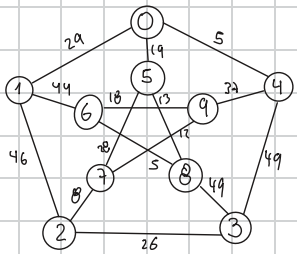
Taller 4 Dalgo

Juan Felipe Ochoa
202320053

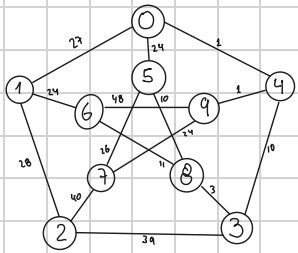
Punto 1

Redes muy nice:

Grafo 1



Grafo 2

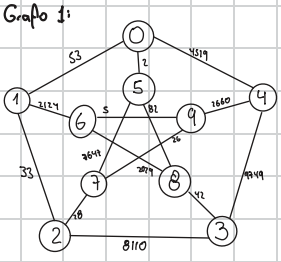



```
---------
Grafo 1:
  Ford-Fulkerson flujo = 38  ops=143
  Edmonds-Karp  flujo = 38  ops=85

Grafo 2:
  Ford-Fulkerson flujo = 25  ops=143
  Edmonds-Karp  flujo = 25  ops=85
```

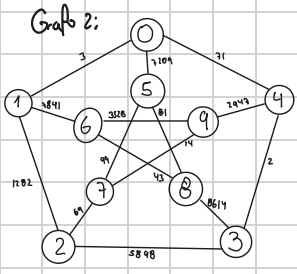
→ Obtuvieron el mismo flujo máximo (39 y 25 respectivamente) con ambos algoritmos. Se diferencian con el número de pasos (OPS). Edmond-karps fue más eficiente en ambos casos con casi 40% menos operaciones.

Redes nice pero peligrosas:

Grafo 1:



Grafo 2:




```
Red 1:
  Ford-Fulkerson flujo=55 ops=70
  Edmonds-Karp  flujo=55 ops=132

Red 2:
  Ford-Fulkerson flujo=16 ops=109
  Edmonds-Karp  flujo=16 ops=85
```

Ambos usan estructuras de tamaño
$O(V+E)$ -> (matriz de capacidades, listas de adyacencia y arrays auxiliares).

FF: visited[], pila implícita de la recursión → $O(V+E)$

EK: parent[], cola de la BFS → $O(V+E)$
Mismo orden; distinta constante (EK usa más estructura auxiliar).

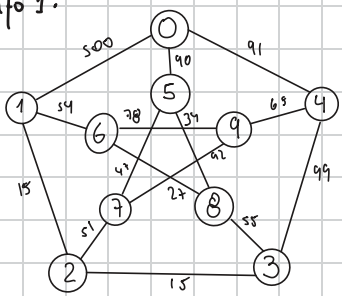
Se cumplieron las condiciones de “nice pero peligrosas” en ambas redes (verificación por conteo).
Flujo máximo: Red 1 = 55, Red 2 = 16 (idéntico con FF y EK).

Costos temporales: en redes con cuellos claros cerca del source y grandes capacidades internas, FF puede resultar más barato; cuando la red exige muchas búsquedas controladas, EK tiende a ser más eficiente.

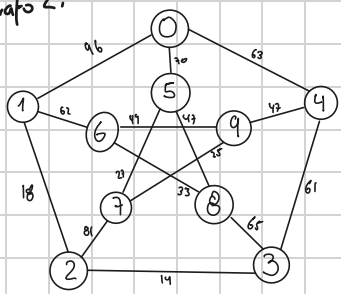
La correctitud quedó validada: ambos métodos coinciden en el valor del flujo máximo.

Redes NO NICE I

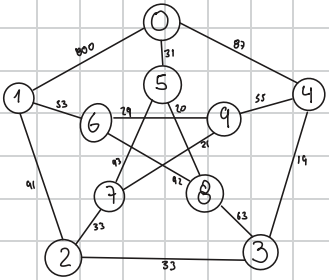
Grafo 1:



Grafo 2:



Grafo 3:



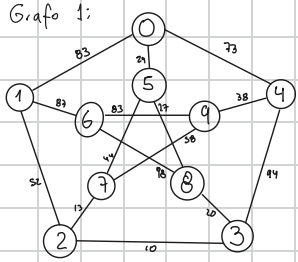

```
Red A (??):
  Ford-Fulkerson flujo=107
  Edmonds-Karp  flujo=107

Red B (??):
  Ford-Fulkerson flujo=121
  Edmonds-Karp  flujo=121

Red C (ambas):
  Ford-Fulkerson flujo=107
  Edmonds-Karp  flujo=107
```

El algoritmo Ford–Fulkerson (DFS) presenta una complejidad temporal de $O(E \cdot f)$, donde E es el número de aristas y f el valor del flujo máximo. Su rendimiento puede variar según los caminos aumentantes encontrados. En cambio, Edmonds–Karp (BFS) tiene una complejidad temporal más estable de $O(V$*E^2), ya que realiza una búsqueda por niveles en cada iteración. En ambos casos, la complejidad espacial es Θ(V^2), dominada por la matriz de capacidades y las estructuras auxiliares utilizadas para representar la red.

