TEC | Tecnológico de Costa Rica

**Área de ingeniería en computadores**

**Lenguajes, compiladores e intérpretes**

**II tarea programada:**

# Wazelog

**PROFESOR**

Marco Rivera Menénes

**ESTUDIANTES**


Sebastián Mora Godínez 2019227554 sebastianmg@estudiantec.cr

Emanuel Marín Gutiérrez 2019067500 emarin702740530@estudiantec.cr


**FECHA**

**9 de abril, 2021**

TEC | Tecnológico de Costa Rica

# Índice

## 1.1 Descripción de los algoritmos de solución desarrollados

La ejecución de la aplicación comienza llamando la cláusula ***wazelog***, en esta se manejará todo el flujo del programa. Primeramente se preguntará al usuario por el origen de la ruta, para esto, se llamará la cláusula ***preguntar_origen***, la cual se encargará de leer la respuesta del usuario, verificar por medio de la gramática libre de contexto establecida si la respuesta se dio de forma correcta y devolver el lugar de origen dado. En caso de que la oración no cumpla con la gramática, se vuelve a preguntar por el origen hasta que la respuesta sea correcta.

Después, la siguiente cláusula en ejecutarse será ***preguntar_intermedios****,* esta se encargará de devolver una lista con los lugares intermerdios dados por el usuario. El usuario debe decir el establecimiento a donde desea pasar durante su trayecto y después se le preguntará en dónde se ubica este establecimiento. Una vez dado el lugar se agrega a una lista y se vuelve a preguntar, en caso de que se tenga varios destinos intermedios. Cada vez que el usuario da una respuesta, se verifica que cumpla con la gramática establecida. En caso de no ser así, se imprime un mensaje de error y se vuelve a preguntar. Para salir continuar con el destino final, el usuario debe dar como respuesta ***no*** cuando se le pregunta si tiene algún destino intermedio.

Luego, se preguntará al usuario por el destino final de la ruta, para esto, se llamará la cláusula ***preguntar_destino***, la cual funciona de la misma manera que *preguntar_origen,* se encargará de leer la respuesta del usuario, verificar por medio de la gramática libre de contexto establecida si la respuesta se dio de forma correcta y devolver el lugar de destino dado. En caso de que la oración no cumpla con la gramática, se vuelve a preguntar por el destino hasta que la respuesta sea correcta.

Una vez que el usuario ha ingresado el punto de origen y el punto de destino, considerando o no puntos intermedios, se llama a la cláusula ***dame_ruta*****,** la cual, mediante el algoritmo de Dijkstra, devolverá una lista con la ruta total a seguir más corta en términos de distancia. Por último, se procederá a imprimirla en la interfaz para mostrarsela al usuario.

