UBA – Facultad de Ciencias Exactas y Naturales – Departamento de Computación

Algoritmos y Estructura de Datos I

Primer cuatrimestre de 2022

Versión 0: 28 de mayo de 2022

# Trabajo Práctico de Implementación

# Esperando el Bondi

**Entrega: 10 de Junio de 2022 (hasta las 23:59)**

## 1. Introducción

El Trabajo Práctico de Implementación (TPI) consiste en llevar a código en C++ las funciones propuestas en el TP de Especificación. Para ello deben seguir la especificación de este enunciado, y no la propia que habían realizado en el transcurso del Trabajo Práctico de Especificación. El trabajo práctico se realiza de manera grupal. Pueden ser los mismos grupos del TPE.

## 2. Consignas

- Implementar todas las funciones que se encuentran en el archivo ejercicios.h. Para ello, deberán usar la especificación que se encuentra en la sección 5 del presente enunciado.
- Todos los tests provistos por la cátedra deben funcionar. No puede haber tests que generen error o no se cumplan (Es decir, todos deben dar verdadero).
- Las soluciones sean prolijas: evitar repetir implementaciones innecesariamente y usar adecuadamente funciones auxiliares.
- Extender el conjunto de casos de tests de manera tal de lograr una cobertura de líneas mayor al 95 %. En caso de no poder alcanzarla, explicar el motivo. La cobertura se chequea de manera simple en CLION.
- Respetar los tiempos de ejecución en el peor caso para las siguientes funciones. Escribir una breve justificación con comentarios en el código de cada función:
  - *tiempoTotal*: $O(n)$ donde $n$ representa la longitud del viaje.
  - *distanciaTotal*: $O(n^2)$ donde $n$ representa la longitud del viaje.
  - *recorridoNoCubierto*: $O(n \times m)$ donde $n$ representa la longitud del viaje y $m$ la longitud del recorrido.
- No está permitido el uso de librerías de C++ fuera de las clásicas: **math**, **vector**, **tuple**, las de input-output, etc. Consultar con la cátedra por cualquier librería adicional.

En el `template-alumnos.zip` que se descarguen desde el campus van a encontrar los siguientes archivos y carpetas:

- `definiciones.h`: Aquí están los renombres mencionados arriba junto con la declaración del **enum** Item.
- `ejercicios.cpp`: Aquí es donde van a volcar sus implementaciones.
- `ejercicios.h`: *headers* de las funciones que tienen que implementar.
- `auxiliares.cpp` y `auxiliares.h`: Donde es posible volcar funciones auxiliares.
- `main.cpp`: Punto de entrada del programa.
- `testsEnunciado`: Estos son algunos Tests Suites provistos por la materia. Aquí deben completar con sus propios Tests para lograr la cobertura pedida.
- `lib`: Todo lo necesario para correr Google Tests. Aquí no deben tocar nada.
- `data`: Contiene dos archivos que corresponden a la salida de dos funciones de auxiliares.cpp y que se pueden usar para graficar en un mapa real el recorrido y la grilla (ver sec. 4).
- `CMakeLists.txt`: Archivo que necesita CLion para la compilación y ejecución del proyecto. **NO** deben sobreescribirlo al importar los fuentes desde CLion. Para ello recomendamos:

1. Lanzar el CLION.
2. Cerrar el proyecto si hubiese uno abierto por *default*: `File->Close Project`
3. En la ventana de Bienvenida de CLION, seleccionar `Open`
4. Seleccionar la carpeta del proyecto `template-alumnos`.
5. Si es necesario, cargar el CMakeList.txt nuevamente mediante `Tools->CMake->Reload CMake Project`
6. No olvidarse descomprimir el GTEST en el folder `lib`.

Es importante recalcar que la especificación de los ejercicios elaborada por la materia es la guía sobre la que debe basarse el equipo a la hora de implementar los problemas.

# 3. Entregable

La fecha de entrega del TPI es el **10 de JUNIO de 2022**.