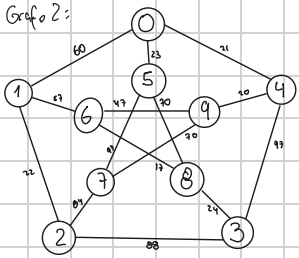
Ambos algoritmos entregaron el mismo flujo máximo, confirmando su corrección. Sin embargo, Ford–Fulkerson puede ser más rápido cuando los caminos aumentantes son pocos y de alta capacidad, mientras que Edmonds–Karp resulta más predecible y eficiente en redes con muchos cuellos o rutas equivalentes. Los resultados también demuestran que tener aristas con capacidades muy grandes no garantiza un mayor flujo total, ya que este depende del corte mínimo de la red. En conclusión, Ford–Fulkerson muestra una mayor variabilidad en tiempo, y Edmonds–Karp ofrece un desempeño más consistente.

NO NICE II

Grafo 1:



Grafo 2:



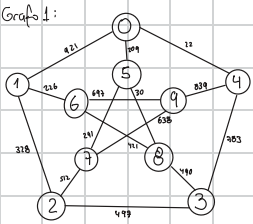
```
Red 1 (no-nice II):
  Ford-Fulkerson flujo=67  ops=166
  Edmonds-Karp  flujo=67  ops=197

Red 2 (no-nice II):
  Ford-Fulkerson flujo=62  ops=252
  Edmonds-Karp  flujo=62  ops=134
```

El algoritmo Ford–Fulkerson (DFS) mantiene una complejidad temporal de $O(E \cdot f)$, ya que el número de iteraciones depende directamente de la cantidad de caminos aumentantes y de la magnitud del flujo máximo. Por su parte, Edmonds–Karp (BFS) conserva su complejidad temporal de $O(V^*E^2)$, más estable y con un límite superior definido, al realizar búsquedas por niveles en cada iteración. En ambos casos, la complejidad espacial es $\Theta(V^2)$, dominada por la matriz de capacidades y las estructuras auxiliares necesarias para el almacenamiento de las aristas, especialmente relevantes en redes con aristas antiparalelas.

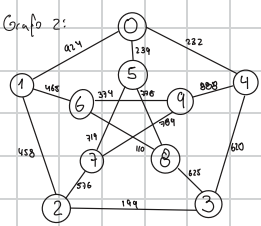
En las dos redes no-nice II, ambas con aristas antiparalelas y capacidades entre 10 y 100, los algoritmos produjeron el mismo flujo máximo, validando su corrección. Sin embargo, las operaciones (τ) muestran diferencias: en la primera red Ford–Fulkerson fue más eficiente (166 vs. 197), mientras que en la segunda Edmonds–Karp necesitó menos operaciones (134 vs. 252). Esto confirma que, en redes con direcciones opuestas entre nodos, el comportamiento de Ford–Fulkerson puede volverse más variable dependiendo del orden de los caminos encontrados, mientras Edmonds–Karp mantiene un rendimiento más regular al usar BFS. En general, las aristas antiparalelas aumentan la complejidad estructural del grafo, pero no afectan el valor final del flujo, solo el esfuerzo computacional necesario para alcanzarlo.

No Nice III

Grafo 1:



Grafo 2:



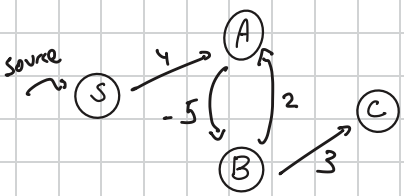
```
Red 1 (no-nice III):
  Ford-Fulkerson flujo=1263  ops=381
  Edmonds-Karp  flujo=1263  ops=262

Red 2 (no-nice III):
  Ford-Fulkerson flujo=1098  ops=314
  Edmonds-Karp  flujo=1098  ops=296
```

Con tu representación con matriz de capacidades, ambos algoritmos usan espacio $\Theta(V^2)$. En tiempo, basicFordFulkerson (DFS) es $O(E \cdot f)$: el número de aumentos depende del valor del flujo máximo y puede crecer cuando las capacidades son grandes (como aquí, 1–1000). Edmonds–Karp (BFS) mantiene un límite superior más estable de $O(VE^2)$ porque cada aumento se encuentra con una BFS por niveles y el número total de aumentos es $O(V^*E)$; su complejidad no depende de la magnitud de las capacidades.

