Wazelog

Ingeniería en Computadores

CE1106-Paradigmas de programación

Grupo 01

Profesor:
Marco Rivera Meneses

Estudiante:
Andrés Molina Redondo | 2020129522
Ignacio Lorenzo Martínez | 2019068171
Luis Alfredo Gonzáles Sánchez |2021024482

Semestre 2, 2023

Índice

Descripción de los hechos y reglas implementadas

Parser y lógica

Descripción de las reglas y los hechos necesarios para el parser y la lógica de las respuestas del programa:
Para el caso del parser existen reglas y hechos menores necesarios que se explicaran a continuación:
-maximo(3).
-decision([no]).

Los hechos anteriores son necesarios para el funcionamiento del programa, para la solución planteada por los estudiantes van a existir como un máximo de 3 nodos intermedios, por lo que se consideró necesario crear un hecho llamado máximo que verifique si el usuario ya inserto 3 destinos intermedios, de ser así retorna la ruta con estos 3 destinos intermedios
Por otra parte el hecho decisión determina si el usuario decide si quiere más destinos intermedios o no, para no tener que estar buscando la palabra clave “no” y buscar la palabra clave “si” , se maneja de que el usuario diga no si no quiere más destinos intermedios y que mencione hacia donde debe de pasar, para indicar un “si” , de esta forma se evita verificar el “si “ mientras que ya se contiene una respuesta insertada por el usuario.

Por otra parte, existen hechos que funcionan como destinos generales o “ambiguos” que obligarán al usuario a mencionar donde se encuentra dicho destino si el usuario no lo ha mencionado , estos hechos deben de ser ampliados para cubrir una vasta gama de posibilidades introducidas por el usuario, esto es un limitante que el equipo de trabajo considera, ya que la respuesta dependerá del conocimiento de la base de datos sobre un lugar en específico, entre esos lugares generales :
lugar(ciudad).
lugar(pueblo).
lugar(panaderia).
lugar(supermercado).
lugar(gasolinera).
lugar(pulperia).
lugar(universidad).

Ahora , se van a mencionar las reglas elaboradas para el funcionamiento de este programa, se va a empezar por las básicas , llegando a las más complejas.

concatenar([],L,L).
concatenar([X|L1],L2,[X|L3):-concatenar(L1,L2,L3).

La regla anterior permite concatenar una lista con otra, esta regla es utilizada cuando se determinan los destinos intermedios y con esto construye la lista que será enviada al algoritmo de dijkstra.Ahora , observe la regla que engloba las demás reglas:

```
leer([]):-write('bienvenido a wazelog, su
navegador fiable.Por favor digame donde se encuentra:'),nl,
 read_line_to_string(user_input, String), tokenize_atom(String,
Lista_result),%consulta el origen
 consulta_inicial(Lista_result,Result1),
 write('Excelente, estamos en: '),write( Result1),nl,write('favor digite su destino'),nl,
 read_line_to_string(user_input, String2), tokenize_atom(String2,
Lista_result2),%consulta el destino.
 consulta_inicial(Lista_result2,Result2),nl,write('Perfecto, vamos hacia
'),write(Result2),nl,
 write('Existe algun destino intermedio?, favor digite no en caso de que no haya, en
caso contrario favor digite el destino intermedio'),nl,
 read_line_to_string(user_input, String3), tokenize_atom(String3, SIONO),
 tomar_decision(SIONO,Lista_resultado),
 concatenar([Result1],Lista_resultado,X),
 concatenar(X,[Result2],Y),
 write('su resultado'),nl,write(Y).
 %write(Lista_resultado).
```

La regla anterior es la regla que funciona como la lógica del programa, si usted nota, esto en realidad es una regla que hace llamado a reglas menores , lo que permite resolver los problemas especificados, y contiene un flujo de consulta y respuesta del usuario.

Se describirán las reglas y sus hechos asociados a continuación:

```
consulta_inicial(Lista,Pclave):-oracion(Lista,[]),destino_general(Lista,Pclave),!. %caso 1, la palabr
consulta_inicial(Lista,Pclave):-oracion(Lista,[]),ubicaciones(Lista,Pclave),!.%caso 2, la palabra cla
consulta_inicial(Lista,Pclave):- write('no entendi a que se referia con'),nl,write(Lista),nl,
    write('favor redigite su consulta de otra manera'),nl,
    read_line_to_string(user_input, X), tokenize_atom(X, Lista2),
    consulta_inicial(Lista2,Pclave). %esto deberia de estar bien.
```

La regla anterior lo que hace es consultar por el origen y el destino, si no logra encontrar ya sea un lugar general , o un lugar en concreto , envía el mensaje de que no entendió y trata de procesar nuevamente la consulta del usuario.De parametros recibe una lista que es el input del usuario y una palabra clave de la cual hará la deducción. La línea “oración” es la verificación de que el bnf se cumple.
Primera línea: consulta por un destino general.
Segunda línea: consulta y encuentra el destino intermedio

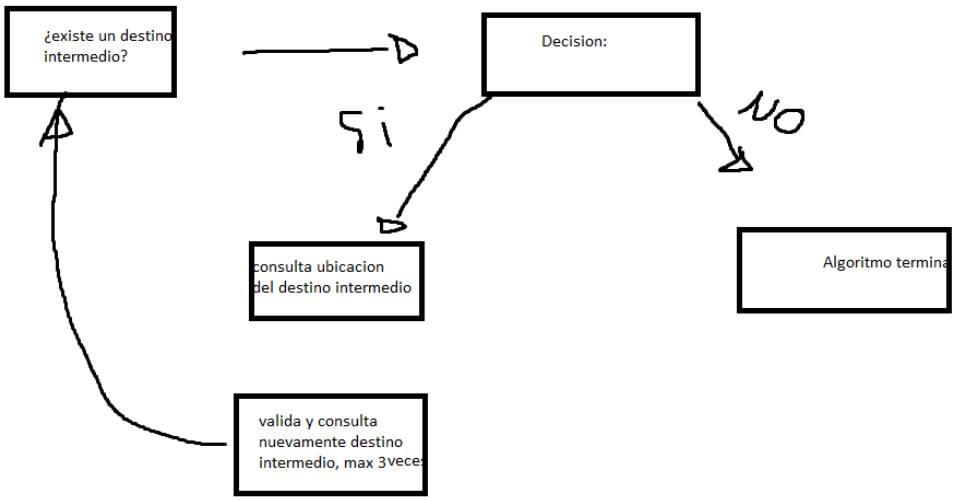
Tercera línea, consulta nuevamente al usuario por un input distinto para entenderle. estas líneas:

```
read_line_to_string(user_input, X), tokenize_atom(X, Lista2),
consulta_inicial(Lista2,Pclave). %esto deberia de estar bien.
##########################################################################################3
```

