Informe de proyecto semestral

Encontrando la ruta más corta entre dos o tres direcciones para desplazarse en el centro de Concepción

Matemáticas Discretas 2023-2

Profesor Pierluigi Cerulo

Integrantes:

Carlos Álvarez Norambuena — DIICC Ingeniería Civil Informática

Benjamín Espinoza Henriquez — DIICC Ingeniería Civil Informática

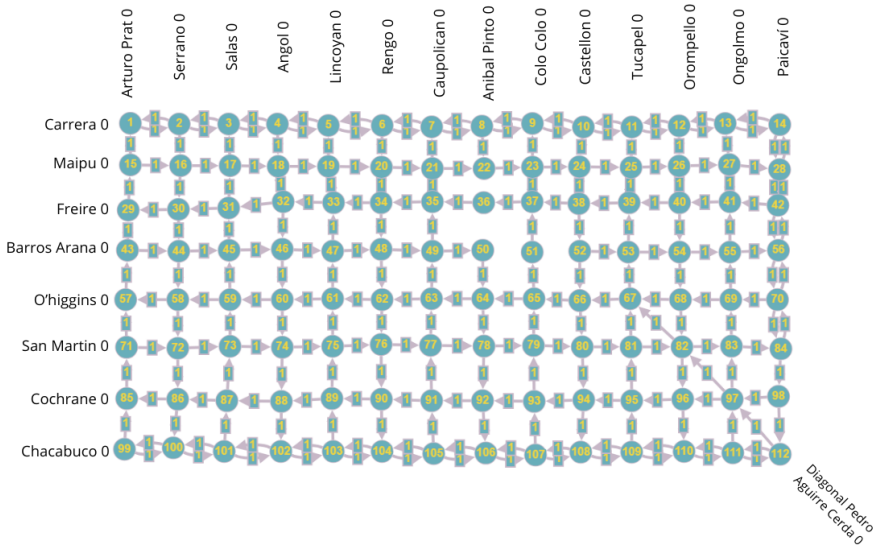
Sofía López Aguilera — DIICC Ingeniería Civil Informática

1. Objetivos

Los objetivos de este proyecto estuvieron desde un principio marcados por aprender a implementar la teoría de grafos dirigidos a la programación, específicamente hablando sobre el algoritmo de Dijkstra, ya que es una excelente vía altamente utilizada para saber el recorrido más corto de un grafo dirigido.
Nuestra meta principal fue comprender, tanto conceptual como prácticamente, el funcionamiento del algoritmo de Dijkstra. Esto incluyó explorar los distintos métodos para su implementación, así como analizar las ventajas y desventajas de cada enfoque. Además, buscamos aplicar este conocimiento en un contexto cercano y común, como es el caso de este proyecto.
Por último con este trabajo nos gustaría ver una aplicación de los conceptos vistos en clases a un caso práctico que podría surgir en un desarrollo de software real.

2. Modelo

Comenzamos haciendo un dibujo del digrafo con el que queríamos representar las calles del centro de Concepción usando la página de internet *graphonline.ru*. El grafo generado fue el siguiente:



Es importante recalcar que a este digrafo le añadimos que calle representa cada una de las intersecciones. Por ejemplo, en esta imagen el nodo 16 sería la intersección de Serrano con Maipú. Notemos también que las calles de Concepción forman algo similar a un plano cartesiano. Con las posiciones en el eje X para las calles horizontales y las posiciones en el eje Y para las calles verticales. Por ejemplo, podríamos decir que en coordenadas cartesianas la posición 0, 0 sería el nodo 1; la posición 3, 5 sería el nodo 74. Estos 2 sistemas de ejes los utilizamos ambos en nuestra implementación. Esta página nos dejó también generar la matriz de adyacencia asociada a este dígrafo. Posteriormente, copiamos la matriz y la utilizamos directamente en nuestro código. Así podemos determinar fácilmente si hay 2 nodos que son adyacentes.

3. Implementación

4.1 Entradas válidas para el programa

Todas las direcciones deben ir separadas de espacios y con el número que se desee. Si hay alguna duda con el uso del programa, por favor acceda al README incluido en el proyecto.

