Análisis y Diseño de Algoritmos

Tarea 6

Daniel Rojo Mata
danielrojomata@gmail.com

Fecha de Entrega: 20 de octubre de 2024

Problema 1

Escribe un resumen de una página de la siguiente biografía de Edsger W. Dijkstra: https://www.cwi.nl/en/about/history/e-w-dijkstra-brilliant-colourful-and-opinionated/. Incluye algún aspecto de la obra de Dijkstra, o de su forma de trabajar, que crees que puede ser relevante o ayudarte en tu carrera como científic@ de la computación.

Resumen/Comentarios

Edsger W. Dijkstra fue una figura influyente en el campo de las ciencias de la computación, reconocido por sus valiosas contribuciones que han dejado un impacto muy grande en diversas áreas. Su trabajo se caracterizaba por un enfoque innovador y por una claridad que le permitió resolver problemas complejos de forma elegante y comprensible.

Uno de sus logros más destacados fue la creación del *Algoritmo de Dijkstra*, un algoritmo diseñado para calcular el camino más corto entre dos nodos en un grafo. Esta solución surgió en solo 20 minutos (según cuentan las historias), motivada por la pregunta: '¿Cuál es la ruta más corta para viajar de Rotterdam a Groningen?'. Publicado en 1959, el algoritmo se convirtió en una piedra angular en la teoría de grafos y ha sido ampliamente utilizado en aplicaciones de redes y sistemas de navegación.

Además de sus contribuciones en teoría de grafos, Dijkstra también dejó su huella en el desarrollo del lenguaje de programación *ALGOL 60*. Aunque inicialmente su nombre no apareció junto con los demás creadores debido a diferencias creativas, colaboró con un colega para escribir un compilador que incluía una nueva implementación de la recursión. Estas mejoras fueron importantes para la evolución de los lenguajes de programación modernos.

Otro de sus aportes fue en el ámbito de la programación concurrente. En 1968, Dijkstra publicó un artículo que dio origen a conceptos fundamentales para la programación de procesos concurrentes, estableciendo bases teóricas que han permitido el desarrollo de sistemas multitarea. En 1971, ilustró el problema del *deadlock* a través del famoso *philosophers problem*, que se ha utilizado desde entonces como una referencia para explicar primitivas de sincronización y gestión de recursos compartidos.

Como profesor en la Universidad de Texas, Dijkstra no solo se destacaba por sus conocimientos técnicos, sino también por su estilo didáctico. Pedía a sus estudiantes que escribieran pruebas para problemas

matemáticos elementales que discutían en clase, y luego los evaluaba con comentarios incisivos pero humorísticos como 'Muchos pecados de omisión'. Esta combinación de rigor académico y un enfoque directo para identificar errores ayudaba a sus estudiantes a comprender la importancia de la precisión y la claridad en las demostraciones matemáticas.

Su manera de plantear las pruebas matemáticas era clara y accesible, sin dejar a un lado el rigor necesario. A menudo lograba expresar conceptos complejos sin recurrir a una gran cantidad de fórmulas, permitiendo que sus ideas fueran comprendidas de manera más intuitiva.

Personalmente, me gusta la filosofía de trabajo que implementaba Dijkstra, ya que demuestra que es posible comunicar ideas complejas de manera sencilla, manteniendo al mismo tiempo la precisión, la coherencia y el rigor matemático que se requiere. Esta perspectiva es una habilidad valiosa para cualquier científico de la computación (y persona no ajena a un área de investigación), y es algo que pretendo aplicar en mi propia carrera.

Problema 2

Muestra que el algoritmo de caminos más cortos de Dijkstra puede no ser correcto si las longitudes en las aristas son negativas. Es decir, muestra una gráfica dirigida con algunas longitudes negativas en las aristas, y una ejecución de Dijkstra que no calcula el camino más corto de por lo menos un par de vértices.

Además, explica en donde falla la prueba de corrección de Dijkstra vista en clase cuando existen aristas de longitud negativa.