1. Entregar la implementación de las funciones que cumplan el comportamiento detallado en la Especificación de la sección 5. El entregable debe estar compuesto por los archivos fuente modificados por el grupo: `ejercicios.cpp`, `auxiliares.cpp`, `auxiliares.h`, etc., y los casos de test adicionales propuestos por el grupo para lograr la cobertura completa. Por ejemplo, aconsejamos hacer tests en el folder `testsGrupo` y adjuntarlos al entregable.
2. El proyecto debe subirse en un archivo comprimido en la solapa Trabajos Prácticos en la tarea SUBIR TPI. Recuerden sólo subir archivos fuente y *no código compilado* (como la carpeta cmake-build-debug).
3. **Importante: Es condición necesaria que la implementación pase todos los casos de tests provistos en el directorio tests. Estos casos sirven de guía para la implementación, existiendo otros TESTS SUITES *secretos* en posesión de la materia que serán usados para la corrección.**

# 4. Interfaz gráfica

El objetivo de esta sección es aportar las herramientas para visualizar los recorridos y las grillas en un mapa real.
En el **template-alumnos.zip**, van a encontrar una notebook Python, el archivo **DibujarGrilla.ipynb**. Esta notebook se debe correr en COLAB (de google), como indican las siguientes instrucciones:

- Ir a colab.research.google.com/ (con la cuenta de Google logeada)
- Haciendo click en la pestaña Üpload"(derecha de todo), cargar el archivo DibujarGrilla.ipynb
- Subir los archivos de ejemplo:
  - Hacer click en la carpeta del panel izquierdo (ver figura 1)
  - Click derecho, Üpload", y buscar los *.csv. Para el ejemplo de la cátedra se tienen dos archivos en el folder data: grilla_ejemplo.csv, recorridos_ejemplos.csv.
- Ejecutar todas las celdas (en orden).

Nota: la visualización no salió bien en Firefox; sí en Chromium (y sus derivados, Chrome y Edge).
La figura 4 muestra la salida a partir de los dos archivos dados por la cátedra en template-alumnos. Notar que además haciendo click en las celdas podrán ver el nombre asociado.
Para generar sus propias grillas y recorridos, en el archivo auxiliares.cpp se incluyen dos funciones:

1. guardarGrillaEnArchivo: recibe como parámetros de entrada un nombre de archivo y la grilla a grabar.
2. guardarRecorridosEnArchivo: esta función recibe como parámetros un vector de recorridos. La idea es que luego, les otorga un color a cada punto de cada uno de los recorridos. Además recibe el nombre del archivo para almacenar los recorridos en el formato esperados por la notebook de python.

Tener en cuenta que para poder generar el mapa de salida en la notebook con los nuevos datos es preciso: subir los archivos al colab y cambiar los nombres de las variables 'grilla_filename'y 'recorridos_filename'.

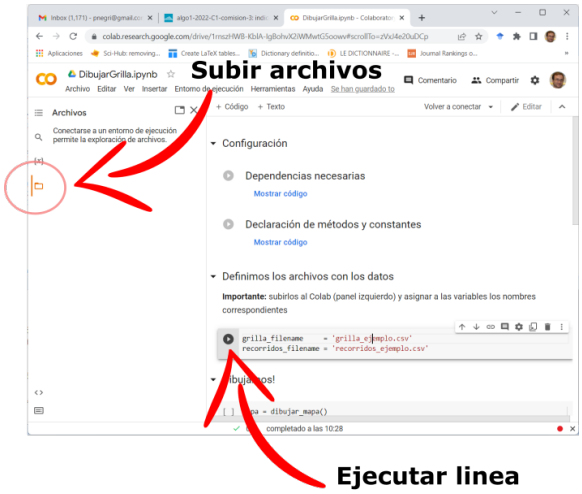


Figura 1: Interfaz de COLAB y botones importantes.