read_line_to_String lee el input del usuario y lo convierte a string, tokenize_atom, convierte dicho string a una lista

```
%CONSULTAS DE DESTINOS INTERMEDIOS##############################:
procesar_consulta(Lista,Lista2,NUMBER):-oraciones(Lista,Pclave),consultarlugar(Pclave,Lista2,NUMBER),!.%busca una palabra c
lave tal que
%pulperia, gasolinera
procesar_consulta(Lista,[X|Y],NUMBER):-ubicaciones(Lista,X),NUM1 is NUMBER+1, consultarintermedio([X|Y],NUM1),!.
procesar_consulta(Lista,[],NUMBER):- write('no entendi a que se referia con'),nl,write(Lista),nl,
    write('favor redigite su consulta de otra manera'),nl,
    read_line_to_string(user_input, X), tokenize_atom(X, Lista2),
    procesar_consulta(Lista2,Pclave,NUMBER), %esto deberia de estar bien.
    write('su consulta es:'),write(Pclave).
% SIENTO QUE EL ERROR ESTA EN QUE UN ARGUMENTO ES VACIO , CUANDO NO
% DEBERIA DE SERLO, PROBAR LUEGO,patch up con el corte , pero es
% necesario conocer la razon del error.
% ------------------------------------------------------------------------
% codigo para seguir consultado EN CASO DE QUE EL USUARIO DIGA QUE ESTA
% EN UN SUPERMERCADO, CONSIDERE 3 DESTINOS INTERMEDIOS MAX, ESTO IMPLICA
% QUE SEA UN VALOR QUE SE PASA. -ESTE CODIGO DEBE DE TENER UN TIPO DE
% ITERACION EN CASO DE QUE NO ENTIENDA QUE LE METIO EL USUARIO.
% CONSIDERE QUE EL USUARIO PUEDE DECIR ESTOY EN LA PULPERIA LA ESTRELLA,
% ENTONCES NO TENDRIA QUE PREGUNTAR EL NOMBRE DE LA PULPERIA, esto es un
% caso extra, o me puedo quitar el tiro y no preguntar el nombre de
% dicha pulperai
```

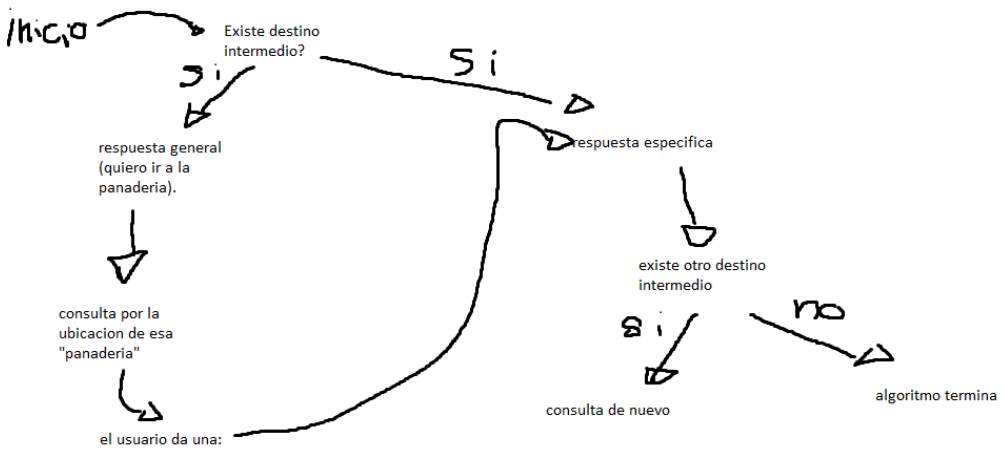
La regla anterior “procesar_consulta” , es una regla que permite la búsqueda de destinos intermedios, dado un input distinto de no, posee de parámetros la lista de input del usuario,[X|Y] es la lista de lugares intermedios, Number es el contador que determina cuándo dejar de consultar por más destinos intermedios esto sucede cuando prolog deduce que NUM=3, , cuando prolog plantea la consulta “existen otros destinos intermedios” , la primera línea en esencia llama a la regla oraciones , oraciones busca palabras clave relacionadas con destinos generales, luego llama a consultar_lugar, una regla que ejecuta el siguiente algoritmo:



Si para ambas consultas retorna true, entonces el algoritmo sigue esa línea de consulta.

La segunda línea de procesar consulta es similar a la primera, pero en vez de buscar una palabra clave en general(panaderia,gasolinera,etc) ,busca un destino específico del grafo, luego le consulta con la regla consultar intermedio , si existe otro destino intermedio, entonces el usuario debe digitar dicho destino intermedio.

Dada la explicación anterior, usted probablemente se plantee, ¿Por qué necesita 2 reglas para buscar palabras clave? ¿no pudo haberla hecho en una línea únicamente? la respuesta a ese cuestionamiento es muy posiblemente sí, posiblemente el diseño no es el más eficiente al momento, sin embargo durante la fase de desarrollo se decidió separarlo por el flujo de la estructura de respuesta del programa (el si o no a si existen destinos intermedios) , esto como una manera de verificar errores de manera más simple en la resolución de las respuestas, al haber 2 reglas para buscar palabras clave, el flujo de respuesta que se menciona es el siguiente:



nuevamente recordar que la línea en las reglas:

```
read_line_to_string(user_input, X), tokenize_atom(X, Lista2),
```

Lo que hace es leer el input del usuario y convertirlo a una lista.

Para la regla de búsqueda de destinos generales:

```
destino_general([X|_],X):-lugar(X),!. %elimine la linea de arriba
%EL PROGRAMA DEBERIA DE TERMINAR ACA.
destino_general([_|Entrada],Salida):-destino_general(Entrada,Salida).
% Predicado para dividir una cadena en palabras y convertir
% Predicado para leer una línea de entrada y convertirla en una lista de palabras.
% ##########################################################################
```

la primera linea es la condición de parada, si el primer elemento de la lista está incluido en los hechos denominados “lugar” , retorna true, si la lista se hace vacía retorna false
La segunda línea es el separador de la lista, va seccionando la lista en partes y llama recursivamente a destino general para verificar si el primer elemento está en los hechos denominados lugar.

```
% %CONSULTA LUGARES ESPECIFICOS EN EL GRAFO
ubicaciones([X|_],X):-ubicacion(X),!.
ubicaciones([_|Entrada],Salida):-ubicaciones(Entrada,Salida).
%##########################################################################
```

La regla anterior tiene el mismo propósito que destino general, sin embargo esta es utilizada para buscar ubicaciones específicas del grafo y no lugares generales.

```
% ################################
consultarlugar(Pclave,[],NUM):-maximo(NUM), write('perfecto, se ha calculado la ruta'),nl,write(Pclave),!.
consultarlugar(Pclave,Lista,NUM):-write('indiqueme donde se encuentra tal'),write(Pclave),nl,
    read_line_to_string(user_input, Z), tokenize_atom(Z, Lista2),procesar_consulta(Lista2,Lista,NUM).
% ------------------------------------------------------------------------
```

La regla anterior funciona como una regla auxiliar a la regla de procesar consulta, esta regla recibe una palabra clave ,una lista, y un numero , de manera recursiva “lista” va a ser la siguiente palabra clave del siguiente input, en caso de que el usuario indique de que necesita pasar por otro destino intermedio, num es la variable que lleva cuenta de cuantos destinos intermedios ha procesado hasta ese momento. Es importante considerar que la regla anterior es un flujo, que funciona para solicitarle al usuario de la siguiente manera:
Si el usuario inserta quiero ir a la panaderia, prolog pregunta digame donde se encuentra tal panaderia, por tanto esta regla se ejecuta, la regla opuesta ,donde el usuario indica de una vez que quiere ir a cartago por ejemplo, es la siguiente:

```
% DIRECTAMENTE EL USUARIO LE INGRESA UN LUGAR DEL GRAFO
consultarintermedio([Pclave|[]],NUM):-maximo(NUM), write('perfecto, SE va a concluir con su ultima consulta'),nl,write(Pclave),!.
consultarintermedio([X|Y],NUM):-write('Excelente, existe otro destino aparte de'),write(X), write(' que desea ir? Favor digitelo'),nl
,read_line_to_string(user_input, Z), tokenize_atom(Z, Lista),tomar_decision(Lista,Y,NUM).
% ##########################################################################
% ##############3
```

Mismo funcionamiento que la regla anterior, solo que este si añade a la lista de intermedios, en vez de solicitarle al usuario que digite la ubicacion del destino general(que diga panaderia).