Solución

La gráfica dirigida propuesta es la que se muestra en la figura 1.

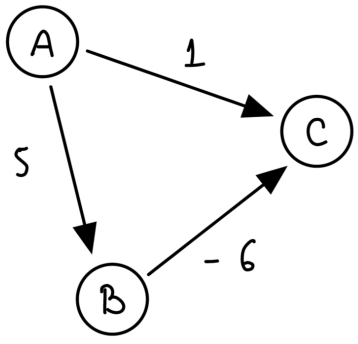

Figura 1: Gráfica con pesos negativos en donde Dijkstra falla.

Se aplica el algoritmo desde A, por lo que la distancia del nodo A a A será 0; por lo tanto, $\texttt{dist}[A] = 0$.

Se establecen las distancias iniciales de todos los nodos, excepto el origen, en infinito.
Por lo que, $\texttt{dist}[B] = \infty$ y $\texttt{dist}[C] = \infty$.

La cola de prioridad (`PQ`) contendrá pares que consisten en {distancia de un nodo desde A, nodo}.

Inicialmente, $\texttt{pq} = [(0, A), (\infty, B), (\infty, C)]$.
Se extrae el par $(0, A)$ de la cola de prioridad porque está en la parte superior y tiene la distancia más

pequeña.

Después de esto, `pq` $= [(\infty, B), (\infty, C)]$.

A tiene una arista hacia B y C, por lo tanto, se actualiza la distancia de B y C.

- `dist`$[B] = 0 + 5 = 5$
- `dist`$[C] = 0 + 1 = 1$.

Ahora, `pq` $= [(1, C), (5, B)]$.
Se extrae el par $(1, C)$, ahora `PQ` $= [(5, B)]$. C no tiene aristas hacia ningún nodo, por lo que el algoritmo continúa.

Ahora, se extrae el último par $(5, B)$.

Nótese que B tiene una arista hacia C con peso -2, pero dado que C no está en `PQ`, significa que la distancia de C ya está calculada. Por lo tanto, no se actualiza la distancia de C.

Ahora, la cola está vacía y se ha calculado la distancia más corta de cada nodo: `dist`$[A] = 0$, `dist`$[B] = 5$ y `dist`$[C] = 1$.

Si se observa, en lugar de visitar C como $A \to C$, se podría haber visitado C como $A \to B \to C$. De esta forma, la distancia habría sido $5 + (-6) = -1$. Por lo tanto, el algoritmo de Dijkstra ha fallado en calcular la respuesta correcta.

Esto ocurre porque, en cada iteración, el algoritmo solo actualiza la respuesta para los nodos en la cola. Así, el algoritmo de Dijkstra no reconsidera un nodo una vez que lo marca como visitado, incluso si existe un camino más corto que el anterior.

Por lo tanto, el algoritmo de Dijkstra falla en grafos con aristas de peso negativo.

El algoritmo falla debido a que al asumir que la distancia a cada nodo es mínima cuando se extrae de la cola de prioridad, se ignora la posibilidad de caminos más cortos que puedan surgir debido a aristas de peso negativo.

Problema 3

Diseña un algoritmo de tiempo $O(|V||E|\log(|V|))$ que, dada una gráfica dirigida G con longitudes no-negativas en sus aristas, calcule la longitud de los ciclos (dirigidos) más cortos en G, y devuelva uno de esos ciclos. Prueba la corrección y haz el análisis de complejidad de tu algoritmo.