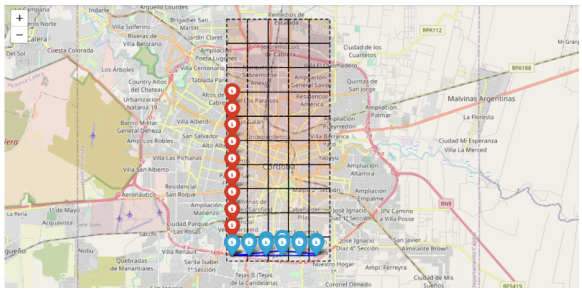

Figura 2: Ejemplo grilla

# 5. Especificación

## 5.1. Tipos

type $Tiempo = \mathbb{R}$

type $Dist = \mathbb{R}$

type $GPS = \mathbb{R} \times \mathbb{R}$

type $Recorrido = seq\langle GPS \rangle$

type $Viaje = seq\langle Tiempo \times GPS \rangle$

type $Nombre = \mathbb{Z} \times \mathbb{Z}$

type $Grilla = seq\langle GPS \times GPS \times Nombre \rangle$

## 5.2. Problemas

**Ejercicio 1.** **proc tiempoTotal**(**in** $v$ : ***Viaje***, **out** $t$ : ***Tiempo***).
Que dado un viaje válido, determine el tiempo total que tardó el colectivo. Este valor debe ser calculado como el tiempo transcurrido desde el primer punto registrado y hasta el último.

```
proc tiempoTotal (in v: Viaje, out t : Tiempo)  {
        Pre {esViajeValido(v)}
        Post {(∃ tmax : Tiempo)(∃ tmin : Tiempo)(maxTiempo(v, tmax) ∧ minTiempo(v, tmin) ∧ t = tmax − tmin}
}
```

**Ejercicio 2. proc distanciaTotal(in $v$ : *Viaje*, out $distancia$ : *Dist*).**
Que dado un viaje válido, determine la distancia recorrida en kilometros aproximada utilizando toda la información registrada en el viaje, es decir, utilizando la información registrada de todos los tramos.

```
proc distanciaTotal( (in v: Viaje, out distancia : Dist) {
    Pre {esViajeValido(v)}
    Post {(∃ vOrd : Viaje)(esViajeOrdenadoPorTiempo(v, vOrd) ∧_L distanciaViaje(vOrd) = d)}
    aux distanciaViaje (v: Viaje) : Dist = Σ_{i=1}^{|v|-1} dist(v[i]_1, v[i-1]_1) ;
}
```

**Ejercicio 3. proc excesoDeVelocidad(in $v$ : *Viaje*, out $res$ : Bool).**
Que dado un viaje válido devuelva verdadero si el colectivo superó los 80 km/h en algún momento del viaje.

```
proc excesoDeVelocidad (in v: Viaje, out res: Bool) {
    Pre {esViajeValido(v)}
    Post {res = true ↔ (∃ vOrd : Viaje)(esViajeOrdenadoPorTiempo(v, vOrd) ∧ superoVelocidad(vOrd))}
    pred superoVelocidad (v: Viaje) {
        (∃ i : ℤ)(0 < i < |v| ∧_L velocidad(v[i-1], v[i]) > 80)
    }
    aux velocidad (p1: Tiempo × GPS, p2: Tiempo × GPS) : ℝ = (dist(gps(p2), gps(p1))/1000)/((tmp(p2) −
    tmp(p1))/3600) ;
}
```