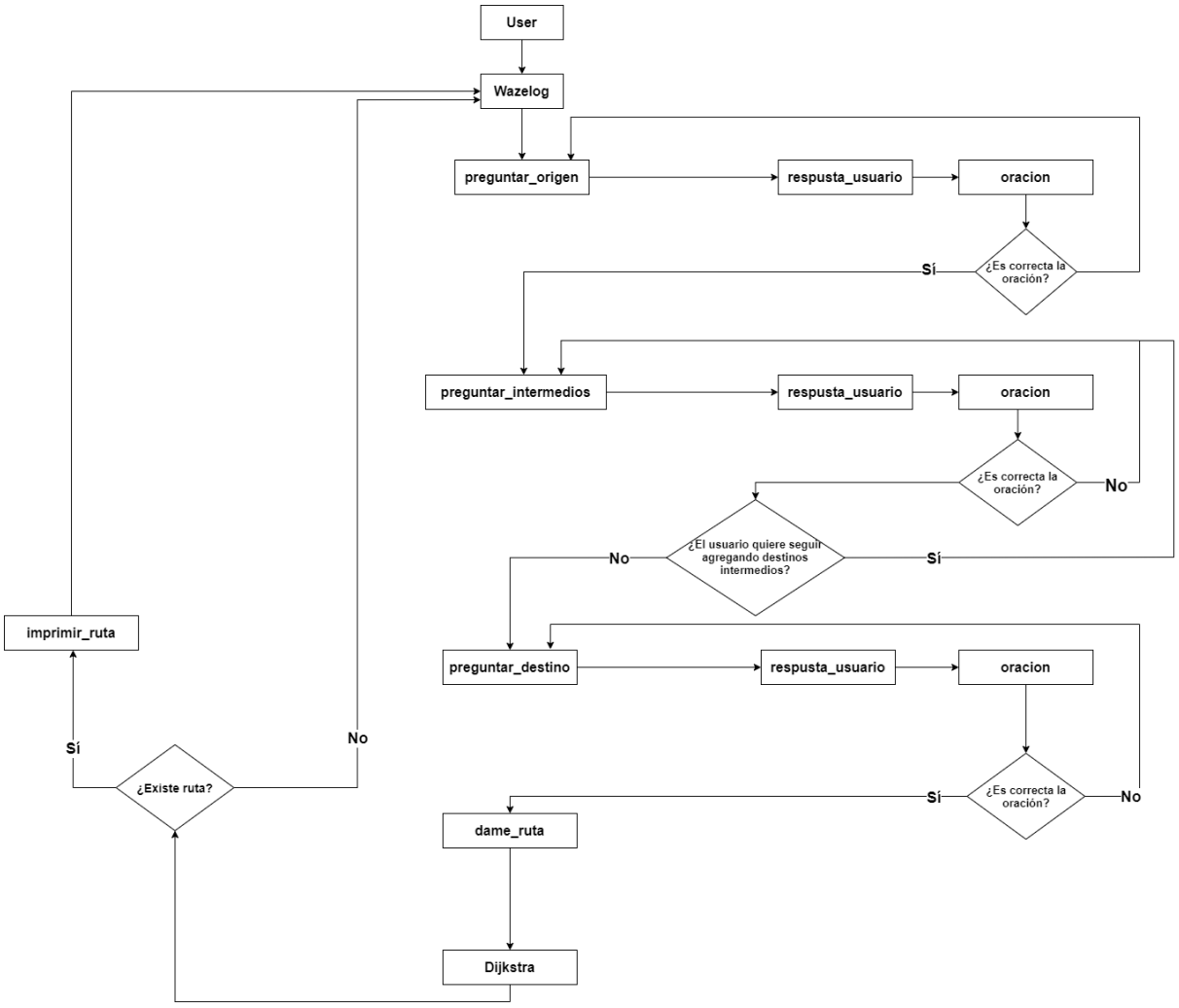


**Figura 1. Diagrama del algoritmo de la solución**

Para una mejor visualización del diagrama visite:
https://github.com/sebas-mora28/WazeLog/blob/master/docs/digrama.png

## 1.2. Descripción de los hechos y reglas implementadas

**Wazelog**

Es la regla principal del programa. Se encarga del flujo del programa. Llama a las demás cláusulas necesarias para el funcionamiento de la aplicación.

**Preguntar_origen(Origen)**

```
preguntar_origen(Origen):- write("Wazelog: Por favor indique donde se encuentra"), nl,
                   write("Usuario: "), readln(Respuesta),
                   respuesta_usuario(Respuesta, Origen),
                   existe_lugar(Origen),!.

preguntar_origen(Origen):- error, nl, preguntar_origen(Origen).
```

Esta regla se encarga de pedirle el lugar de origen al usuario. Una vez que el usuario da la respuesta, se procede a verificar que la oración cumple con la gramática establecida. Devuelve el nombre del lugar de origen.

En caso de que no se cumpla con la gramática establecida, se ejecutará la segunda cláusula, donde se imprimirá un mensaje de error y se llamará recursivamente.

**Preguntar_intermedios(Intermedios)**

```
preguntar_intermedio(Intermedios):- write("Wazelog: ¿tiene algun destino intermedio?"),nl,
                                    write("Usuario: "), readln(Respuesta), preguntar_intermedio(Respuesta, Intermedios),!.
```

Es la cláusula principal de esta parte. Se encarga de preguntar si tiene un destino intermedio y recibir la respuesta del usuario. Una vez con la respuesta del usuario, se llamada a la cláusula con su mismo nombre pero con dos parámetros. La variables Intermedios al final devolverá la lista con destinos intermedios dado por el usuario.

