y = 0;
        D = abajo, Y = LM
    ).
```

Listing 2: Movimiento del ratón, al momento de encontrar una pared

En esta función estaremos simulando cuando encontremos una pared, donde dependiendo de la dirección, colocamos una nueva posición del ratón el cual se dará por la dirección. Así, definimos X y Y con una variable auxiliar que sera la sustitución de alguno de ellos.

```
% (si esta en direccion de pared y no borracho) gira_izquierda hasta
   que la condicion no se cumpla
% (si no esta en direccion de pared y borracho) gira aleatorio
```

```
% Mantiene todo igual en otros casos
revisa_direccion(Raton, NRaton) :-
    Raton = raton(X, Y, D, B),
    (
        viendo_pared(Raton), B = 0 -> girar_izquierda(D, ND),
            revisa_direccion(raton(X, Y, ND, B), NRaton);
        not(viendo_pared(Raton)), B > 0 -> random_direction(ND), NRaton
            = raton(X, Y, ND, B);
        NRaton = Raton
    ).
```

Listing 3: Movimiento del ratón, al momento de revisar la dirección

Con esta función estaremos revisando en cada momento la dirección del ratón, donde se tiene dos posibles casos, cuando esta viendo la pared, se gira a la izquierda y se revisa de nuevo la dirección con la nueva, mientras que el segundo caso, en dado caso que este borracho, se selecciona una nueva dirección en aleatorio y se actualiza el estado, por último, si no pasa ningún caso, se mantiene el estado con el nuevo.

```
% (si esta en direccion de pared y borracho) se mantiene todo y
    borracho -= 1
% (si borracho) avanza y borracho -= 1
% avanza en otros casos
mover(Raton, NRaton) :-
    Raton = raton(X, Y, D, B),
    NB is B - 1,
    avanza(X, Y, D, NX, NY),
    (
        viendo_pared(Raton), B > 0 -> NRaton = raton(X, Y, D, NB);
        B > 0 -> NRaton = raton(NX, NY, D, NB);
        NRaton = raton(NX, NY, D, B)
    ).
```

Listing 4: Movimiento del ratón

Por último, para el movimiento necesitamos calcular el nivel de borrachera del ratón y con avanza, calculamos la nueva posición del ratón. Dicho esto, en caso que este viendo la pared y su nivel de borrachera sea mayor a 0, no cambiamos de dirección pero si decrementamos su nivel. De otro modo, seguirá avanzando aunque este borracho (decrementara su nivel).

Con esto, hemos construido el movimiento general del ratón incluyendo el caso cuando el ratón esta borracho, pero no es lo único que nos debemos de fijar, ya que uno de los desafíos de este problema es revisar el estado del ratón y el contenido de cada casilla. Por lo tanto, tendremos la siguiente parte del código:

```
% Cambia el estado del tablero (si hay algun queso)
% Cambia su contador borracho a 7 (si hay ron)
% Cambia direccion a muerto (si borracho -= 0 y hay veneno)
% Cambia direccion a salida (si llego a salida)
% Mantiene direccion, contador y estado en otros casos
revisa_celda(Raton, NRaton) :-
    Raton = raton(X, Y, D, B),
    celda(X, Y, TipoCelda),
    (
        TipoCelda = queso_con_ron -> NRaton = raton(X, Y, D, 7), write(
            'Comiendo queso con ron.'), nl;
        TipoCelda = queso_con_veneno, B > 0 -> NRaton = raton(X, Y,
            muerto, B), write('Comiendo queso con veneno.'), nl;
```

```
12          TipoCelda = salida -> NRaton = raton(X, Y, salida, B), write('
                Sobre la salida.'), nl;
13          NRaton = Raton
14      ).
```

Listing 5: Revisar celda

Con este predicado, podemos revisar la celda y actualizar el estado del ratón, donde si hay un queso con ron, entonces el contador del estado de borracho inicia en 7. Si la celda contiene queso con veneno y el ratón esta borracho, es decir, el estado es mayor que cero, entonces come el queso y muere el ratón. Si es la celda de salida, entonces se cambia el estado a salida y termina el proceso. Por último, si no sucede estos casos, se mantiene igual el ratón.

