Algoritmos y Estructuras de Datos - DC - UBA

Segundo cuatrimestre de 2024

Trabajo práctico 1: Especificación y WP

Normativa

Límite de entrega: Domingo 15 de Septiembre a las 23:55hs.

Normas de entrega: Subir el pdf a la tarea del campus.

Enunciado: "En búsqueda del camino".

En la vida cotidiana, a menudo necesitamos encontrar la forma más rápida o conveniente de llegar de un lugar a otro. Por ejemplo, cuando usamos aplicaciones de mapas, estas nos muestran la mejor ruta desde nuestra casa hasta el destino deseado, considerando distancias o el tiempo que tardaremos en llegar. Esta idea de buscar el "camino más corto" entre dos puntos es un problema clásico de optimización, cuyas soluciones algorítmicas se remontan a los años 50 con el algoritmo de Dijkstra (1956), el cual sigue en uso hoy en día.[1] El mismo se basa en la idea de ir explorando las rutas posibles desde un punto de origen, y seleccionar la que minimice la distancia recorrida, hasta llegar al destino. No obstante, este algoritmo no es el único existente. En 1958, Richard Bellman propuso un algoritmo que, a diferencia del de Dijkstra y a costa de más pasos de cómputo, no requiere que las distancias sean positivas, lo cual lo hace más general.[2]

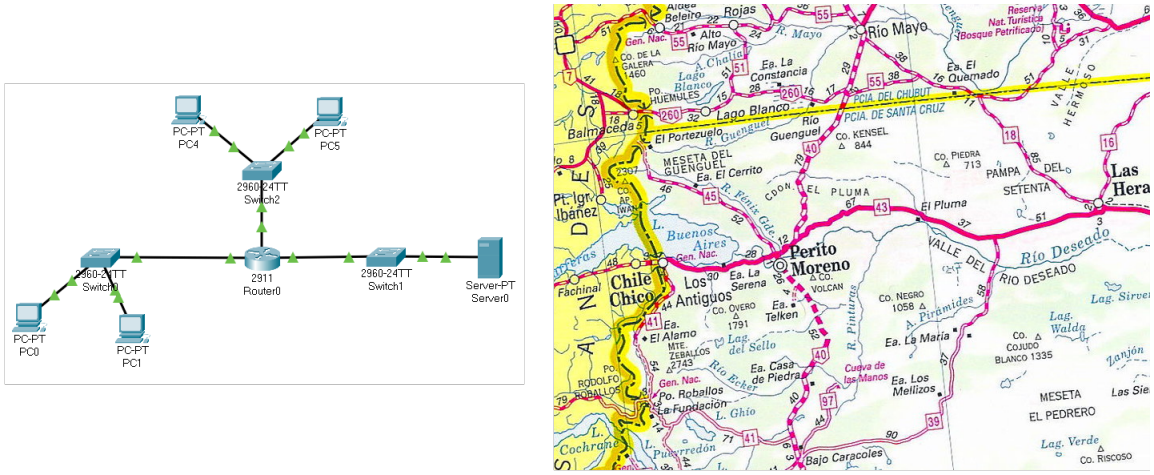


Figura 1: Mapa de enrutamiento entre nodos (izquierda) y fracción del mapa de Santa Cruz (derecha).

Las aplicaciones de estos algoritmos son variadas y no se restringen únicamente a la búsqueda de caminos en mapas. En el campo de las redes informáticas, por ejemplo, es de suma importancia encontrar la forma más eficiente de enrutar paquetes de datos entre dos computadoras. Los paquetes "saltan" de un nodo al otro y se busca minimizar la cantidad de saltos que se realizan (Fig. 1). El protocolo de enrutado OSPF (Open Shortest Path First) se basa en el algoritmo de Dijkstra para encontrar el camino más corto entre dos nodos y se actualiza en tiempo real a medida que cambian las condiciones de la red (por ejemplo, si un nodo se desconecta de la red y deja de estar disponible).[3]

En este trabajo práctico, exploraremos cómo especificar, en un lenguaje formal, algunos problemas relacionados a ciudades y sus habitantes, incluyendo la búsqueda de caminos en sus distintas variantes.

Consignas

Una Ciudad se define como una tupla <nombre, habitantes>, donde *nombre* es un string y *habitantes* es un número entero que representa la cantidad de personas que viven en la ciudad. Cuando nos referimos a las ciudades como puntos en el mapa, las identificaremos a partir de números naturales.

[1] `https://ir.cwi.nl/pub/9256/9256D.pdf`

[2] `https://www.ams.org/journals/qam/1958-16-01/S0033-569X-1958-0102435-2/S0033-569X-1958-0102435-2.pdf`

[3] `https://www.rfc-editor.org/rfc/rfc1245`

1. Especificación

1. **grandesCiudades**: A partir de una lista de ciudades, devuelve aquellas que tienen más de 50.000 habitantes.