### Preguntar_intermedios(Respuesta, [Destino_intermedio | Intermedios])

```
preguntar_intermedio(Respuesta, [Destino_intermedio | Intermedios]):-   respuesta_usuario(Respuesta,Establecimiento), dif(Respuesta, no),
                                                                        existe_establecimiento(Establecimiento),
                                                                        write("Wazelog: ¿Donde se encuentra el/la "),
                                                                        write(Establecimiento), write("?"),nl,
                                                                        write("Usuario: "), readln(Respuesta2),
                                                                respuesta_usuario(Respuesta2,Destino_intermedio),
                                                                        existe_lugar(Destino_intermedio),
                                                                        write("Wazelog: Destino intermedio agregado: "),
                                                                write(Destino_intermedio),nl,
                                                                        preguntar_intermedio(Intermedios),!.
```

Esta es la función encargada de formar la lista con los destinos intermedios. Recibe la respuesta del usuario y verifica que esté correcta. Luego pregunta dónde se encuentra el establecimiento dado en la respuesta y agrega ese destino a la lista, se llama recursivamente para continuar con el ciclo hasta que el usuario no desee ingresar más destinos intermedios.

### Preguntar_intermedios(Respuesta, [])

```
preguntar_intermedio(Respuesta, []):- respuesta_usuario(Respuesta), negacion(Respuesta), !.
```

Es el punto de parada de, se encarga de verificar que la respuesta dada por el usuario corresponda a **no** para terminar esta parte.

### Preguntar_intermedios(Respuesta, Intermedios)

```
preguntar_intermedio(_, Intermedios):- error, nl,  preguntar_intermedio(Intermedios).
```

Se ejecuta en el momento en que la respuesta dada por el usuario sea incorrecta. Se imprime un mensaje de error  y se llama recursivamente.

### Eliminar_consecutivos

```
eliminar_consecutivo([], []).
eliminar_consecutivo([X], [X]).
eliminar_consecutivo([X1, X2 | Xs], [X1 | Ys]) :- dif(X1, X2), eliminar_consecutivo([X2|Xs], Ys).
eliminar_consecutivo([X, X | Xs], Ys) :- eliminar_consecutivo([X | Xs], Ys).
```

Eliminar elementos consecutivos de una lista.

**Preguntar_destino(Destino)**

```
preguntar_destino(Destino):- write("Wazelog: ¿Cuál es su destino?"), nl,
                    write("Usuario: "),readln(Respuesta),
                    respuesta_usuario(Respuesta, Destino),
                    existe_lugar(Destino),!.

tino(Destino):- error, nl, preguntar_destino(Destino).
```

Esta regla se encarga de pedirle el lugar de destino final al usuario. Una vez que el usuario da la respuesta, se procede a verificar que la oración cumple con la gramática establecida. Devuelve el nombre del lugar de destino.

En caso de que no se cumpla con la gramática establecida, se ejecutará la segunda cláusula, donde se imprimirá un mensaje de error y se llamará recursivamente.

**Respuesta_usuario(Respuesta, Lugar)**

```
respuesta_usuario(Respuesta, Lugar):- oracion(Respuesta, [Lugar|_]).

respuesta_usuario(Respuesta):- oracion(Respuesta, _).
```

Esta regla se encarga de recibir la respuesta del usuario y llamar la cláusula *oración* encargada de verificar que la respuesta cuente con la estructura correcta. Devuelve el nombre del lugar o establecimiento  dado por el usuario en la respuesta.

**Existe_establecimiento(Establecimiento)**

```
existe_establecimiento(Establecimiento):- establecimiento([Establecimiento]).
```

Verifica si el establecimiento contenido en la respuesta del usuario existe o no.

**Imprimir_ruta([[Origen, _, Distancia, Tiempo_estimado, Tiempo_estimado_en_presa] | Resto ], Distancia_total, Tiempo_estimado_total, Tiempo_estimado_en_presa_total)**

```
imprimir_ruta([[Origen,Destino,Distancia, Tiempo_estimado, Tiempo_estimado_en_presa]|[]], Distancia,
Tiempo_estimado,Tiempo_estimado_en_presa):- write(Origen), write(", "), write(Destino) .


imprimir_ruta([[Origen,_,Distancia, Tiempo_estimado, Tiempo_estimado_en_presa]|Resto], DistanciaTotal,Tiempo_estimadoTotal,
Tiempo_estimado_en_presaTotal):-
                                                    write(Origen), write(", "),
                                                    imprimir_ruta(Resto, X, Y, Z),
                                                    DistanciaTotal is Distancia + X,
                                                    Tiempo_estimadoTotal is Tiempo_estimado + Y,
                                                    Tiempo_estimado_en_presaTotal is Tiempo_estimado_en_presa + Z.
```

Se encarga de imprimir la ruta entre los lugares dados por el usuario. Además, también imprime la suma de la distancia, tiempo total, tiempo en presa total.