Ejemplo de uso (en negrita va a aparecer lo que un supuesto usuario tendría que añadir):

Bienvenido/a a nuestro Buscador de Rutas por el centro de Concepcion.
Por favor, seleccione la direccion de partida: ***Cochrane 212***
Por favor, seleccione la direccion de destino: ***Castellon 200***
¿Desea hacer una parada? [y/n]: ***y***
Por favor, ingrese la calle donde desea parar: ***Carrera 121***

En la sección de arriba se omiten deliberadamente los tildes ya que estos no funcionan correctamente en C++.

4.2 Matriz de adyacencia

La matriz de adyacencia fue añadida en un archivo constants.cpp, aunque cabe recalcar que hay unas pequeñas diferencias a nivel de implementación. Una de esas diferencias es que nuestros nodos los enumeramos en nuestro programa desde el 0 y no desde el 1 como lo hicimos en nuestro grafo inicial. Esto debido a que C++ comienza a indexar las matrices desde el 0 y nos permite hacer unas transformaciones de las coordenadas cartesianas al número de nodo más fácilmente.

4.3 Pseudocódigo

En cuanto al pseudocódigo, presentaremos un esquema de la totalidad del código main y luego una sección donde se explican en detalle las funciones que se llaman en este segmento. Es importante que cada parte del pseudocódigo de main.cpp que está recalcada en negrita se verá anexada en la sección 4.3.2

Observación: Nótese que cada referencia dicha en esta sección como "coordenadas cartesianas" se refiere a nuestro propio modelo del mapa de Concepción. Así como también se puede ver el concepto "coordenada (nodo)" como el número de nodo del modelo.

4.3.1 Pseudocódigo de main.cpp

```
Se ingresa calle_partida
Se ingresa calle_destino

// Con el fin de luego ingresar estos datos  al algoritmo de
// Dijkstra (el cual recibe 2 parámetros) hacemos el siguiente // swap:

Si <existe parada> Entonces:
      Se ingresa calle de parada
      swap(calle_destino, calle_parada)
finSi


Repetir
<
      Se define distancia_primer_nodo
      Se ejecuta coordenada_string_cartesiana(calle_partida) y se guarda en
      calle_partida_xy
      Se ejecuta coordenada_string_cartesiana(calle_destino) y se guarda en
      calle_destino_xy

      Si <cualquier coordenada de calle_partida = -1> Entonces:
            Lanza error de input y sale del código
      finSi

      Si <cualquier coordenada de calle_destino = -1> Entonces:
            Lanza error de input y sale del código
      finSi

      Se definen nodo_partida1, nodo_partida2, nodo_destino1, nodo_destino2 que
      son cuatro posibles nodos desde donde puede partir y finalizar Dijkstra
      Se define resultado_dijkstra
```

Si <(existe arco entre nodo_partida2 y nodo_partida1) y (existe arco entre nodo_destino1 y nodo_destino2)> Entonces:
Se ejecuta **dijkstra(**nodo_partida1, nodo_destino1**)**y se almacena en resultado_dijkstra
Se calcula distancia de dirección inicial al primer nodo y se almacena en distancia_a_primer_nodo
Se calcula la suma de la distancia del camino calculado a partir de los los nodos más distancia_a_primer_nodo
Si <respuesta1 es menor que menor_distancia> Entonces
Se establece este caso como respuesta
finSi
finSi

Se realiza esta misma condición 3 veces más, cambiando lo recalcado en amarillo para abarcar todos los posibles casos de distancia.
Se guarda distancia_primer_nodo en distancias_a_primer_nodo
Se establece se_repite_loop en false

Si <existe una calle de parada> Entonces:
Se intercambian la calle de inicio por la calle de parada y la calle de destino vuelve a su valor original
Se guarda en una auxiliar el camino desde el inicio a la parada junto a todos los nodos involucrados
Se establece se_repite_loop en true
Se guarda menor_distancia en el arreglo menor_distancia_por_recorrido
Se guarda nodo_partida en el arreglo nodos_partida
Se guarda nodo_destino en el arreglo nodos_destino
finSi
>
Hasta que <se_repite_loop sea verdadero>