Solución:

EncontrarCicloMásCorto(G)

- `Entrada: Entrada:` $G = (V, E, \ell)$ `donde` V `son los nodos,` E `son las aristas, y` ℓ `representa las longitudes de las aristas`
- `Salida: Salida: El ciclo dirigido más corto y su longitud`

```
1: min_ciclo_longitud ← ∞
2: mejor_ciclo ← NULO
3: for each s ∈ V do                                  ▷ Ejecutar Dijkstra desde el nodo s
4:     π, pred ← DIJKSTRA(V, E, ℓ, s)
                                                      ▷ Buscar ciclos que regresen al nodo s
5:     for each (v, s) ∈ E do
6:         if π[v] + ℓ(v, s) < min_ciclo_longitud then
7:             min_ciclo_longitud ← π[v] + ℓ(v, s)
8:             mejor_ciclo ← ReconstruirCiclo(pred, s, v)
9:         end if
10:    end for
11: end for
12: return mejor_ciclo, min_ciclo_longitud
```

ReconstruirCiclo

- `Entrada: Diccionario de predecesores pred, nodoinicio y nodo fin`
- `Salida: Una lista con el ciclo reconstruido`

```
1: function ReconstruirCiclo(pred, inicio, fin)
2:     ciclo ← [fin]
3:     nodo_actual ← fin
4:     while nodo_actual ≠ inicio do
5:         nodo_actual ← pred[nodo_actual]
6:         Añadir nodo_actual al inicio de ciclo
7:     end while
8:     Añadir inicio al final de ciclo (para completar el ciclo)
9:     return ciclo
10: end function
```

DIJKSTRA

- Entrada: V conjunto de nodos, E conjunto de aristas, ℓ longitudes de aristas, s nodo de origen
- Salida: Distancias π y predecesores pred desde s a cada nodo

```
1: function DIJKSTRA(V, E, ℓ, s)
2:     for each v ∈ V, v ≠ s do
3:         π[v] ← ∞
4:         pred[v] ← NULO
5:     end for
6:     π[s] ← 0
7:     Crear una cola de prioridad vacía pq
8:     for each v ∈ V do
9:         INSERT(pq, v, π[v])
10:    end for
11:    while IS-NOT-EMPTY(pq) do
12:        u ← DEL-MIN(pq)
13:        for each e = (u, v) ∈ E que sale de u do
14:            if π[v] > π[u] + ℓ[e] then
15:                DECREASE-KEY(pq, v, π[u] + ℓ[e])
16:                π[v] ← π[u] + ℓ[e]
17:                pred[v] ← u
18:            end if
19:        end for
20:    end while
21:    return π, pred
22: end function
```

Explicación del Algoritmo

- **EncontrarCicloMásCorto (función principal):**
  - Se itera sobre cada nodo s como punto inicial.
  - Se ejecuta Dijkstra desde s para calcular distancias mínimas π y predecesores pred.
  - Busca aristas que vayan desde otros nodos v de regreso a s (ciclos que se pueden formar de la forma (s, C, s)).
    Si la distancia $\pi[v] + \ell(v, s)$ es menor que la longitud del ciclo más corto encontrado hasta ahora, se actualiza el ciclo más corto.
- **DIJKSTRA:**
  - Implementa el cálculo de caminos más cortos desde el nodo de inicio s a todos los demás nodos en la gráfica.
  - Se utiliza una cola de prioridad para mantener y actualizar las distancias mínimas.

- Actualiza las distancias π y los predecesores pred para mantener el camino más corto.

- **ReconstruirCiclo:**

  - Utiliza los predecesores obtenidos para reconstruir el ciclo dirigido desde s que pasa por v y regresa a s.

Corrección del algoritmo

Se propone el siguiente invariante de ciclo para demostrar la corrección del algoritmo `EncontrarCicloMasCorto`.

Invariante de ciclo: Después de cada iteración del ciclo `for 5-10`, la variable `min_ciclo_longitud` tiene almaceda la longitud del ciclo menor encontrado hasta el momento, y la variable `mejor_ciclo` contiene el ciclo con esa longitud.

- **Inicialización:**
  Al comienzo del algoritmo, `min_ciclo_longitud` está inicializada en ∞ y `mejor_ciclo` en `NULL`. Esto es correcto porque antes de comenzar a explorar los ciclos, no se ha encontrado ningún ciclo válido, por lo que no se ha identificado ninguna longitud mínima de ciclo, cumpliéndose así el invariante.