**Pronombre**

```
pronombre([pronombre|S],S).


Ejemplo:
    pronombre([yo |[quiero, ir, al supermercado], [quiero, ir, al supermercado]).
```

Contiene un posible pronombre. Se verifica que el primer elemento de la lista sea uno de los pronombres establecidos. Devuelve el resto de la oración.

**Artículo**

```
articulo([articulo|S],S).


Ejemplo:
    pronombre([la |[farmacia], [farmacia]).
```

Contiene un posible artículo. Se verifica que el primer elemento de la lista sea uno de los artículos establecidos. Devuelve el resto de la oración.

**Preposición**

```
preposicion([en|S],S).

Ejemplo:
    preposicion([en ||[cartago], [cartago]).
```

Contiene una posible preposición. Se verifica que el primer elemento de la lista sea uno de las preposiciones establecidas. Devuelve el resto de la oración.

**Verbo**

```
verbo([verbo|S],S).

Ejemplo:
    verbo([estoy ||[en, cartago], [en, cartago]).
```

Contiene un posible verbo. Se verifica que el primer elemento de la lista sea uno de los verbos establecidos. Devuelve el resto de la oración.

**Establecimiento**

```
establecimiento([establecimiento|_]).

Ejemplo:
    establecimiento([supermercado |_ ], _)
```

Contiene un posible establecimiento. Se verifica que el primer elemento de la lista sea uno de los establecimientos definidos. Devuelve el resto de la oración, la cual, en este caso siempre corresponde a una lista vacía porque dentro de la gramática se definió que el establecimiento es lo último de la oración.

**oracion(Oracion, S)**

```
oracion(Oracion, S):- sintagma_verbal(Oracion, S), localizacion(S).
oracion(Oracion, S):- sintagma_nominal(Oracion, Oracion1), sintagma_verbal(Oracion1, S), localizacion(S).
oracion(Oracion, S):- sintagma_nominal(Oracion, Oracion1), sintagma_verbal(Oracion1,Oracion2), sintagma_nominal(Oracion2,S).
oracion(Oracion, S):- sintagma_verbal(Oracion,Oracion1), sintagma_nominal(Oracion1,S).
oracion(Oracion, S):- sintagma_nominal(Oracion, S).
```

Establece las estructuras de las oraciones. Las oraciones están definidas por sintagma verbal o nominal. Oración corresponde a la lista de la oración a analizar.

**sintagma_verbal(Oracion, S)**

```
sintagma_verbal(Oracion, S):- verbo(Oracion,Oracion1), preposicion(Oracion1,S).
sintagma_verbal(Oracion, S):- verbo(Oracion,Oracion1), pronombre_relativo(Oracion1, Oracion2), verbo(Oracion2, Oracion3), preposicion(Oracion3,S).
sintagma_verbal(Oracion, S):- verbo(Oracion,Oracion1), pronombre_relativo(Oracion1, Oracion2), verbo(Oracion2, S).
sintagma_verbal(Oracion,S):- verbo(Oracion, Oracion1), verbo(Oracion1,S).
sintagma_verbal(Oracion,S):- verbo(Oracion, S).
sintagma_verbal(Oracion, S):- verbo(Oracion,Oracion1), verbo(Oracion1, Oracion2), preposicion(Oracion2,S)
```

Estas reglas definen la estructura del sintagma verbal. El sintagma verbal puede estar formado por verbos, preposiciones, pronombres relativos. **Oración** corresponde a la lista de la oración a analizar.

**sintagma_nominal(Oracion, S)**

```
sintagma_nominal(Oracion, S):- pronombre(Oracion,S).
sintagma_nominal(Oracion, S):- pronombre(Oracion,Oracion1), pronombre_reflexivo(Oracion1, S).
sintagma_nominal(Oracion, S):- negacion(Oracion,S).
sintagma_nominal(Oracion,S):- articulo(Oracion, S), establecimiento(S).
```

Estas reglas definen la estructura del sintagma nominal. El sintagma verbal puede estar formado por pronombres, pronombres reflexivos, artículos, establecimiento, negación. **Oración** corresponde a la lista de la oración a analizar.