```
1  % Acciones del raton en cada paso
2  accion_raton(Raton, NRaton) :-
3      % Quita el queso de las celdas y cambia el estado del raton
           dependiendo del queso que habia en la celda
4      revisa_celda(Raton, QRaton),
5      QRaton = raton(_, _, Final, _),
6      (
7          Final = salida -> NRaton = QRaton;
8          Final = muerto -> NRaton = QRaton;
9          % Si no ha terminado hace las siguientes dos acciones
10         % Cambia la direccion del raton dependiendo si esta en pared o
               borracho
11         revisa_direccion(QRaton, DRaton),
12         % Mueve al raton
13         mover(DRaton, NRaton)
14     ).
```

Listing 6: Accion del raton

Por último, tendremos la acción del ratón el cual revisamos la celda, donde se guarda el estado final, y dependiendo de su estado, se cambia el ratón. En el caso que el estado final sea ”salida” o ”muerto”, entonces se mantiene el nuevo ratón. En otro caso, revisamos la dirección y movemos al ratón.

Hallazgos y Desafíos

Aunque se haya mencionado la forma que se construyo el código de manera sencilla, uno de los desafíos fue el abstraer la información del problema, debido a que se considera varios casos y estados al momento de que el ratón avance y revise el contenido de la celda. De este modo, fue uno de los elementos que costo trabajo.

Retomando el punto anterior, otro problema es considera los posibles casos de avance de ratón y el contenido de la celda, ya que al no cuidar bien estos detalles, es posible que el programa no cumpla con el propósito principal.

A su vez, un problema con las celdas, es no actualizar su información, debido a que puede suceder que se termine realizando un bucle infinito, de este modo, dejando libre la posibilidad de que el programa con ciertas condiciones, deje de funcionar. Y por último, complementario a lo anterior, el mantener un contador de estado de ebriedad del ratón, fue necesario mantener dentro predicado ratón, una variable con lo cual se pueda manejar, debido al caso del queso con veneno.

Conclusiones

Por medio de la investigación realizada y el código hecho el equipo llego a la conclusión de que el paradigma de la programación lógica es una herramienta bastante útil y poderosa, pues es en parte importante una de las razones de cambio de las nuevas tecnologías, consideramos que es un paradigma distinto a el que se asocia comúnmente a la programación que es el paradigma orientado a objetos y es por esto que puede resultar difícil para programadores que nunca habían trabajado con el paradigma aprenderlo e implementarlo.

En cuanto al programa resulto complicado la recepción de celdas para crear el tablero, así que optamos por crear un tablero de 5x5 default en el que el ratón se mueva, con celdas modificables desde el programa. La posición de inicio también puede ser modificada en el programa, por esta razón no resulta tan amigable el programa comparado a otros pero en rendimiento el programa muestra las celdas y ejecuta correctamente los escenarios. Aunque, se dejo una versión preliminar de interfaz de usuario, para colocar la información manual de cada celda y de la posición del ratón.

Por lo tanto, para programas que se necesite inferir ciertas reglas para que se busque "pensar" la computadora, este paradigma de la programación es bastante útil, y tan solo es una de las herramientas fundamentales para acércanos a la traducción automática entre dos lenguas.

Repositorio de Git.

Este proyecto, el código default de la implementación y del prototipo de interfaz de usuario, se encuentran en un repositorio de github, donde se encuentran las indicaciones de compilación y ejemplo de como ejecutar el segundo programa.

Link al Repo del proyecto.

References

[ARMENTANO-OLLER, 2007] ARMENTANO-OLLER, Carme, e. a. (2007). Apertium, una plataforma de código abierto para el desarrollo de sistemas de traducción automática. *Proceedings of the FLOSS International Conference 2007*. Recuperado de http://hdl.handle.net/10045/27531.

[Pascual Julían Irazno, 2007] Pascual Julían Irazno, M. A. F. (2007). *Programacion logica Teoría y Practíca*. PEARSON.