BNF

El BNF tiene los hechos que son las palabras que pueden formar los sintagmas y oraciones, en el caso de los sintagmas y oraciones son reglas que tienen que cumplir con ciertas condiciones para que sean oraciones válidas para el BNF
Un ejemplo de los hechos que se utilizan son:

```
nodo([cartago|S],S).
saludo([hola|S],S).
despedida([adios|S],S).
verbo([ir|S],S).
verbo([encuentro|S],S).
verbo([tengo|S],S).
verbo([dirijo|S],S).
preposicion([en|S],S).
preposicion([a|S],S).
preposicion([hasta|S],S).
pronombre([me|S],S).
pronombre([mi|S],S).
adjetivo([ubicado|S],S).
adjetivo([posicionado|S],S).
adjetivo([establecido|S],S).
sustantivo([destino|S],S).
sustantivo([ubicacion|S],S).
sustantivo([ganas|S],S).
conjucion([que|S],S).
lugar([supermercado|S],S).
afirmacion([si|S],S).
negacion([no|S],S).
```

Como se pueden ver son diferentes hechos, desde nodo que son las ciudades hasta nombre que en este caso sería el nombre de la aplicación, también existen conectores que se utilizan para unir 2 palabras del última parte del sintagma verbal a la última parte del sintagma nominal como se va a ver en las reglas de los sintagmas y oraciones después.
Otro punto importante es que como se puede observar entre los paréntesis tiene que enviarme una lista y el resto de la lista , tomando en cuenta que antes del | es la cabeza de la lista lo que se hará la comparación de si la cabeza de la lista es igual a

alguno de los hechos es válido, y la S es el resto de la lista que se usa para devolver el resto de la lista menos la cabeza para seguir comparando el resto de la lista y verificar que la oración introducida por el cliente sea correcta y validada por el BNF

Las reglas para el BNF son para tener oraciones que son válidas, estas oraciones tienen que seguir cierta composición y orden para que sean válidas para el BNF, además también los sintagmas son otras reglas que también son parte de la oración y como se dijo anteriormente para que sea sintagma verbal o sintagma nominal tienen que estar conformados por ciertas partes y con cierto orden, en este caso los sintagmas son formados por hechos, mientras que oraciones pueden ser conformados por sintagmas que son reglas y por hechos.Las reglas para las oraciones son de este estilo, estos son algunos ejemplos (son todas a no ser que se haya actualizado el código de último momento) :

```
oracion(A,B):-saludo(A,B).
%Para que la frase recibida solo sea un saludo de los posibles
oracion(A,B):-afirmacion(A,B).
%Para que la frase recibida pueda ser solo una de las posibles afirmaciones
oracion(A,B):-negacion(A,B).
%Para que la frase recibida pueda ser solo una de las posibles negaciones
oracion(A,B):-despedida(A,B).
%Para que la frase recibida pueda ser solo una de las posibles despedidas
oracion(A,B):-despedida(A,C),despedida(C,B).
%Para que la frase recibida sea una despedida de las posibles y uno de los nombres disponibles
oracion(A,B):-nodo(A,B).
%Para que la frase pueda ser solo sea un nodo/ciudad que pertenece al grafo
oracion(A,B):-sintagmanominal(A,B).
% para que la frase pueda ser un conector y el nodo como ejemplo [a,cartago]
oracion(A,B):-lugar(A,B).
% para que la frase pueda ser un conector y el nodo como ejemplo [a,cartago]
oracion(A,B):-sintagmaverbal(A,C),sintagmanominal(C,B).
%Parteprincipal del BNF para frases relativamente complejas y que se usaran generalmente, conformadas por un sintagma verbal y un sintagma nominal
oracion(A,B):-saludo(A,C),sintagmaverbal(C,Z),sintagmanominal(Z,B).
%Esta parte es para poder recibir una oración más compleja, que contenga un saludo, un sintagma verbal y un sintagma nominal
oracion(A,B):-saludo(A,C),nombre(C,Z),sintagmaverbal(Z,L),sintagmanominal(L,B).
%Esta parte es para poder recibir una oración más compleja, que contenga un saludo, un nombre, un sintagma verbal y un sintagma nominal
```

y las reglas de los sintagmas son formados de las siguiente manera:

```
%Definicion de los sintagmas
```

```
%Definicion de los sintagmas nominales
sintagmanominal(S0,S):-preposicion(S0,S1),nodo(S1,S).
sintagmanominal(S0,S):-nodo(S0,S).
sintagmanominal(S0,S):-artiindefinido(S0,S1),lugar(S1,S).
sintagmanominal(S0,S):-lugar(S0,S).
sintagmanominal(S0,S):-preposicion(S0,S1),sintagmanominal(S1,S).
sintagmanominal(S0,S):-verbo(S0,S1),nodo(S1,S).
sintagmanominal(S0,S):-verbo(S0,S1),sintagmanominal(S1,S).

%Definicion de los sintagmas verbales
sintagmaverbal(S0,S):-verbo(S0,S1),verbo(S1,S).
sintagmaverbal(S0,S):-verbo(S0,S1),sintagmanominal(S1,S).
sintagmaverbal(S0,S):-verbo(S0,S).
sintagmaverbal(S0,S):-verbo(S0,S1),sintagmaverbal(S1,S).
sintagmaverbal(S0,S):-pronombre(S0,S1),verbo(S1,S).
sintagmaverbal(S0,S):-verbo(S0,S1),conjucion(S1,S).
sintagmaverbal(S0,S):-verbo(S0,S1),adjetivo(S1,S).
sintagmaverbal(S0,S):-sustantivo(S0,S1),verbo(S1,S).
sintagmaverbal(S0,S):-verbo(S0,S1),sustantivo(S1,S).
sintagmaverbal(S0,S):-pronombre(S0,S1),sintagmaverbal(S1,S).
sintagmaverbal(S0,S):-artiindefinido(S0,S1),verbo(S1,S).
```

Tanto las oraciones como los sintagmas pueden ser modificados para cambiar las cosas que los combinan, o agregar nuevos tipos de oraciones o diferentes sintagmas que sean formados por más cosas, menos cosas, o diferentes cosas, el bnf puede ser tan grande como la persona lo quiera mientras que vaya agregando todo correctamente, por eso se recomienda tener un buen manejo de código para entender qué parte pertenece a qué y qué cosa se pueden modificar sin problemas además de poder tener a mano si es necesario cambiar alguna palabra de las registradas o agregar nuevas

Para ver todas las palabras, sintagmas,oraciones posibles, además de los lugares, verbos,conjunciones,artículos indefinidos, etc para formar palabras que tiene el programa registrado se puede revisar el Manual de Usuario de Wazelog donde en la parte de limitaciones esta todo esto