**arco(CiudadOrigen, CiudadDestino, DistanciaKM, TiempoEstimado, TiempoEstimadoEnPresa)**

```
arco(CiudadOrigen, CiudadDestino, DistanciaKm, TiempoEstimado,
TiempoEstimadoEnPresa).
Ejemplo:
   arco(cartago, taras, 3, 10, 25)
```

Un arco está formado por una ciudad de origen y una de destino, la distancia entre ambas ciudades y el tiempo estimado de viaje nominal y considerando presas.

**existe_lugar(Lugar)**

```
existe_lugar(Lugar):- arco(Lugar,_,_,_,_), !; arco(_,Lugar,_,_,_), !.
```

Esta regla se encarga de verificar si el lugar ingresado se encuentra en la base de conocimientos, es decir, si existe un arco que contenga como ciudad de origen o ciudad de destino el lugar pasado como argumento.

**dame_ruta(Origen, Destinos, Lista_Respuesta)**

```
dame_ruta(Origen, Destinos, Lista_Respuesta):-
   length(Destinos, Longitud),
   Longitud == 1,
   dame_nodo(Destinos, Destino),
   dijkstra(Origen, Destino, [_|Ruta]),
   ruta_directa(Origen, Ruta, Lista_Respuesta), !.
```

Esta regla devuelve una estructura de datos (lista con sublistas) que contiene la mejor ruta a seguir en términos de distancia para ir desde una ciudad de origen a otra de destino, sin considerar puntos intermedios. Las sublistas contienen los arcos que permiten llegar a dicho destino.

**dame_ruta(Origen, Destinos, Lista_Respuesta)**

```
dame_ruta(Origen, Destinos, Lista_Respuesta):-
   length(Destinos, Longitud),
   Longitud > 1,
   dame_ruta_aux(Origen, Destinos, Lista_Respuesta).
```

Esta regla es similar a la anterior, solo que considera puntos intermedios por el cual el usuario desea pasar antes de llegar al destino. Devuelve la misma estructura de datos solo que más extensa.

**dame_ruta_aux(_, [ ], Lista_Respuesta)**

```
dame_ruta_aux(_, [], Lista_Respuesta):- Lista_Respuesta = [], !.
```

Esta regla define la condición de finalización para el caso en que la ruta total considera pasar por puntos intermedios antes llegar a la ciudad de destino.

**dame_ruta_aux(Origen, [Destino|Destinos], Lista_Respuesta)**

```
dame_ruta_aux(Origen, [Destino|Destinos], Lista_Respuesta):-
   dijkstra(Origen, Destino, [_|Ruta]),
   ruta_directa(Origen, Ruta, Lista_Inicial),
   dame_ruta_aux(Destino, Destinos, Lista_Final),
   concatenar(Lista_Inicial, Lista_Final, Lista_Respuesta).
```

Esta regla se encarga de devolver la estructura de datos de la trayectoria total a seguir cuando el usuario desea pasar primero por puntos intermedios.

**ruta_directa(_, [ ], Lista_Respuesta)**

```
ruta_directa(_, [], Lista_Respuesta):- Lista_Respuesta = [], !.
```

Regla que indica la condición de finalización de la estructura de datos que contiene la trayectoria total entre una ciudad de origen y una de destino, considerando ciudades intermedias.

**ruta_directa(Origen, [Destino|Ruta], Lista_Respuesta)**

```
ruta_directa(Origen, [Destino|Ruta], Lista_Respuesta):-
   ruta_directa_aux(Origen, Destino, Lista_Inicial),
   ruta_directa(Destino, Ruta, Lista_Final),
   concatenar(Lista_Inicial, Lista_Final, Lista_Respuesta).
```

Regla que se encarga de formar la estructura de datos de la trayectoria total entre una ciudad de origen y una de destino, considerando ciudades intermedias.

**ruta_directa_aux(Origen, Destino, Ruta)**

```
ruta_directa_aux(Origen, Destino, Ruta):-
```

```
   arco(Origen, Destino, Distancia, Tiempo, TiempoPresa),
   Lista = [Origen, Destino, Distancia, Tiempo, TiempoPresa],
   Ruta = [Lista].
```

Esta regla devuelve el arco en el que se encuentran relacionadas la ciudad de origen y la ciudad de destino pasadas como argumento, siempre y cuando se encuentre definido en la base de conocimientos.