**Ejercicio 4. proc flota(in $V$ : $seq\langle$ *Viaje*$\rangle$, in $t_0$ : *Tiempo*, in $t_f$ : *Tiempo*, out $res$ : $\mathbb{Z}$).**
Que dada una lista de viajes válidos, calcule la cantidad de viajes que se encontraban en ruta en cualquier momento entre $t_0$ y $t_f$ inclusives. Por ejemplo, si un viaje comenzó a las 13:30 y terminó a las 14:30 y la franja es de 14:00 a 15:00, el viaje debería estar considerado. Lo mismo ocurre si el viaje comenzó a las 14:10 y terminó a las 14:15 o si comenzó a las 13:30 y terminó a las 16:00.

```
proc flota (in V: seq⟨Viaje⟩, in t_0 : Tiempo, in t_f : Tiempo, out res : ℤ) {
    Pre {t_f > t_0 ∧ t_0 ≥ 0 ∧ viajesValidos(V)}
    Post {res = Σ_{i=0}^{|v|-1} if viajeEnFranjaHoraria(V[i], t_0, t_f) then 1 else 0 fi}
    pred viajeEnFranjaHoraria (v: Viaje, t_0: Tiempo, t_f: Tiempo) {
        ¬((∃ tmax : Tiempo)(∃ tmin : Tiempo)(maxTiempo(v, tmax) ∧ minTiempo(v, tmin) ∧ tmax < t_0 ∨ tmin >
        t_f)
    }
}
```

**Ejercicio 5. proc recorridoNoCubierto(in $v$ : *Viaje*, in $r$ : *Recorrido*, in $u$ : *Dist*, out $res$ : $seq\langle GPS\rangle$).**
Que dado un viaje $v$ válido, un recorrido $r$ válido y un umbral $u$ (en kilometros), devuelva todos los puntos del recorrido que no fueron cubiertos por ningún punto del viaje. Se considera que un punto $p$ del recorrido está cubierto si al menos un punto del viaje está a menos de $u$ kilometros del punto $p$.

```
proc recorridoNoCubierto (in v: Viaje, in r: Recorrido, in u : Dist, out res : seq⟨GPS⟩) {
    Pre {esViajeValido(v) ∧ esRecorridoValido(r) ∧ u > 0}
    Post {(|res| = cantNoCubiertos(v, r, u)) ∧ noCubiertosEnRes(v, r, u, res)}
    pred cubierto (v: Viaje, u: Dist, p: GPS) {
        (∃ x : GPS)(estaEnViaje(x, v) ∧ dist(x, p) < u)
    }
    pred noCubiertosEnRes (v: Viaje, r: Recorrido, u: Dist, res : seq⟨GPS⟩) {
        (∀ p : GPS)(p ∈ r →_L p ∈ res ↔ ¬cubierto(v, u, g)
    }
    aux cantNoCubiertos (v: Viaje, r: Recorrido, u: Dist) : ℤ = Σ_{i=0}^{|r|-1} if cubierto(v, u, r[i]) then 0 else 1 fi;
}
```