Reloj

El reloj de wazelog está compuesto hecho de simplemente reglas, existiendo 3 grandes partes por decirlo así, primeramente la regla convertirHora(Hora):-.
que esta lo que hará es como una función, utilizando un par de líneas para obtener la hora UTC de la computadora, esto para obtener la hora en hora militar que sería de 0 a 23, esto para sí en caso de ser mayor a 12 o menor a 12 las otras reglas teniendo esto en cuenta hacen una cosa u otra cosa, como se puede ver en la siguiente imagen es una regla bastante sencilla de implementar, además contiene para mostrar el consola la hora actual en forma militar

```
% Predicado para convertir la hora en formato militar
convertirHora(Hora) :-
    get_time(Tiempo), %Para obtener la hora de la computadora
    stamp_date_time(Tiempo, FechaHora, 'UTC'), %Esto es para obetener la FechaH
    date_time_value(hour, FechaHora, Hora24), %Esto nos da la hora en formato d
    date_time_value(minute, FechaHora, Minutos),  %Esto nos da la los minutos d
    Hora is Hora24 ,writeln('La hora actual es: '),writeln(Hora:Minutos).%Para
```

Las otras reglas son simplemente teniendo en cuenta la hora en forma militar para multiplicar el Dato, que en este caso serían los kilómetros dados por el dijkstra, como se puede ver a continuación

```
% Predicado para multiplicar un dato por otro dato dependiendo de si
multiplicarDato(Dato, Resultado) :-
    convertirHora(Hora),
    Hora < 12,
    Resultado is Dato * 2.
multiplicarDato(Dato, Resultado) :-
    convertirHora(Hora),
    Hora >= 12,
    Resultado is Dato * 4.

%Predicado solo para multiplicar el dato para el tiempo estimado
multiplicarDato2(Dato, Resultado) :-
    Resultado is Dato * 2.
```

por último tenemos las reglas que son las que utilizan todas las anteriores para obtener un resultado, tenemos la regla tiempoenpresa para obtener el tiempo estimado en presa, y la regla tiempoestimado que se utiliza para obtener el tiempo estimado de la ruta, el tiempo estimado en presa necesita el tiempo estimado de la ruta, por lo cual se necesita usar primero la regla del tiempo estimado, además esta

regla de tiempo estimado necesita los kilómetros totales de la ruta para dar un tiempo estimado.

```
%Esta regla es para encontrar el tiempo estimado pero en presa, tomando en cuenta la hora  de costa rica
%Si la hora es mayor que 12 el tiempo estimado se multiplica por 6
%si la hora es menor que 12 el tiempo estimado se multiplica por 2
%El dato lo regresa como Tiempoenpresa
tiempopresa(Tiempoestimado,Tiempoenpresa):-multiplicarDato(Tiempoestimado,Resultado),Tiempoenpresa is Resultado.

%Esta regla es para encontrar el tiempo estimado de la ruta segun los kilometros, actualmente se multiplica simplente por
tiempoestimado(Kilometros,Tiempoestimado):-multiplicarDato2(Kilometros,Resultado),Tiempoestimado is Resultado.
```

Descripción de las estructuras de datos desarrolladas

Para el caso la lógica del programa y el algoritmo de dijkstra se implementó el uso de la lista como medio de guardado, en el caso de la lógica del programa , se ha de guardar en una lista los destinos intermedios(en caso de que existan) , que sean ingresados por el usuario, por otra parte existirán listas por aparte que representan el nodo origen y después el nodo destino, al final se hace la unificación de las listas en otra lista, donde su estructura es la siguiente:
nodo origen,nodos intermedios, nodo destino. Dicha lista es enviada al algoritmo de dijkstra , que a su vez retorna una lista aún más grande con cada ruta más corta entre cada uno de sus nodos, es decir , origen-nodo1,nodo1-nodo2,nodo2-nodo3,nodo3-destino.

Descripción detallada de los algoritmos desarrollados

BNF

El código consta de varias secciones:

Definición de hechos: se definen hechos que representan nodos en un grafo, saludos posibles, despedidas posibles, verbos, conectores, lugares, afirmaciones, negaciones y nombres. Estos hechos se utilizan posteriormente en la definición de la gramática.

Definición de la gramática: se definen reglas de producción que especifican las posibles oraciones que se pueden recibir y que son correctas según este BNF. Las reglas de producción incluyen oraciones de una sola palabra o sencillas, oraciones más complejas y combinaciones de saludos, nombres, sintagmas verbales y sintagmas nominales.

Definición de los sintagmas nominales y verbales: se definen reglas de producción para los sintagmas nominales y verbales, que se utilizan en la definición de la gramática.

En resumen, este código implementa un algoritmo de análisis sintáctico en Prolog para comprender oraciones en lenguaje natural ingresadas por un usuario en un chatbot. El algoritmo utiliza la notación BNF para definir las posibles oraciones y los componentes gramaticales que las componen.

Parser y lógica de respuesta

A continuación se describen los algoritmos desarrollados, la explicación detallada de cada regla se realizó en descripción de hechos y reglas

```
leer([]):-write('bienvenido a wazelog, su
navegador fiable.Por favor digame donde se encuentra:'),nl,
 read_line_to_string(user_input, String), tokenize_atom(String, Lista_result),%con
sulta el origen
 consulta_inicial(Lista_result,Result1),
 write('Excelente, estamos en: '),write( Result1),nl,write('favor digite su destin
o'),nl,
 read_line_to_string(user_input, String2), tokenize_atom(String2, Lista_result2),%
consulta el destino.
 consulta_inicial(Lista_result2,Result2),nl,write('Perfecto, vamos hacia '),write(
Result2),nl,
 write('Existe algun destino intermedio?, favor dijite no en caso de que no haya,
en caso contrario favor dijite el destino intermedio'),nl,
 read_line_to_string(user_input, String3), tokenize_atom(String3, SIONO),
 tomar_decision(SIONO,Lista_resultado,0),
 concatenar([Result1],Lista_resultado,X),
 concatenar(X,[Result2],Y),
 write('su resultado'),nl,write(Y).
 %write(Lista_resultado).
```

Esta es la regla general del parser, de esta regla sale la lista que va a pasar al dijkstra, el algoritmo empieza solicitando al usuario un punto inicial y un punto final, luego le consulta al usuario si existen puntos intermedios a consultar, si el usuario le digital ESPECÍFICAMENTE “no” , el algoritmo termina, si el usuario digita otra cosa , empieza el algoritmo de armado de la lista de destinos intermedios inicia:

Algoritmo de búsqueda de destinos intermedios:

```
%CONSULTAS DE DESTINOS INTERMEDIOS##################################:
procesar_consulta(Lista,Lista2,NUMBER):-oraciones(Lista,Pclave),consultarlugar(Pclave,Lista2,NUMBER),!.%busca
lave tal que
%pulperia, gasolinera
procesar_consulta(Lista,[X|Y],NUMBER):-ubicaciones(Lista,X),NUM1 is NUMBER+1, consultarintermedio([X|Y],NUM1),
procesar_consulta(Lista,[],NUMBER):- write('no entendi a que se referia con'),nl,write(Lista),nl,
    write('favor redigite su consulta de otra manera'),nl,
    read_line_to_string(user_input, X), tokenize_atom(X, Lista2),
    procesar_consulta(Lista2,Pclave,NUMBER), %esto deberia de estar bien.
    write('su consulta es:'),write(Pclave).
% SIENTO QUE EL ERROR ESTA EN QUE UN ARGUMENTO ES VACIO , CUANDO NO
% DEBERIA DE SERLO, PROBAR LUEGO,patch up con el corte , pero es
% necesario conocer la razon del error.
% ------------------------------------------------------------------------
% codigo para seguir consultado EN CASO DE QUE EL USUARIO DIGA QUE ESTA
% EN UN SUPERMERCADO, CONSIDERE 3 DESTINOS INTERMEDIOS MAX, ESTO IMPLICA
% QUE SEA UN VALOR QUE SE PASA. -ESTE CODIGO DEBE DE TENER UN TIPO DE
% ITERACION EN CASO DE QUE NO ENTIENDA QUE LE METIO EL USUARIO.
% CONSIDERE QUE EL USUARIO PUEDE DECIR ESTOY EN LA PULPERIA LA ESTRELLA,
% ENTONCES NO TENDRIA QUE PREGUNTAR EL NOMBRE DE LA PULPERIA, esto es un
% caso extra, o me puedo quitar el tiro y no preguntar el nombre de
% dicha pulperai
```

En el algoritmo anterior se trata de buscar palabras claves de 2 maneras diferentes, por un lado se buscan destinos concretos , con los nombres específicos de destinos del grafo, por otra parte , se buscan palabras claves de destinos generales, por ejemplo panadería, gasolinera, restaurante, etc a lo que prolog devolverá una pregunta de donde se ubica, este algoritmo además considera la variable NUMBER que se va acumulando ,cuando number sea 3, deja de buscar destinos intermedios y retorna la lista, a continuación se explicará el algoritmo consultar lugar y consultar intermedio:

```
consultarlugar(Pclave,[],NUM):-maximo(NUM), write('perfecto, se ha calculado la ruta'),nl,write(Pclave),!.
consultarlugar(Pclave,Lista,NUM):-write('indiqueme donde se encuentra tal'),write(Pclave),nl,
    read_line_to_string(user_input, Z), tokenize_atom(Z, Lista2),procesar_consulta(Lista2,Lista,NUM).
% ----------------------------------------------------------------
```

consultar lugar en esencia pregunta , si la variable num es 3, si devuelve true el algoritmo de crear la lista de intermedios termina , si no vuelve a pedir un input y vuelve a procesar la consulta, hasta que num sea 3.

```
consultarintermedio([Pclave|[]],NUM):-maximo(NUM), write('perfecto, SE va a concluir con su ultima consulta'),nl,write(Pclave),!.
consultarintermedio([X|Y],NUM):-write('papito Existe otro destino aparte de'),write(X), write(' que desea ir? Favor digitelo'),nl,
   read_line_to_string(user_input, Z), tokenize_atom(Z, Lista),tomar_decision(Lista,Y,NUM).
```

consultar intermedio en esencia hace lo mismo, sin embargo evade la pregunta de “indiqueme donde se encuentra tal (lugar general) ,, pregunta por otro destino intermedio en caso de que NUMBER no sea 3.

```
tomar_decision(SIONO,[],Numero):-decision(SIONO),write(Numero).%lista en caso de que sea no
tomar_decision(Lista_Destino,Lista_resultado,Numero):-not(decision(Lista_Destino)) ,procesar_consulta(Lista_Destino,Lista_resultado,Numero).%en caso de que sea SI.

%funcion que une un elemento con una lista
```

El algoritmo anterior en esencia lo que hace es procesar la consulta del usuario, si el usuario digita no, el algoritmo consultar intermedio o consultar lugar se detiene, si el usuario digita otra cosa, el algoritmo procede a realizar la consulta de lo que el usuario le haya escrito.

```
%funcion que une un elemento con una lista
concatenar([],L,L).
concatenar([X|L1],L2,[X|L3]):-concatenar(L1,L2,L3).
```

el algoritmo concatenar lo que hace es dar 2 listas, las une.
Y por último , el algoritmo:

```
oraciones([X|_],X):-lugar(X),!. %elimine la linea de arriba
%EL PROGRAMA DEBERIA DE TERMINAR ACA.
oraciones([_|Entrada],Salida):-oraciones(Entrada,Salida).
% Predicado para dividir una cadena en palabras y convertir
% Predicado para leer una línea de entrada y convertirla en una lista de palabras.
```

realiza las consultas para buscar una palabra clave, en caso de que sea un lugar general como panaderia, gasolinera, entre otros.

Implementación del dijkstra

```
% Entrada: Recibe Camino que es la ruta junto con los costos, Ruta es
% una lista de solo la ruta, Distancias es una lista con las
% distintas distancias
% Funcion: Este hecho verifica cuando se debe detener la llamada
% recursiva y muestra en consola la informacion deseada
% Salida: Muestra el resultado en consola de la ruta mas corta y la
% distancia respectiva.
dijkstra([_],Camino,Ruta,Distancias):-nl,reverse(Camino,Camino_Ordenado),
    write("Esta es la ruta ":Camino_Ordenado),
    nl,write("Esta es la ruta junta ":Ruta),
    nl, sumar_lista(Distancias,Km),
    write("Este es el costo total ":Km).


% Entrada: Recibe la lista con el origen, destinos intermedios y destino
% Funcion: Esta es la llamada del dijkstra que junto con findall
% encuentra todas las rutas posibles, y usa a min_lista para escoger la
% mas corta e ir formando una lista con el camino final y una lista con
% los distintos costos.
% Salida: Muestra el resultado en consola de la ruta mas corta con su
% distancia respectiva.
dijkstra([Origen,Destino|Resto], Resultado, Camino, Km):-
    findall([Ruta,Costo], posibles_rutas(Origen, Destino, Ruta, Costo),Rutas),
    min_lista(Rutas, RutaMin),
    primeros_dos_elementos(RutaMin, PrimerElemento, SegundoElemento),
    append(Camino,PrimerElemento,Camino2),
    dijkstra([Destino|Resto],[RutaMin|Resultado],Camino2, [SegundoElemento|Km]),
    !.

% Entrada: Recibe la lista con el origen, destinos intermedios y destino
% Funcion: Es un hecho intermedio para llamar a dijkstra con los
% parametros necesarios
inicia_dijkstra(Lista):-dijkstra(Lista,[],[],[]).
```

De manera inicial para ejecutar el conjunto de reglas y hechos que llevan a cabo toda la búsqueda de la ruta más corta y mostrarla se hace el llamado a inicia_dijkstra este hecho tiene la regla principal en la que se hace el llamado al algoritmo de dijkstra.
Dentro del algoritmo de dijkstra se llevan a cabo distintas ejecuciones de reglas en las que se encuentran las rutas de los distintos nodos que se le pasan al dijkstra. En la imagen que se muestra a continuación se puede observar un conjunto de hechos y reglas dentro de posibles rutas.

```
% Esta regla es la que ayuda a encontrar el camino mas corto
posibles_rutas(Origen, Destino, Ruta, Longitud) :-
    posibles_rutas(Origen, Destino, [Origen], Ruta_acomodada, Longitud), reverse(Ruta,Ruta_acomodada). %Se usa reverse
% porque sino la ruta va a quedar ordenada al reves, o sea de destino
% a origen

% En este hecho se verifica cuando se llega al caso de parada.
posibles_rutas(Destino, Destino, Ruta, Ruta, 0).

% Este es el caso recursivo para encontrar el camino más corto
posibles_rutas(Origen, Destino, Visitados, Ruta, Longitud) :-
    grafo(Origen, X, D),                % Encontrar un vecino X
    \+ member(X, Visitados),            % Asegurarse de que X no esté en la lista visitada
    posibles_rutas(X, Destino, [X|Visitados], Ruta, Rest_longitud),   % Recursión
    Longitud is D + Rest_longitud.

%Este hecho sirver para retornar los dos primeros elementos de una lista
primeros_dos_elementos([X, Y|_], X, Y).
```

Esto básicamente sirve para encontrar las rutas posibles, es como un auxiliar del dijkstra, lleva a cabo las distintas verificaciones para evaluar cuándo debe detenerse así como el revertir el orden de las listas desarrolladas para que queden en un orden correcto.