**dame_nodo([Destino|_], Destino)**

```
dame_nodo([Destino|_], Destino).
Ejemplo:
   dame_nodo([heredia|_], heredia).
```

Este hecho permite obtener el primer destino de una lista de destinos al que el usuario desea llegar en primera instancia.

**concatenar**

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

El hecho y regla aquí definido permite concatenar dos listas y devolverla como respuesta.

**dijkstra(Origen, Destino, Ruta)**

```
dijkstra(Origen, Destino, Ruta) :-
   atravesar(Origen),
   rpath([Destino|RutaD], _)->
     reverse([Destino|RutaD], Ruta),!
```

Esta regla representa el algoritmo de dijkstra que se utiliza para encontrar la ruta más corta entre dos nodos de un grafo, en este caso devuelve una lista con las ciudades intermedias o directas que permiten llegar desde un nodo de origen a otro de destino.

**ruta(Origen, Destino, Distancia)**

```
ruta(Origen,Destino,Distancia):-arco(Origen,Destino,Distancia,_,_).
```

La regla verifica que exista al menos un arco en la base de conocimientos (rutas.pl) para poder determinar la ruta más corta en términos de distancia.

**distancia(Desde, Hasta, Distancia)**

```
distancia(Desde,Hasta,Distancia):-ruta(Desde,Hasta,Distancia).
```

Devuelve la distancia existente entre una ciudad de origen y otra de destino que estén conectadas directamente.

**ruta_mas_corta([Cabeza|Ruta], Destino)**

```
ruta_mas_corta([Cabeza|Ruta], Distancia) :-
   rpath([Cabeza|_], D), !, Distancia < D,
   retract(rpath([Cabeza|_],_)),
   assert(rpath([Cabeza|Ruta], Distancia)).
```

Tal y como su nombre lo indica, esta regla devuelve la ruta más corta en ir de una ciudad a otra, analizando ciudades intermedias en el proceso.

**ruta_mas_corta(Ruta, Destino)**

```
ruta_mas_corta(Ruta, Distancia) :-
   assert(rpath(Ruta,Distancia)).
```

Esta regla lo que hace es almacenar la ruta y la distancia en ir de una ciudad a otra para luego evaluar si es la más óptima o corta.

**atravesar(Desde, Ruta, Distancia)**

```
atravesar(Desde, Ruta, Distancia) :-
   distancia(Desde, Hasta, D),
   not(memberchk(Hasta, Ruta)),
   ruta_mas_corta([Hasta,Desde|Ruta], Distancia+D),
   atravesar(Hasta,[Desde|Ruta],Distancia+D).

atravesar(Desde):-
   retractall(rpath(_,_)),
   atravesar(Desde,[],0).
atravesar(_).
```

Como su nombre indica, esta regla atraviesa o evalúa todos los nodos que permiten llegar desde un punto de origen hasta el punto considerado como destino.

## 1.3. Descripción de las estructuras de datos

Un ejemplo de la estructura de datos que se utilizó en este proyecto tiene la forma:
[ [cartago, paraiso, 10.0, 10, 20],
  [paraiso, orosi, 8.0, 8, 16] ]

Esta es la lista respuesta de todo el trayecto que el usuario debe recorrer para ir de una ciudad de origen a otra de destino, considerando o no pasar por punto intermedios. Esta lista está formada por N sublistas y en el caso ejemplo, la primera lista indica que el viaje empieza en Cartago y viaja a Paraíso con una distancia de 10.0km para lo cual va a durar entre 10 y 20 minutos de acuerdo al tráfico, la segunda lista indica que el segundo segmento del viaje empieza en Paraíso y termina en Orosi con una distancia de 8km y que va a durar entre 8 y 16 minutos aproximadamente.

La ruta total que Wazelog imprime cuando el usuario indica correctamente dónde se encuentra, a dónde se dirige y si quiere pasar por puntos intermedios, está representada por la siguiente estructura de datos que se forma a partir de la estructura de datos descrita anteriormente, por lo que un ejemplo de esta sería:
[cartago, paraiso, orosi, 18.0, 18, 36]

Esta lista resume el trayecto total que debe recorrer el usuario para llegar a su punto de destino, indica las ciudades que debe atravesar y la distancia y tiempo total que esto conlleva. En el caso ejemplo, la lista indica que el viaje empieza en cartago, luego se dirige a paraiso y finalmente a orosi; además, la lista indica que la distancia total del recorrido es de 18Km y que se puede durar un total de 18 a 36 minutos considerando el tráfico y las horas pico.