**Ejercicio 6. proc construirGrilla(in $esq1$ : ***GPS***, in $esq2$ : ***GPS***, in $n$ : $\mathbb{Z}$, in $m$ : $\mathbb{Z}$, out $g$ : ***Grilla***).**
Que dados dos puntos GPS, construye una grilla de $n \times m$ (que significa nro. de líneas $\times$ nro. de columnas). Estas grillas están conformadas por celdas contiguas rectangulares. Los lados latitudinales (respectivamente longitudinales) de todas las celdas miden la misma cantidad de grados. Cada celda está caracterizada por sus puntos superior izquierdo e inferior derecho (coordenadas GPS) y un *nombre*, que es un par ordenado de enteros que representa la posición de la celda en la grilla. Estos pares ordenados van desde $(1, 1)$ en el punto que se encuentre en la celda que comienza en la posición $esq1$ y hasta $(n, m)$ en la posición en donde se encuentra la celda con esquina $esq2$. La latitud de $esq1$ debe ser mayor a la latitud de $esq2$ y la longitud de $esq1$ debe ser menor a la longitud de $esq2$

```
proc construirGrilla (in esq1: GPS, in esq2: GPS, in n: ℤ, in m: ℤ, out g: Grilla) {
    Pre {gpsValido(esq1) ∧ gpsValido(esq2) ∧ lat(esq1) > lat(esq2) ∧ lng(esq1) < lng(esq2) ∧ n > 0 ∧ m > 0}
    Post {|g| = n*m ∧_L (todosLosNombresEnGrilla(g, n, m) ∧ todasCeldasConRefCorrectas(g, esq1, esq2, n, m)}
    pred todosLosNombresEnGrilla (g: Grilla, n: ℤ, m: ℤ) {
        (∀ i : ℤ)(∀ j : ℤ)((1 ≤ i ≤ n ∧ 1 ≤ j ≤ m) → nombreEnGrilla(g, i, j)
    }
    pred nombreEnGrilla (g: Grilla, i: ℤ, j: ℤ) {
        (∃ celda : GPS × GPS × Nombre)(celda ∈ g ∧ (celda_2)_0 = i ∧ (celda_2)_1 = j)
    }
    pred todasCeldasConRefCorrectas (g: Grilla, esq1: GPS, esq2: GPS, n: ℤ, m: ℤ) {
        (∀ i : ℤ)(0 ≤ i < |g| ⟶_L celdaConReferenciaCorrecta(g[i], esq1, esq2, n, m))
    }
    pred celdaConReferenciaCorrecta (celda: GPS × GPS × Nombre, esq1: GPS, esq2: GPS, n: ℤ, m: ℤ) {
        lat(celda_0) = lat(esq1) − altoCelda(esq1, esq2, m) * ((celda_2)_0 − 1) ∧
        lat(celda_1) = lat(celda_0) − altoCelda(esq1, esq2, m) ∧
        lon(celda_0) = lon(esq1) + anchoCelda(esq1, esq2, n) * ((celda_2)_1 − 1) ∧
        lon(celda_1) = lon(celda_0) + anchoCelda(esq1, esq2, n)
    }
    aux altoCelda (esq1: GPS, esq2: GPS, n: ℤ) : ℝ = (lat(esq1) − lat(esq2))/n;
    aux anchoCelda (esq1: GPS, esq2: GPS, m: ℤ) : ℝ = (lon(esq2) − lon(esq1))/m;
}
```

**Ejercicio 7.** **proc cantidadDeSaltos**(**in** $g$ : ***Grilla***, **in** $v$ : ***Viaje***, **out** $res : \mathbb{Z}$).
Que dado un viaje válido y una grilla válida, determine cuántos saltos hay en el viaje. Diremos que hay un *salto* si dos puntos del viaje consecutivos temporalmente se encuentran a dos o más celdas de distancia.

```
proc cantidadDeSaltos (in g: Grilla, in v: Viaje, out res : Z) {
    Pre {esViajeValido(v) ∧ grillaValida(g) ∧ viajeEnGrilla(v, g)}
    Post {(∃ N : seq⟨Nombre⟩)(|N| = |v| ∧ esSecuenciaCeldasDeViajeEnGrilla(N, v, g) ∧ res = cantidadSaltos(N))}
    pred esSecuenciaCeldasDeViajeEnGrilla (N: seq⟨Nombre⟩, v: Viaje, g: Grilla) {
        (∃ vOrd : Viaje)(esViajeOrdenadoPorTiempo(v, vOrd) ∧_L
        (∀ i : Z)(0 ≤ i < |N| →_L celdaDeViajeEnGrilla(N[i], vOrd[i]_1, g)))
    }
    pred celdaDeViajeEnGrilla (nombre: Nombre, p: GPS, g: Grilla) {
        (∃ celda : GPS × GPS × Nombre)(celda ∈ g ∧ esCeldaDeCoordenada(p, celda) ∧ celda_2 = nombre)
    }
    pred esCeldaDeCoordenada (p: GPS, celda: GPS × GPS × Nombre) {
        lat(celda_0) ≤ lat(p) < lat(celda_1) ∧ lng(celda_0) ≤ lng(p) < lng(celda_1)
    }
    aux cantidadSaltos (N:seq⟨Nombre⟩) : Z =
    Σ_{i=1}^{|N|-1} if distanciaEntreCeldas(N[i-1], N[i]) > 1 then 1 else 0 fi ;
    aux distanciaEntreCeldas (n1:Nombre, n2:Nombre) : Z = sqrt((n1_0 - n2_0)^2 + (n1_1 - n2_1)^2) ;
}
```