- **Mantenimiento:**

  - Considere que la invariante se cumple al inicio de cada iteración del ciclo principal `for` (líneas 5-10).
  - Durante cada iteración, el algoritmo ejecuta Dijkstra desde un nodo s para determinar las distancias mínimas hacia los demás nodos.
  - A continuación, el algoritmo examina las aristas que se conectan de vuelta al nodo s. Si encuentra una arista (v, s) que, junto con la distancia $\pi[v]$, forma un ciclo de menor longitud que el valor actual de `min_ciclo_longitud`, entonces actualiza `min_ciclo_longitud` con esta nueva longitud y `mejor_ciclo` con el nuevo ciclo encontrado.
  - De este modo, se garantiza que, al final de cada iteración del ciclo principal `for`, `min_ciclo_longitud` refleje la longitud del ciclo más corto hallado hasta ese momento, cumpliéndose la invariante.

- **Terminación:**

  - Al finalizar el ciclo `for`, se han examinado todos los nodos posibles como puntos de partida para Dijkstra, y por ende, se han considerado todos los ciclos dirigidos en el grafo.
  - Dado que la invariante se mantuvo durante todas las iteraciones, al concluir el algoritmo, `min_ciclo_longitud` representa la longitud del ciclo dirigido más corto en la gráfica.
  - Además, `mejor_ciclo` contiene el ciclo más corto correspondiente.

Complejidad Temporal

- La implementación de Dijkstra usando colas de prioridad tiene por complejidad, $O(|E| \log |V|)$. Puesto que se está ejecutando para cada nodo $|V|$ entonces la complejidad es de la forma: $O(|V| \cdot |E| \cdot \log |V|)$.

- Revisar las aristas para formar ciclos tiene una complejidad de $O(|E|)$, al hacer esto por cada vértice, se tiene una contribución de $O(|V| \cdot |E|)$

Ahora, es cierto que

$$|V| \cdot |E| \leq |V| \cdot |E|$$

Pero como $\log |V| \geq 1$, entonces se satisface $|V| \cdot |E| \leq |V| \cdot |E| \cdot \log |V|$. Es por ello que:

$$O(|V| \cdot |E|) \subset O(|V| \cdot |E| \cdot \log |V|)$$

Es por ello que la complejidad temporal del algoritmo es del orden de $O(|V| \cdot |E| \cdot \log |V|)$.

Problema 4

Demuestra la corrección del algoritmo de árbol generador de peso mínimo de Prim, Kruskal o Borrado-Inverso (queda a tu elección), usando el concepto de regla roja y regla azul vistas en en clase.

Solución

El algoritmo de Prim es el siguiente:

Prim

```
1:  function PRIM(V, E, c)
2:      S ← ∅
3:      T ← ∅
4:      s ← cualquier nodo en V
5:
6:      for each v ≠ s do
7:          π[v] ← ∞, pred[v] ← null
8:      end for
9:
10:     π[s] ← 0
11:     Crear una cola de prioridad vacía pq
12:
13:     for each v en V do
14:         insert(pq, v, π[v])
15:     end for
16:
17:     while IS-NOT-EMPTY(pq) do
18:         u ← Del_min(pq)
19:         S ← S ∪ {u}, T ← T ∪ {pred[u]}
20:         for cada arista e = (u, v) en E con v ∉ S do
21:             if c_e < π[v] then
22:                 Decrease-Key(pq, v, c_e)
23:                 π[v] ← c_e; pred[v] ← e
24:             end if
25:         end for
26:     end while
27: end function
```

Demostración de corrección de la regla azul:

Se demuestra por inducción en el número de iteraciones.

Invariante: Existe un MST (árbol generador de peso mínimo) (V, T^*) cuyas aristas son todas azules, y no contiene aristas rojas.

- **Caso base:** En este caso no hay aristas coloreadas por lo que todos los MST cumplen con la

invariante.