## 1.4. Problemas sin solución

- Si se cierra el editor del entorno de desarrollo de Prolog mientras se está ejecutando el programa se da el problema de que se crea un ciclo infinito, provocando que el editor se quede congelado, para evitar este inconveniente se agrega la opción que si el usuario da como respuesta ***q*** en cualquier momento, la aplicación se detiene.

## 1.5. Problemas encontrados

- Existe un problema a la hora de utilizar tildes, ñ y mayúsculas en las palabras. Por lo que se decidió a que no fueran utilizadas en las oración. Se especifica en el manual de usuario.

- Se encontró un problema con aquellos lugares que su nombre esté formado por dos o más palabras. Se decidió que el usuario debe ingresar el nombre uniendo las palabras con “_ “. Se especifica en el manual de usuario.

## 1.6. Plan de actividades

| Descripción de la tarea | Tiempo estimado | Responsable | Fecha de entrega |
|---|---|---|---|
| Implementar la gramática libre de contexto | 1 día | Sebastián Mora Godínez | Viernes 26 de marzo |
| Implementar interfaz de usuario | 1 día | Sebastián Mora Godínez | Lunes 29 de marzo |
| Implementar funcionalidad para pedir el lugar de origen | 2 horas | Sebastián Mora Godínez | Domingo 28 de marzo |
| Implementar funcionalidad para pedir destinos intermedios | 5 horas | Sebastián Mora Godínez | Domingo 28 de marzo |
| Implementar funcionalidad para pedir lugar de destino | 2 horas | Sebastián Mora Godínez | Domingo 28 de marzo |
| Ver tutorial: Introducción a la inteligencia artificial con el lenguaje prolog | 2 días | Emanuel Marín Gutiérrez | Jueves 25 de marzo |
| Lectura sobre base de datos en prolog | 3 horas | Emanuel Marín Gutiérrez | Sábado 27 de marzo |
| Comprensión gramáticas libres de contexto y usos del corte | 5 horas | Emanuel Marín Gutiérrez | Domingo 28 de marzo |
| Creación base de conocimiento de rutas a manera de ejemplo para pruebas | 30 minutos | Emanuel Marín Gutiérrez | Lunes 29 de marzo |
| Reglas para la manipulación de la base de conocimiento | 1 hora | Emanuel Marín Gutiérrez | Lunes 29 de marzo |
| Implementación algoritmo dijkstra y estructura de dato de la ruta total recorrida | 2 horas | Emanuel Marín Gutiérrez | Miércoles 31 de marzo |
| Pruebas y documentación interna | 2 horas | Emanuel Marín Gutiérrez | Jueves 1 de abril |
| Funcionalidad que imprime la ruta, la distancia, el tiempo de espera y el tiempo de espera en presa. | 3 horas | Sebastián Mora Godínez | Jueves 1 de abril |

## 1.7. Conclusiones

- Se reafirma el conocimiento sobre el paradigma de programación lógico desarrollando una aplicación en Prolog.

- Se utilizó el lenguaje de programación Prolog para desarrollar un Sistema Experto.

- Se aplicaron conceptos propios de la programación lógica adquiridos en la clase facilitando el desarrollo del proyecto.

- Se utilizaron listas como la principal estructuras de datos del programa lo que facilitó el manejo de datos.

## 1.8. Recomendaciones

- Se recomienda utilizar el lenguaje de programación Prolog para crear gramáticas libres de contexto.

- Por la naturaleza y funcionamiento de los sistemas expertos, se recomienda hacer que estos sean muy flexibles entorno al aumento de su base de conocimientos y la manipulación de la misma.

- Se recomienda utilizar el algoritmo Dijkstra para encontrar la ruta más corta, en términos de distancia, tiempo u otras variables, entre dos nodos de un grafo.