**Ejercicio 8.** **proc corregirViaje**(**inout** $v$ : ***Viaje***, **in** $faltantes : seq\langle\mathbb{Z}\rangle$).
Para este ejercicio se cuenta con un viaje válido de más de 5 puntos, y la lista **errores** que indica cada momento para el cual el valor registrado por el GPS fue erróneo y que debe ser corregido automáticamente. Para la corrección, se buscan los dos puntos más cercanos temporalmente (y correctos), que permiten calcular la velocidad media del vehículo en ese tramo del viaje. Luego, esos dos puntos definen una recta, sobre la cual se va a definir el punto GPS corregido, de acuerdo a la distancia recorrida, usando para ello la velocidad media. Se debe tener en cuenta que la cantidad de puntos a corregir en la lista **errores** no puede superar el 30 % del largo del viaje y que la lista **errores** puede indicar cualquier punto del viaje.

```
proc corregirViaje (inout v: Viaje, in errores: seq⟨Tiempo⟩) {
    Pre {|v| > 5 ∧ |errores| ≤ 0.3 * |v| ∧ semiValido(v, errores) ∧ erroresEnViaje(errores, v) ∧ v = vPrev}
    Post {|v| = |vPrev| ∧_L losCorrectosNoCambian(v, vPrev, errores) ∧_L todosErroresCorregidos(v, vPrev, errores)}
    pred semiValido (v: Viaje, errores: seq⟨Tiempo⟩) {
        tiemposDistintos(v) ∧ (∀ i : Z)(0 ≤ i < |v| →_L (tmp(v[i]) ≥ 0 ∧ (!gpsValido(gps(v[i])) ↔ tmp(v[i]) ∈
        errores)))
    }
    pred erroresEnViaje (errores: seq⟨Tiempo⟩, v: Viaje ) {
        (∀ i : Z)(0 ≤ i < |errores| ⟶_L ((∃k : Z) 0 ≤ k < |v| ∧_L posicionEnInstanteT(v, errores[i], k)))
    }
    pred posicionEnInstanteT (v: Viaje, t: Tiempo, k: Z) {
        tmp(v[k]) = t
    }
    pred todosErroresCorregidos (v: Viaje, errores: seq⟨Tiempo⟩ ) {
        (∀ i : Z)(0 ≤ i < |errores| →_L ((∃k : Z)( 0 ≤ k < |v|
        ∧_L posicionEnInstanteT(v, errores[i], k) ∧ gpsCorregido(v, k, errores) ∧ gpsValido(gps(v[k]))
    }
```

pred gpsCorregido (v: *Viaje*, k: $\mathbb{Z}$, errores: $seq\langle Tiempo\rangle$ ) {

$(\exists\, p : \mathbb{Z})(\exists\, q : \mathbb{Z})(0 \leq p < |v| \wedge 0 \leq q < |v| \wedge p \neq q \wedge_L (gpsCorrecto(v, p, errores) \wedge gpsCorrecto(v, q, errores) \wedge losDosPuntosMasCercanos(v, k, p, q) \wedge gpsSobreRecta(gps(v[k]), gps(v[p]), gps(v[q])) \wedge gpsProporcionalVelocidad(v,$

}

pred gpsCorrecto (v: *Viaje*, i: $\mathbb{Z}$, errores: $seq\langle Tiempo\rangle$) {

$(\forall\, j : \mathbb{Z})(0 \leq j < |errores| \longrightarrow_L errores[j] \neq tmp(v[i]))$

}

pred gpsSobreRecta (x: *GPS*, a: *GPS*, b: *GPS*) {

$(a_0 \neq b_0 \wedge_L x_1 = \frac{(a_1 - b_1)}{(a_0 - b_0)}\ x_0 + \frac{(b_1\ a_0 - a_1\ b_0)}{(a_0 - b_0)}) \vee (a_0 = b_0 \wedge x_0 = a_0)$