Problemas sin solución

Un aspecto que no se pudo solucionar es al final del programa cuando se muestra la ruta final, hay repeticiones de lugar debido a que son los enlaces que existen entre cada ruta antes de ser unida, sin embargo, la ejecución del programa no se ve afectada por este aspecto, es más que todo un aspecto visual.

Actividades a realizar por estudiante

Creación provisional del BNF
Se tiene que crear un BNF sencillo el cual es provisional, para poder seguir con la otra parte del proyecto, este se puede agrandar más para tener más posibilidades de oraciones
Encargado: Ignacio Lorenzo Martínez
Tiempo estimado: 4-6 horas repartidas en varios días debido al horario del encargado

Flujo de conversación provisional
Esta tarea es un pequeño chatbot donde simplemente se valida si las oraciones son válidas para continuar la conversación, usada para comprobar las diferentes reglas y hechos definidos en BNF además de probar el modularidad que se puede utilizar en Prolog que en este caso sería utilizando
consult('nombre de archivo.pl').
 Encargado: Ignacio Lorenzo Martínez
Tiempo estimado: 30 mins a 1 hora
Importante recordar que esto es solo para probar las diferentes posibilidades de recibir la oración y comprobarla con el BNF

Mejoramiento del BNF
Esta tarea es para agrandar el BNF y mejorarlo, para que tenga más hechos y diferentes tipos de oraciones las cuales toma como correctas, esta tarea es mejoramiento para tener más opciones de las oraciones recibidas por el cliente

Encargado: Ignacio Lorenzo Martínez, pero los otros integrantes pueden ayudar o aportar ideas para esta parte, y si es necesario añadir partes que sean necesarias para otra parte de la lógica
Tiempo estimado: No hay un tiempo estimado ya que se puede dedicar muchísimo tiempo o poco tiempo según avance el resto del proyecto

Creación inicial del Parser y lógica de respuestas:
Se debe de crear el parser , quien debe de buscar palabras clave luego de verificarse que el input del usuario está bien formulado ,se plantean 2 rutas, una donde el usuario indica un lugar general(supermercado,gasolinera,escuela) , etc y

los lugares específicos(que son parte del grafo definido por el profesor), el parser trabaja en conjunto con la lógica de respuestas
Encargado: Luis Alfredo González Sánchez
Tiempo estimado: 4-5 horas repartidas en 3 días.

Creación de lógica de respuestas:
La lógica de respuestas en si es lo que entiende del parser, por ejemplo si el usuario indica un lugar general prolog debe de preguntar la ubicación , si el usuario da una ubicación se añade a una lista , la lógica construye dicha lista que es enviada luego al algoritmo de dijkstra, se deben de definir bastantes ubicaciones generales como hechos, para su amplio funcionamiento
Encargado: Luis Alfredo González Sanchez
Tiempo estimado de 4-5 horas repartidas en múltiples días, con el fin de perfeccionar el algoritmo.

Investigación de un algoritmo de búsqueda.
Esta tarea tiene el fin de buscar un algoritmo que ayude a encontrar la ruta mas corta entre los nodos de origen y destino así como la conexión entre los puntos intermedios que se desean usar.
Encargado:Andres Molina
Tiempo estimado: El tiempo utilizado fue de aproximadamente 2 días.

Mejoramiento del algoritmo de dijkstra
Luego de la última reunión con los compañeros se aclararon dudas y se dejaron aspectos por mejorar como el manejo de la información que el dijkstra debe manejar de manera inicial y también en cómo la debe retornar.
Encargado:Andrés Molina, el compañero Luis Alfredo colaboró ajustando los últimos detalles.
Tiempo estimado: esta tarea se llevó a cabo en aproximadamente de 2 a 3 días.

Problemas solucionados

Hay un problema al definir hechos para el BNF en Prolog, los cuales son que nombres de ciudades que son separados como por ejemplo: San josé o Tres ríos , no se pueden colocar en hechos como `nodo([cartago|S],S).` ya que al colocar

```
17 +   nodo([san jose|S],S).
```

daría el siguiente error:

```
1 ?- consult('BNF.pl').
ERROR: c:/users/nacho/onedrive - estudiantes itcr/documentos/prolog/bnf.pl:17:10: Syntax err
or: Operator expected
true.
```

y aunque devolviera true dando a entender que se compiló correctamente, al buscar san jose no serviria y podría caerse el programa, ya que también la frase recibida por el cliente se parte como por ejemplo : [Necesito,ir, a San,Jose] , ya que cada espacio en blanco es tomado como una nueva palabra para agregarla a una lista de palabras la cual trabajara el BNF para comprobar si es una oración correcta

Otro problema que sale también de lo anterior es que para verbos y otras partes de la oración no se podía hacer, por lo cual se tuvo que hacer que el sintagma verbal también pudiera ser verbo y verbo para que frases como quiero ir o necesito ir sean válidas en este BNF

Conclusiones del proyecto

Recordar que deben estar basadas en los hechos, en hallazgos realizados al finalizar el proyecto

El BNF en Prolog se puede utilizar para resolver problemas de listas, entre otros. Es importante comprender los argumentos necesarios en cada caso y utilizar ejemplos para familiarizarse con su funcionamiento

BNF en Prolog es una herramienta poderosa para representar gramáticas de contexto libre y resolver problemas utilizando la lógica de predicados y las cláusulas de Horn. Un manejo claro y correcto del código, junto con una comprensión adecuada de las estructuras de control y la unificación, permitirá ampliar y mejorar el BNF a largo plazo.

Recomendaciones del proyecto

Un manejo adecuado del código no solo mejora la escalabilidad del BNF, sino también su mantenibilidad y legibilidad. Esto permite una colaboración eficiente entre los desarrolladores y facilita futuras actualizaciones y mejoras.

Además de separar los diferentes elementos del BNF, es esencial documentar el código de manera exhaustiva. Esta documentación debe incluir explicaciones sobre el propósito y el uso de cada elemento, así como ejemplos o pautas relevantes. Un BNF bien documentado beneficiará no solo al equipo de desarrollo actual, sino también a los futuros desarrolladores que puedan necesitar trabajar con el código.

Las revisiones regulares del código y su refactorización pueden ayudar a identificar y solucionar cualquier problema o ineficiencia en el BNF. Este proceso de mejora continua garantizará que el BNF siga siendo una herramienta valiosa y eficaz para el proyecto.

Por último, es importante tener en cuenta los objetivos y requisitos a largo plazo del proyecto al diseñar y gestionar el BNF. Al anticipar las necesidades futuras y los posibles cambios, podemos crear un BNF más robusto y adaptable que siga respaldando el crecimiento y el éxito del proyecto.