## 1.9. Bibliografía

Escrig, T., Pacheco, J. y Toledo, F. (2001). *El lenguaje de programación PROLOG.* Universidad Jaume https://openlibra.com/es/book/el-lenguaje-de-programacion-prolog

MyCyberAcademy (12 de abril de 2013) *Tutorial Prolog - 1 - Bienvenido*. [Archivo de Video].https://www.youtube.com/watch?v=7lwW78BljzI&list=PLHNkID2PAnJmoXlM0MtgMmNVpkarf2Cd4

*SWI-Prolog documentation*. (s. f.). SWI-Prolog documentation. https://www.swi-prolog.org/pldoc/index.html

## 1.10. Bitácora de Sebastián Mora

**Fecha**: 26 de marzo, **hora**: 3:00 pm

**Participantes:** Sebastián Mora

**Descripción:**
Se comienza a desarrollar la gramática libre de contexto. Se establecen los pronombres, verbos, preposiciones, establecimientos, artículos. Además de las estructuras permitidas por el programa de la oración.

**Fecha**: 28 de marzo, **hora**: 5:00 pm
**Participantes:** Sebastián Mora

**Descripción:**
Se comenzó a desarrollar la interfaz de usuario. Se implementa la funcionalidad para pedir el lugar de origen, los destinos intermedios y el lugar final de destino.

**Fecha**: 1 de abril, **hora**: 7:00 pm
**Participantes:** Sebastián Mora

**Descripción:**
El día de hoy se comenzó y terminó la funcionalidad para imprimir las ruta con los lugares dado por el usuario. Se imprime la ruta, la distancia, el tiempo esperado y el tiempo esperado en presa.

**Fecha**: 4 de abril, **hora**: 7:00 pm
**Participantes:** Sebastián Mora

**Descripción:**
El día de hoy agregué nuevas estructuras de oraciones para expandir la gramática permitida de la aplicación.

**Fecha**: 6 de abril, **hora**: 7:00 pm
**Participantes:** Sebastián Mora

**Descripción:**
El día de hoy agregaron nuevos verbos, preposiciones, pronombres para expandir la gramática permitida por el juego.

## 1.11. Bitácora de Emanuel Marín

**Fecha**: 25 de marzo, **hora**: 11:00 am

**Participante:** Emanuel Marín

**Descripción:**
Hoy empecé a ver un tutorial sobre inteligencia artificial con el lenguaje de programación prolog, el tutorial tiene una extensión de 30 videos con una duración promedio de 10 minutos cada uno, establece las bases para comprender el paradigma lógico y desarrollar programas en prolog, al ser este el mayor referente de este paradigma. [Link: https://www.youtube.com/watch?v=7lwW78BljzI&list=PLHNkID2PAnJmoXlM0MtgMmNVpkarf2Cd4].

**Fecha**: 27 de marzo, **hora**: 7:00 pm
**Participante:** Emanuel Marín

**Descripción:**
El día de hoy leí el capítulo 5: Programación Lógica y Base de Datos, del libro Prolog que forma parte de la bibliografía del curso, para comprender más sus usos, creación y manipulación, en conjunto con lo aprendido en el tutorial.

**Fecha**: 28 de marzo, **hora**: 9:00 am
**Participante:** Emanuel Marín

**Descripción:**
Para comprender por completo la especificación de la tarea, me puse a estudiar y practicar las gramáticas libres de texto, los sistemas expertos y el uso del corte en la programación lógica.

**Fecha**: 29 de marzo, **hora**: 3:00 pm
**Participante:** Emanuel Marín

**Descripción:**
Hoy hice una base de conocimientos (grafo) con algunas rutas a manera de ejemplo, para poder hacer pruebas relacionadas con la lógica del Wazelog una vez que se haya desmenuzado la información pertinente con el grafo, hice también algunas reglas para poder obtener información individual de los arcos del grafo (obtener individualmente la ciudad de origen, de destino, la distancia y los tiempos estimados de cada arco).

**Fecha**: 31 de marzo, **hora**: 1:00 pm
**Participante:** Emanuel Marín

**Descripción:**
Hoy implementé el algoritmo de dijkstra tomando como referencia códigos en github y lo utilicé como base principal para la formación de una estructura de datos formada por una

lista con sublistas que contiene la ciudad de origen, la ciudad de destino y la distancia que existe entre ambas, buscando siempre que esta sea la menor (ruta más corta).

**Fecha**: 1 de abril, **hora**: 10:00 am
**Participante:** Emanuel Marín

**Descripción:**
El día de hoy pasé haciendo pruebas del código desarrollado y verifiqué que este cumpla con las necesidades que requiere el archivo wazelog.pl a la hora de solicitar la ruta entre una ciudad de origen y otra de destino, considerando si se necesita o no pasar por ciudades intermedias. Por último hice la documentación interna del código.