- **Hipótesis de inducción:** La invariante de color se mantiene antes de aplicar la regla azul.
- **Paso inductivo:**
  - Sea D el conjunto de corte seleccionado en esta iteración del algoritmo, y sea f la arista que se elige para colorear de azul en este paso. Por la definición de la regla azul, f es la arista de menor peso que cruza el corte D.
  - Si f ya pertenece al árbol generador mínimo (MST) T^*, entonces T^* sigue cumpliendo la invariante, ya que no se ha introducido ninguna arista roja.
  - Supongamos que $f \notin T^*$, es decir, f no es una arista que inicialmente pertenece al MST T^*. Al agregar f a T^*, se forma un ciclo fundamental C en el grafo. Esto ocurre porque T^* es un árbol, y la adición de una arista extra crea exactamente un ciclo.
  - Dentro de este ciclo C, podemos identificar al menos una arista $e \neq f$ que también cruza el corte D. Debido a cómo se seleccionó f (la arista de menor peso que cruza D), podemos deducir que el peso de e (c_e) es mayor que el peso de f (c_f). Esto se debe a las siguientes razones:
    - $e \in T^*$**:** Como e ya está en el MST T^*, sabemos que e no ha sido coloreada de rojo. Las aristas del MST no se colorean de rojo, porque eso indicaría que hay una arista de menor peso que conecta las mismas componentes, contradiciendo la minimalidad del MST.
    - **Aplicación de la regla azul:** La regla azul asegura que f es la arista de menor peso que cruza D, lo que implica que $c_f < c_e$ para cualquier otra arista e que cruza el mismo corte y que no haya sido coloreada de azul.
  - Ahora, al agregar f al conjunto de aristas T^*, formamos un ciclo C. Podemos 'romper' el ciclo eliminando la arista e (donde $e \in C$ y $c_e > c_f$). Así, el nuevo conjunto de aristas $T' = T^* \cup \{f\} - \{e\}$ sigue siendo un árbol generador, pero con un peso menor o igual que T^*. De esta forma, hemos creado un nuevo árbol generador mínimo que incluye f y mantiene la propiedad de que todas sus aristas son azules.
  - La sustitución de e por f asegura que T' sigue cumpliendo con la invariante: T' es un árbol generador mínimo que no contiene aristas rojas y mantiene todas sus aristas azules. Por lo tanto, el algoritmo procede de forma correcta y preserva la invariante a lo largo de todas las iteraciones.

Problema 5

Considera la siguiente situación. Se tienen n *objetivos* posicionados en una línea horizontal L_o, cuyas coordenadas están marcadas con números enteros. La posición del objetivo i-ésimo se denota como o_i, $1 \leq i \leq n$, y no hay dos objetivos en la misma posición. De forma similar, se tienen n *cazadores* posicionados en otra linea horizontal L_c, paralela a L_o, cuyas coordenadas también están marcadas con número enteros, y alineadas con L_o. La posición del cazador j-ésimo se denota como c_j, $1 \leq j \leq n$, y puede haber cazadores en la misma posición. Ahora bien, cada cazador se puede mover a la derecha o izquierda, de forma tal que pueda apuntar a algún objetivo, es decir, quede en la misma posición. Los cazadores no se bloquean entre sí al encontrarse en un mismo punto moviéndose en direcciones opuestas. Mover un cazador de la posición c_j a la $c_j \pm x$ toma exactamente x unidades de tiempo.

Lo que buscamos en este problema es diseñar un algoritmo que indique a cada cazador que tantas unidades se debe mover a la derecha o izquierda de forma tal que (1) cada cazador apunte a un objetivo, (2) cada objetivo es apuntado por un cazador, y (3) el tiempo en que todos los objetivos son apuntados es el mínimo posible.

Diseña un algoritmo *greedy* de tiempo polinomial que solucione este problema, y explica porque consideras que tu algoritmo es *greedy*. Además, prueba corrección y complejidad de tu algoritmo.