Se guardan la distancia del recorrido en el arreglo de distancias menor_distancia_por_recorrido
Se guarda nodo_partida en el arreglo nodos_partida
Se guarda nodo_destino en el arreglo nodos_destino
Se concatena el camino de la partida a la parada junto al camino de la parada al destino y se almacena en respuesta_mas_corta
Se accede a los nodos previos de nodos_llegada y se guardan en la variable recorrido1
Se crea un iterador con el último nodo del camino de partida a parada (en caso de existir la parada)

Mientras <iterador sea distinto a el primer nodo de nodos_partida> Hacer:

```
      Se agrega a recorrido1 el nodo encontrado en la lista de nodos de
      respuesta_mas_corta en la posición iterador
finMientras
Antes de la impresión de resultados se pasan las listas de nodos a String con
la función cartesiana_a_string iterando las veces que haga falta para completar
el camino en cuestión.
Finalmente, se imprimen los resultados iterando sobre el tamaño de recorrido1
en caso de no haber parada.

Si <existe parada> Entonces:
      Se define un arreglo de enteros recorrido2
      Se accede a los nodos previos de nodos_llegada y se guardan en la
      variable recorrido2
      Se crea un iterador con el último nodo del camino de parada a destino
      Mientras <iterador sea distinto a el primer nodo de nodos_partida> Hacer:
            Se agrega a recorrido2 el nodo encontrado en la lista de nodos de
            respuesta_mas_corta en la posición iterador

      finMientras
finSi

Antes de la impresión de resultados se pasan las listas de nodos a String con
la función cartesiana_a_string iterando las veces que haga falta para completar
el camino en cuestión.
Finalmente, se imprimen los resultados iterando sobre recorrido2 en caso de
haber parada.
```

4.3.2 Funciones utilizadas en el pseudocódigo y su explicación.

Funcion	Parámetros	Explicación del código
coordenada_cartesiana_nodo	int x, int y	Función que calcula el nodo en función de sus coordenadas cartesianas
coordenada_string_cartesiana	string input	Función que transforma un string con el input del usuario y lo transforma en un par de coordenadas cartesianas para poder trabajar con estas
coordenada_nodo_cartesiana	int nodo	Función que transforma un número de nodo a un par con las coordenadas cartesianas de este nodo

cartesiana_a_string	int x, int y	Función que transforma de coordenadas cartesianas a un string con la intersección que se genera en ese nodo
retorna_numero_de_string	string input	Función que retorna el número que se ingresó junto a la calle en forma de entero
dijkstra	int inicio, int final	Función que emplea el algoritmo de Dijkstra entre inicio e final

Nuestro proyecto fue implementado en C++. El compilador utilizado fue g++ (GCC) 13.2.1 e hicimos nuestras pruebas en Manjaro linux con versión de kernel 5.15.130.

4. Conclusiones

Como equipo pensamos que se logró implementar con éxito la teoría de grafos dirigidos, específicamente el algoritmo de Dijkstra, en la programación. Se alcanzaron los objetivos de comprender su funcionamiento, explorar métodos de implementación y aplicar este conocimiento al mapa del centro de Concepción.

Respecto al rendimiento del código, notamos que hay muchos aspectos que podrían ser optimizados. Una reflexión importante para nosotros fue que nos percatamos de que habría maneras más efectivas para resolver este problema, quizás usando listas de adyacencia en lugar de la matriz, lo cual ralentiza un poco el proceso de compilación. A pesar de que nos percatamos de estas problemáticas a tiempo, no nos vimos con la capacidad de modificarlo por temas de tiempo de desarrollo.

En conclusión, el proyecto cumplió exitosamente con sus metas al abordar la implementación del algoritmo de Dijkstra en un contexto práctico, superando desafíos técnicos. Se realizó con éxito el objetivo principal de la tarea que fue encontrar el camino más corto dadas dos o tres direcciones del centro de Concepción.