Tener definido Ejemplos de cómo recibiría cada una de las reglas de las oraciones para tener visualmente entendimiento de esta regla y poder modificarla o arreglar en caso de ser necesario
Martes 2 de octubre:
Se unieron todas las partes además cree un reloj para obtener la hora y poder hacer el tiempo estimado y el tiempo estimado en presa según la distancia total de la ruta que obtenemos de dijkstra, actualmente la hora que hay menos presa son antes de las 12, y después de las 12 hay presa

Bibliografía consultada en todo el proyecto

Para crear el reloj se utilizó la documentación de Prolog que sería el siguiente enlace
https://www.swi-prolog.org/pldoc/man?section=timedate

Esta es una librería que viene con Prolog desde la primera vez que se instala lo cual facilita su uso ya que no se tiene que descargar nada externo

Las presentaciones brindadas por el profesor del curso en el 2 semestre 2023.

Bitácora

Ignacio Lorenzo Martínez

23 de septiembre:
Primera reunión
Se me asignó el bnf y parser, ya tenemos la estructura donde concordamos los 3 en usar esa,
Andres se encargará de dijkstra y Luis de la base de datos (Hechos y grafo)
Comencé la investigación sobre BNF en prolog, y revisando un archivo que usó Luis anteriormente para investigar sobre esto, la estructura inicial del BNF es sintagma Verbal y Sintagma Nominal

Más noche se habló vía whatsapp con el profesor sobre dudas y para afirmar las frases utilizadas inicialmente las cuales fueron definidas por el grupo en la reunión anterior, teniendo una respuesta negativa del profesor con las frases seleccionadas, donde se refiere a estas como muy "tiesas" y que nosotros no hablamos así, durante la charla el profesor me explico el uso de varios hechos para declarar diferentes tipo de oraciones, al no tener claro el tema el profesor realizó una llamada la cual se podría como una reunión rápida para aclarar dudas respecto al tema, donde quedo claro como se tiene que utilizar el bnf, y que este simplemente tiene que verificar si la oración cumple con las reglas que hemos predefinido anteriormente, recomendó utilizar saludos, despedidas, nombre de la aplicación para hacer las frases menos "tiesas", también personalmente se ideó que pudiera recibir afirmaciones,negaciones debido a la naturaleza del proyecto además de otras cosas.

25 de septiembre:
Se comenzó una investigación sobre el BNF de diferentes proyectos y otros usos que se han usado, la mayoría de los casos son chatbots, esto para tener una idea de las frases que se tenían que utilizar, como definir los hechos para cada parte de los sintagmas, además de como funciona la comprobación de la oración con el BNF, teniendo en cuenta lo hablado el 23 con el profesor el cuál nos recomendó hacer un BNF menos "tieso" se tiene pensado agregar saludos, despedida y el nombre de la aplicación para que existan oraciones que contengan esto y sean correctas

27 de septiembre:
 Se concluyó un BNF actualmente básico y simple donde tiene varias definiciones de hechos para diferentes tipos de oraciones, desde solo recibir el lugar/ciudad a la que se quiere ir o donde está ubicado actualmente, hasta frases más complejas como hola wazelog quiero ir a cartago, esto teniendo en cuenta que quiero ir tiene que ser una comprobación de 2 verbos debido a que prolog no permite utilizar

palabras separadas, esto se tendrá en cuenta para preguntarle al profe debido a que ciertas ciudades tienen el nombre separado como seria san jose juan viñas
29 de septiembre:
Con la confirmación del profesor en la clase se utiliza las ciudades que tienen nombre separados como nombres pegados como seria sanjose,tres rios, etc
Además de esto se definió más hechos en verbos y otras partes de las posibles oraciones para tener más ejemplos y posibilidades de oraciones introducidas por el cliente, además de todo esto se realizó un flujo de conversación el cual solamente verifica si la oración está bien escrita siguiendo las reglas del BNF, actualmente se puede responder a la pregunta con cualquier oración que corresponda al BNF y la conversación continua

30 de septiembre:
se modificaron los hechos de los nodos con el grafo enviado por el profesor, además de realizar las pruebas con el flujo lo cual independientemente funciona correctamente hasta este punto

1 de octubre:
Se tuvo una reunión con los compañeros para unir las partes de cada uno, y para encontrar errores al hacer esto para poder arreglarlos antes de la entrega del proyecto, además de por parte personal se inició y se hizo gran parte de la documentación, y casi por completo el manual de usuario de wazelog, en la parte de código se dijo que ayudaría a comprobar con BNF para en caso de no ser correcto la oración usando el BNF el sistema pregunte nuevamente y en caso de estar bien con el continuar con la conversación entre el chatbot y el usuario

2 de octubre:
Se revisó el bnf para terminar el manual de usuario, además como Luis logró la parte principal se inició la unión por completo del BNF y el flujo de conversación, no se terminó ya que primero se hace una lectura de código para entenderlo y saber donde hay que colocar estas verificaciones del BNF y en caso de algún error poder arreglarlo

Andres Molina Redondo

23 de septiembre:
Se llevó a cabo una reunión de los 3 integrantes en donde se acordó cuál va a ser la estructura que se va a usar en la tarea. Posteriormente se comenzaron a asignar tareas importantes que cada uno debe ir desarrollando, se me asignó la tarea de encargarme del algoritmo de Dijkstra
Nacho se va a encargar del bnf y el parser
Mientras que luis se encargará de armar la base de datos, es decir, el grafo y los hechos

25 de septiembre:
Se comienza con la investigación del algoritmo de dijkstra que se implemente en prolog. De manera inicial primero se busca recordar completamente cual es el procedimiento que sigue el algoritmo para entender la implementación que se le va a dar al proyecto.
Después de probar con varios algoritmos se encontró uno que sirve con los casos de prueba utilizados, hay que esperar para probar con el grafo desarrollado por los compañeros para ver si funciona.

26 de septiembre:
Se realizó una reunión con el compañero luis para aclarar algunas dudas sobre las verificaciones del grafo dando como resultado que las dudas de implementación que tenía luis no generan problema debido a que el algoritmo de dijkstra que desarrollé verifica esos casos de error.
El algoritmo de dijkstra retorna 1 a 1 las posibles rutas encontradas, no solo la más corta, por lo tanto se llegó a la conclusión de que sería bueno modificar la implementación de manera que sólo muestre la menor ruta.

27 de septiembre:
Se logró obtener una lista en la que se muestran todas las rutas con su costo, ahora falta usar dicha lista para retornar únicamente la ruta con menor costo

29 de septiembre
Se logró implementar una regla de manera que se retorna la ruta más corta de todas las encontradas en el dijkstra.

1 de octubre
Se llevó a cabo una reunión con los compañeros para mostrar los avance y evacuar dudas, de mi parte se hicieron consultas sobre el formato en el que le van a llegar los datos al algoritmo de dijkstra, ya que, inicialmente se estaba desarrollando dicho algoritmo de manera externa a la parte de los compañeros, por lo tanto recibía 2 valores los cuales eran el nodo de origen y el nodo destino que se ingresaban

manualmente, luego de hablarlo con los compañeros se llegó a la conclusión de que al algoritmo de dijkstra le iba a llegar una lista con el origen,destinos intermedios y destino, y había que crear una regla intermedia que se encargará de mandar la información al dijkstra en pares ordenados.