Solución

L_c y L_o representan dos arreglos de dimensión n, donde la posición del objetivo i-esimo es denotada por o_i y la del cazador j-esimo por c_j.
El algoritmo propuesto es el siguiente:

Posiciones

```
1:  function POSICIONES(Lc, Lo)
2:      Lc_ordenado ← MergeSort(Lc)
3:      Lo_ordenado ← MergeSort(Lo)
4:
5:      for i = 1 up to n do
6:          Lc_final[i] ← 0
7:      end for
8:
9:      for i = 1 up to n do
10:         c_j ← Lc_ordenado[i]
11:         o_i ← Lo_ordenado[i]
12:         x ← c_j − o_i
13:         Lc_final[i] ← x
14:     end for
15:
16:     return Lc_final
17: end function
```

Explicación del algoritmo

- Ordenar las posiciones de los cazadores (L_c) y objetivos (L_o) de menor a mayor.

- Asignar a cada cazador ordenado al objetivo correspondiente en la misma posición de la lista ordenada.
- Calcular la cantidad de movimiento necesario para cada cazador, restando la distancia que debe recorrer para alcanzar su objetivo.
- Devuelve una lista de movimientos, donde:
  - Los valores positivos indican que un cazador debe moverse a la derecha.
  - Los valores negativos indican movimiento a la izquierda.
  - Cero significa que no debe moverse.

Corrección del algoritmo

Para demostrar que el algoritmo da una solución óptima, se necesita ver por qué cualquier otro emparejamiento daría como resultado un costo mayor.

- **Lema de Intercambio:**
  - Suponga que existe una solución óptima diferente a la que genera el algoritmo propuesto, pero que esta solución tiene un costo menor. Entiéndase que hay un mejor emparejamiento entre cazador y objetivo que el que hace el algoritmo propuesto.
- Sin embargo, cualquier intercambio de cazadores en la solución óptima que no siga el orden dado por el algoritmo propuesto siempre resultará en un aumento del costo total.
  - La estrategia de emparejar el cazador más pequeño con el objetivo más pequeño (después de ordenar) busca minimizar las distancias $x = c_i - o_i$ porque cualquier otro emparejamiento aumentaría la suma de las distancias.
    Considere lo siguiente. Se tiene el arreglo:

    $$Lc\_ordenado = [c_1, c_2, c_3, ..., c_i, ..., c'_i, ..., c_n]$$

    Tómese un elemento de éste, llámese c'_i, tal que $c_i < c'_i$, quedando como nuevo arreglo de cazadores.

    $$Lc\_nuevo = [c_1, c_2, c_3, ..., c'_i, ..., c_i, , ..., c_n]$$

    El algoritmo calcula la cantidad $x = c_i - o_i$ para la posición ordenada, que minimiza la distancia de movimiento para el cazador c_i al objetivo o_i. Ahora, si se intercambia c_i y $c_{i'}$, el cálculo del costo para esa posición cambiaría a $x' = c'_i - o_i$. Dado que $c_i < c'_i$ y o_i es una valor fijo, la cantidad $|c'_i - o_i|$ será mayor que $|c_i - o_i|$. Esto significa que cualquier intercambio de cazadores (otra configuración) aumentará la distancia total, no la disminuirá.

El intercambio de posiciones en los cazadores lleva a un incremento en la distancia total, lo que contradice la suposición de que existe una solución más óptima que la del algoritmo propuesto. Por lo tanto, se concluye que la solución generada por el algoritmo propuesto es la óptima.

Complejidad temporal

- Se utiliza el algoritmo `MergeSort` para ordenar los elementos del arreglo, por lo que la complejidad, hasta este punto, es del orden de $O(n \log n)$.
- Crear el arreglo `Lc_final` (líneas 5-7) toma tiempo a lo más lineal, por lo que la contribución de esto es del orden $O(n)$.
- Hacer la resta de los elementos de los arreglos toma tiempo lineal (líneas 9-14), por lo que se tiene una contribución de $O(n)$.

Es así que la complejidad total del algoritmo es de orden: $O(n \log n)$.