   `proc grandesCiudades` (in *ciudades*: $seq\langle Ciudad\rangle$) : $seq\langle Ciudad\rangle$

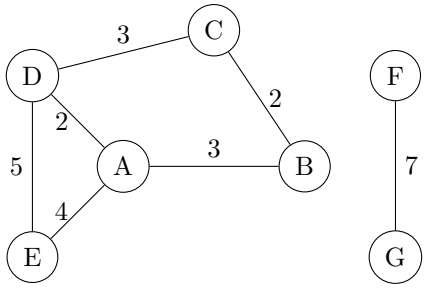
2. **sumaDeHabitantes**: Por cuestiones de planificación urbana, las ciudades registran sus habitantes mayores de edad por un lado y menores de edad por el otro. Dadas dos listas de ciudades del mismo largo con los mismos nombres, una con sus habitantes mayores y otra con sus habitantes menores, este procedimiento debe devolver una lista de ciudades con la cantidad total de sus habitantes.

   `proc sumaDeHabitantes` (in *menoresDeCiudades*: $seq\langle Ciudad\rangle$, in *mayoresDeCiudades*: $seq\langle Ciudad\rangle$) : $seq\langle Ciudad\rangle$

3. **hayCamino**: Un mapa de ciudades está conformada por ciudades y caminos que unen a algunas de ellas. A partir de este mapa, podemos definir las distancias entre ciudades como una matriz donde cada celda i, j representa la distancia entre la ciudad i y la ciudad j (Fig. 2). Una distancia de 0 equivale a no haber camino entre i y j. Notar que la distancia de una ciudad hacia sí misma es cero y la distancia entre A y B es la misma que entre B y A.

   Dadas dos ciudades y una matriz de distancias, se pide determinar si existe un camino entre ambas ciudades.

   `proc hayCamino` (in *distancias*: $seq\langle seq\langle\mathbb{Z}\rangle\rangle$, in *desde*: $\mathbb{Z}$, in *hasta*: $\mathbb{Z}$) : Bool



	A	B	C	D	E	F	G
A	0	3	0	2	4	0	0
B	3	0	2	0	0	0	0
C	0	2	0	3	0	0	0
D	2	0	3	0	5	0	0
E	4	0	0	5	0	0	0
F	0	0	0	0	0	0	7
G	0	0	0	0	0	7	0

Figura 2: Mapa de ciudades con su correspondiente matriz de distancias.

4. **cantidadCaminosNSaltos**: Dentro del contexto de redes informáticas, nos interesa contar la cantidad de "saltos" que realizan los paquetes de datos, donde un *salto* se define como pasar por un nodo.

   Así como definimos la matriz de distancias, podemos definir la matriz de conexión entre nodos, donde cada celda i, j tiene un 1 si hay un único camino a un salto de distancia entre el nodo i y el nodo j, y un 0 en caso contrario. En este caso, se trata de una matriz de conexión de orden 1, ya que indica cuáles pares de nodos poseen 1 camino entre ellos a 1 salto de distancia.

   Dada la matriz de conexión de orden 1, este procedimineto debe obtener aquella de orden n que indica cuántos caminos de n saltos hay entre los distintos nodos. Notar que la multiplicación de una matriz de conexión de orden 1 consigo misma nos da la matriz de conexión de orden 2, y así sucesivamente.[4]

   `proc cantidadCaminosNSaltos` (inout *conexión*: $seq\langle seq\langle\mathbb{Z}\rangle\rangle$, in n: $\mathbb{Z}$)

5. **caminoMínimo**: Dada una matriz de distancias, una ciudad de origen y una ciudad de destino, este procedimiento debe devolver la lista de ciudades que conforman el camino más corto entre ambas. En caso de no existir un camino, se debe devolver una lista vacía.

   `proc caminoMínimo` (in *origen*: $\mathbb{Z}$, in *destino*: $\mathbb{Z}$, in *distancias*: $seq\langle seq\langle\mathbb{Z}\rangle\rangle$) : $seq\langle\mathbb{Z}\rangle$

[4] https://www.um.edu.mt/library/oar/bitstream/123456789/24439/1/powers%20of%20the%20adjacency%20matrix.pdf

2. Demostraciones de correctitud

La función **poblaciónTotal** recibe una lista de ciudades donde al menos una de ellas es grande (es decir, supera los 50.000 habitantes) y devuelve la cantidad total de habitantes. Dada la siguiente especificación:

$$\begin{aligned}&\texttt{proc poblaciónTotal (in } ciudades\texttt{: } seq\langle Ciudad\rangle\texttt{)} : \mathbb{Z}\\ &\quad \texttt{requiere } \{(\exists i : \mathbb{Z})(0 \leq i < |ciudades| \wedge_L ciudades[i].habitantes > 50{,}000) \wedge\\ &\qquad\qquad (\forall i : \mathbb{Z})(0 \leq i < |ciudades| \longrightarrow_L ciudades[i].habitantes \geq 0) \wedge\\ &\qquad\qquad (\forall i : \mathbb{Z})(\forall j : \mathbb{Z})(0 \leq i < j < |ciudades| \longrightarrow_L ciudades[i].nombre \neq ciudades[j].nombre)\}\\ &\quad \texttt{asegura } \{res = \sum_{i=0}^{|ciudades|-1} ciudades[i].habitantes\}\end{aligned}$$

Con la siguiente implementación:

```
res = 0
i = 0
while (i < ciudades.length) do
    res = res + ciudades[i].habitantes
    i = i + 1
endwhile
```

1. Demostrar que la implementación es correcta con respecto a la especificación.
2. Demostrar que el valor devuelto es mayor a 50.000.