}

pred losDosPuntosMasCercanos (v: *Viaje*, k: $\mathbb{Z}$, p:$\mathbb{Z}$, q:$\mathbb{Z}$) {

$(\forall\, i : \mathbb{Z})(0 \leq i < |v| \wedge i \neq k \wedge i \neq p \wedge i \neq q \longrightarrow_L (dist(gps(v[i]), gps(v[k])) > dist(gps(v[p]), gps(v[k])) \wedge dist(gps(v[i]), gps(v[k])) > dist(gps(v[q]), gps(v[k]))))$

}

pred gpsProporcionalVelocidad (v: *Viaje*, k: $\mathbb{Z}$, p: $\mathbb{Z}$, q: $\mathbb{Z}$) {

$moduloCorrectoProporcionalVelocidad(v, k, p, q) \wedge posicionCorrectaEnRecta(v, k, p, q)$

}

pred moduloCorrectoProporcionalVelocidad (v: *Viaje*, k: $\mathbb{Z}$, p: $\mathbb{Z}$, q: $\mathbb{Z}$) {

$dist(gps(v[k]), gps(v[p])) = dist(gps(v[p]), gps(v[q]))^{\frac{(tmp(v[k]) - tmp(v[p]))}{(tmp(v[q]) - tmp(v[p]))}}$

}

# La posicion correcta, verifica las coordenadas del punto corregido respecto al mas cercano que puedo asumir como p (el existe hace que alguna vez el p sea el mas cercano).

pred posicionCorrectaEnRecta (v: *Viaje*, k: $\mathbb{Z}$, p: $\mathbb{Z}$, q: $\mathbb{Z}$) {

$correctoEnRectaOrdinaria(v, k, p, q) \vee correctoEnRectaVertical(v, k, p, q)$

}

pred correctoEnRectaOrdinaria (v: *Viaje*, k: $\mathbb{Z}$, p: $\mathbb{Z}$, q: $\mathbb{Z}$) {

$lat(v[p]) < lat(v[q]) \wedge correctoRespectoPivot(tmp(v[p]), tmp(v[k]), lat(v[p]), lat(v[k]))$

}

pred correctoEnRectaVertical (v: *Viaje*, k: $\mathbb{Z}$, p: $\mathbb{Z}$, q: $\mathbb{Z}$) {

$lat(v[p]) = lat(v[q]) \wedge correctoRespectoPivot(tmp(v[p]), tmp(v[k]), lon(v[p]), lon(v[k]))$

}

pred correctoRespectoPivot (tp: *Tiempo*, tk: *Tiempo*, xp: $\mathbb{R}$, xk: $\mathbb{R}$) {

$sign(tp - tk) = sign(xp - xk)$

}

pred losCorrectosNoCambian (v: *Viaje* , *vPrev*: *Viaje*, errores: $seq\langle Tiempo\rangle$ ) {

$(\forall\, i : \mathbb{Z})(0 \leq i < |v| \rightarrow_L v[i] = vPrev[i] \leftrightarrow tmp(vPrev[i]) \notin errores)$

}

}

## 5.3. Predicados y funciones auxiliares

aux lat (p : *GPS*) : $\mathbb{R}$ $= p_0$ ;

aux lng (p : *GPS*) : $\mathbb{R}$ $= p_1$ ;

aux tmp (medicion : *Tiempo* $\times$ *GPS*) : *Tiempo* $= medicion_0$ ;

`aux gps` (medicion : $Tiempo \times GPS$) : $GPS = medicion_1$ **;**

`pred esTrayectoDeViaje` (t: $seq\langle GPS \rangle$, v: *Viaje*) {

$|t| = |v| \land_L (\forall p : GPS)(p \in t \rightarrow estaEnViaje(p, v))$

}

`pred esViajeValido` (v: *Viaje*) {

$|v| > 2 \land tiemposDistintos(v) \land (\forall i : \mathbb{Z})(0 \leq i < |v| \rightarrow_L (t(v[i]) \geq 0 \land gpsValido(gps(v[i]))))$