2 de octubre
Se realizaron varios algoritmos para modificar la lista recibida por el dijkstra para manipular la información y poder brindarla de la forma deseada, se pudieron avanzar en algunos aspectos pero no se pudo terminar con la ejecución del algoritmo

4 de octubre
El compañero Luis Alfredo brindó ayuda para poder simplificar la manera en la que se le brinda la información al algoritmo de dijkstra, de esta manera se pudo concluir con el algoritmo debido a que se pudieron obtener las rutas necesarias junto con su distancia. Se ayudó a desarrollar la documentación respectiva de las partes solicitadas.

Luis Alfredo González Sánchez

23 de septiembre:
Acuerdos respecto a las reunión inicial : el proyecto consta de 6 partes importantes, el bnf , un parser, la lógica del bot, el dijkstra, la solución y la respuesta inválida(no entendió la frase), se acordó que Nacho realice el BNF (esto por que ya estuve batallando con el bnf sin resultado exitoso) además del parser, andrés se encarga de realizar el algoritmo de dijkstra, y yo me encargo de realizar la estructura del grafo, plantear los hechos (Ciudades) , y los hechos de nodos intermedios , por ejemplo supermercado (DELIMART) . esto para que sea utilizado en la lógica
al terminar el grafo de manera rápida, se procederá a desarrollar la lógica del programa y las respuestas, esto debe de ser un desarrollo en conjunto , plantear al menos el esqueleto de respuestas de la parte de la lógica por ejemplo
(leer frase -> buscar palabras clave -> encuentra? si , añade a la lista que es la que es insertada en dijkstra para consultar la ruta más corta, con o sin nodos intermedios , si no vuelve a preguntar para seguir añadiendo a esa lista, una vez que el usuario le indica no más nodos o intermedios o ya contiene 3 nodos intermedios, da como resultado la lista creada y se la pasa al dijkstra
IMPORTANTE DESTACAR, LA ESTRUCTURA DEL BNF A SEGUIR ES SINTAGMA VERBAL Y SINTAGMA NOMINAL, LA FRASE DEBE DE EMPEZAR CON EL SINTAGMA VERBAL Y EN ESPECÍFICO UN VERBO
estoy en , quiero , necesito pasar , etc

25 de septiembre:
se trabajó lo mencionado en la bitácora pasada, completitud parcial, posterior se encontró un problema a la hora de hacer las consultas, por lo que se decide trabajar en el parser, bajo la siguiente lógica , observe la imagen
Imagen
el problema que existe es que necesito una manera de que prolog lea un input tal que "estoy en superpan" y transforme ese input a lista , tal que [estoy,en,superpan] , para luego ser procesado y encontrar palabras clave
de momento no se ha encontrado solución a este problema

26 de septiembre:
se encontró una manera de traducir el input del usuario a una lista para que luego sea procesada , mediante el siguiente código :

```
leer([]):-write('bienvenido a wazelog, su
% navegador fiable.Por favor digame donde se encuentra:'),nl,
 read_line_to_string(user_input, X), tokenize_atom(X, L),
 Write(L).
```

en esencia read_line to_string es el read(x) que menciona el libro , pero para lista de caracteres.no para un solo elemento, mientras que tokenize atom agarra ese string y lo convierte en una lista

parte 2: luego de reunirse con andrés para discutir el dykstra y evaluar si el necesita un código (Que ya creé) en donde me diga si un nodo tiene relación o no, nos dimos cuenta que de momento no es necesario ya que el código de dijkstra que realizó andres ya lo considera, le probamos casos por ejemplo ciclos infinitos de relaciones a ver qué pasaba y de momento no generó algún ciclo infinito, por tanto el código que realice de grafo es descartado, sigo pendiente de realizar el grafo con los hechos , pero eso está en espera ya que el profe no lo ha mandado.
por el momento me voy a dedicar a realizar la lógica del robot, ya está hecho de que transforme input de usuario en una lista, y esta pseudo medio hecho el buscar palabras claves de lugares generales como panadería gasolinera entre otros
esos lugares generales son hechos y debe de ser ampliado para posibilidades de usuario

27 de septiembre:
de momento se contiene una estructura básica de procesamiento y búsqueda de palabras clave , dicha palabra clave puede ser un lugar general como "pulpería" u objetos del grafo como una ubicación concreta, tipo "cartago" si el programa no encuentra palabra clave en si , el programa le indica al usuario que no entendio la pregunta, y que por favor repita lo que quiere decir de manera diferente. Ahora , para el caso 1 donde la palabra clave puede ser pulpería , gasolinera, es esencial ampliar la gama de palabras claves, ya que se van a trabajar estas búsquedas de palabras clave como hechos (dado la necesidad de avanzar con el proyecto) , por tanto la base de datos de lugares generales deberá ser lo suficientemente amplia para cumplir estas necesidades. Otro dato adicional , dado este caso, es necesario modificar las reglas de búsqueda , dado que el usuario puede poner , estoy en la gasolinera, por lo que el sistema deberá de preguntar cual gasolinera? y continuar desde ahi, a como el usuario puede decir estoy en la gasolinera que se ubica en tres rios., entonces ahi ya tendria casos, por lo que considere como uno de los casos preguntar primero si es una palabra clave, y si lo es buscar en la misma frase a ver si el usuario menciona ubicacion , si no menciona ubicacion que la pregunte, y el ultimo caso es que el usuario de señales "ticas" , tipo estoy en el supermercado de color celeste cerca de la antigua amapola, o que diga que estoy en el supermercado "la estrella" , por lo que se salta la consulta del nombre y pregunta simplemente la ubicacion
en resumen : para consultar un destino intermedio existen 3 posibilidades,
1- que el usuario me indique únicamente que está en la pulpería, así en seco, de ahí pregunto el nombre y la ubicación.
2-que el usuario me diga la ubicación de la pulpería, que diga tengo que pasar a la pulpería que está por alajuela, entonces debe deducir pulpería y alajuela , esto es

deduce que después de pulpería hay mas texto, entonces manda nuevamente a buscar la palabra clave lugar
3- que me diga en tico : "tengo que pasar por la pulpería que está por la antigua mata de amapola

29-septiembre 23:
Después de lo discutido con Nacho para el procesamiento del dijkstra se decide que los destinos intermedios sean pocos, se mantiene la posición de que la base de datos de destinos intermedios sea incrementada, por otra parte se concluyen las labores de crear un parser que retorne palabras clave y contenga puntos intermedios , tal que el punto intermedio sea un lugar general , o directamente un lugar en concreto del grafo, el punto intermedio lleva a un punto en específico
Por otra parte se logra implementar un sistema básico de preguntas, tal que “bienvenido a wazelog, digame donde se encuentra” debe de ser un punto en específico , dígame su destino final, punto en específico, necesita pasar a un destino intermedio? , de los puntos intermedios se limitaron a un máximo de 3 .

2 Octubre 23:
Se logra implementar que el parser en conjunto con la lógica retorne una lista de manera: [punto inicial, intermedios, punto final) , se utiliza la función concatenar listas para concatenar el resultado de los intermedios , esta lista posteriormente va a ser enviada al dijkstra.
4 Octubre 23:
Se ayuda a Andres a corregir un error con el dikstra, simplificando el algoritmo y se termina de terminar la documentación.