}

`pred esRecorridoValido` (r: *Recorrido*) {

$|r| > 0 \land (\forall_1 : \mathbb{Z})(0 \leq i < |r| \rightarrow_L gpsValido(r[i]) \land \#apariciones(r[i], r) = 1)$

}

`pred tiemposDistintos` (v: *Viaje*) {

$(\forall i : \mathbb{Z})(\forall j : \mathbb{Z})((0 \leq i < |v| \land 0 \leq j < |v| \land i \neq j) \rightarrow_L t(v[i]) \neq t(v[j])$

}

`pred gpsValido` (g: *GPS*) {

$-90 \leq lat(g) \leq 90 \land -180 \leq lng(g) \leq 180$

}

`pred maxTiempo` (v: *Viaje*, m: $\mathbb{R}$) {

$(\exists pos : \mathbb{Z})(0 \leq pos < |v| \land_L t(v[pos]) = m) \land (\forall\ i : \mathbb{Z})(0 \leq i < |v| \rightarrow_L m \geq t(v[i]))$

}

`pred minTiempo` (v: *Viaje*, m: $\mathbb{R}$) {

$(\exists pos : \mathbb{Z})(0 \leq pos < |v| \land_L t(v[pos]) = m) \land (\forall\ i : \mathbb{Z})(0 \leq i < |v| \rightarrow_L m \leq t(v[i]))$

}

`pred esViajeOrdenadoPorTiempo` (v: *Viaje*, vOrd: *Viaje*) {

$mismoViaje(v, vOrd) \land ordenadoPorTiempo(vOrd)$

}

`pred mismoViaje` (v: *Viaje*, vOrd: *Viaje*) {

$|v| = |vOrd| \land (\forall x : Tiempo \times GPS)(\#apariciones(x, v) = \#apariciones(x, vOrd))$

}

`pred ordenadoPorTiempo` (v: *Viaje*) {

$(\forall i : \mathbb{Z})1 \leq i < |v| \rightarrow_L t(v[i]) > t(v[i-1])$

}

`pred estaEnViaje` (g: *GPS*, v: *Viaje*) {

$(\exists\ i : \mathbb{Z})(0 \leq i < |v| \land_L gps(v[i]) = g)$

}

`pred grillaValida` (g: *Grilla*) {

$(\exists n : \mathbb{Z})(\exists m : \mathbb{Z})(\exists esq1 : GPS)(\exists esq2 : GPS)(gpsValido(esq1) \land$
$gpsValido(esq2) \land lat(esq1) < lat(esq2) \land lng(esq2) < lng(esq1) \land n > 0 \land m > 0$
$\land |g| = n * m \land todosLosNombresEnGrilla(g, n, m) \land todasCeldasConReferenciasCorrectas(g, esq1, esq2, n, m))$

}

`pred viajeEnGrilla` (v: *Viaje*, g: *Grilla*) {

$(\exists\ trayecto : seq\langle GPS \rangle)(esTrayectoDeViaje(trayecto, v) \land trayectoEnGrilla(trayecto, g))$

}

`pred viajesValidos` (V: $seq\langle Viaje \rangle$) {

$(\forall i : \mathbb{Z})(0 \leq i < |V| \rightarrow_L esViajeValido(V[i]))$

}

`pred trayectoEnGrilla` (trayecto: $seq\langle GPS\rangle$, g: $Grilla$) {

$(\forall t : GPS)(t \in trayecto \rightarrow hayCeldaParaCoord(t, g))$

}

`pred hayCeldaParaCoord` (t: $GPS$, g: $Grilla$) {

$(\exists celda : GPS \times GPS \times Nombre)(c \in g \wedge esCeldaDeCoordenada(t, celda))$

}