En ambas redes (con $\delta^+(\sigma)>0$, $\delta^-(\tau)>0$ y tres pares antiparalelos), FF y EK devolvieron el mismo flujo máximo (Red 1: 1263, Red 2: 1098), confirmando la correctitud. En eficiencia, EK fue sistemáticamente mejor: Red 1 τEK=262<τFF=381 y Red 2 296<314. La combinación de capacidades muy altas y antiparalelas genera muchas opciones de caminos y flujo de retorno en la residual; EK evita zig-zags profundos al escoger siempre caminos más cortos en número de aristas, mientras que FF puede gastar aumentos “largos” y por eso su τ crece con f. Como en todos los casos, el valor final lo determina el min-cut: las aristas gigantes solo ayudan si no hay cuellos posteriores en los cortes hacia el sink.

Punto 2

Grafo 4 nodos:



Bellman-Ford: 4 nodos, 3 iteraciones

① S: 0, A: ∞, B: ∞, C: ∞ → S: 0, A: 4, B: -1, C: 2

② S: 0, A: 4, B: -1, C: 2 → S: 0, A: -2, B: -4, C: -1

③ S: 0, A: -2, B: -4, C: -1 → S: 0, A: -5, B: -7, C: -4

↳ Vemos que en A→B aún relaja: -5-6 = -10 < -7, hay todavía un ciclo negativo alcanzable.

↳ Como el ciclo A↔B es alcanzable desde S y desde S alcanzamos A, B y C, no existe distancia mínima a esos nodos.

↳ Floyd-Warshall:

① k=0

(Cada α filas = ∞)	S	A	B	C
S	0	4		
A		0	-5	
B		2	0	3
C				0

k=1 →

	S	A	B	C
S	0	4	-1	
A		0	-5	
B		2	0	3
C				0

k=2 →

	S	A	B	C
S	0	1	-1	2
A		0	-5	-2
B		2	0	3
C				0

k=3 y final

	S	A	B	C
S	0	1	-4	-1
A		-3	-8	-5
B		-1	-6	-3
C				0

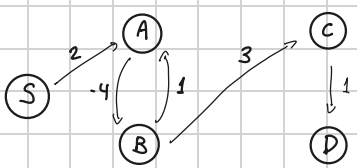
Deja resultados negativos ya que no puede dar ciclos y "vuelta" infinitas.

Resultado desde nodo 0:

BF: [0, -∞, -∞, -∞]

FW: [0, 1, -4, -1]

Grafo 5 nodos:



Bellman-Ford: 5 nodos, 4 iteraciones

① S: 0, A: ∞, B: ∞, C: ∞, D: ∞ → S: 0, A: -1, B: -2, C: 1, D: 2

② S: 0, A: -1, B: -2, C: 1, D: 2 → S: 0, A: -4, B: -5, C: -2, D: -1

③ S: 0, A: -4, B: -5, C: -2, D: -1 → S: 0, A: -7, B: -8, C: -5, D: -4

④ S: 0, A: -7, B: -8, C: -5, D: -4 → S: 0, A: -11, B: -12, C: -9, D: -8

↳ Hay ciclo negativo alcanzable desde S: No existe distancia mínima a los nodos A, B, C y D.

↳ Floyd-Warshall

① k=0

(Cada α filas = ∞)	S	A	B	C	D
S	0	2			
A		0	-4		
B		1	0	3	
C				0	1
D					0

② k=1

	S	A	B	C	D
S	0	2	-2		
A		0	-4		
B		1	-3	3	
C				0	1
D					0

③ k=2

	S	A	B	C	D
S	0	-1	-5	-2	
A		-3	-7	-4	
B		-2	-6	-3	
C				0	1
D					0

④ k=3

	S	A	B	C	D
S	0	-1	-5	-2	-1
A		-3	-7	-4	-3
B		-2	-6	-3	-2
C				0	1
D					0

⑤ k=4

	S	A	B	C	D
S	0	-1	-5	-2	-1
A		-3	-7	-4	-3
B		-2	-6	-3	-2
C				0	1
D					0

Deja ese resultados al no poder dar ciclos y "vueltas" infinitas.

Resultado desde nodo 0:

BF: [0, -∞, -∞, -∞, -∞]

FW: [0, -1, -5, -2, -1]

Conclusión: Bellman-Ford marca -∞ para cualquier nodo alcanzable desde un ciclo negativo alcanzable (no existe mínimo). Floyd-Warshall no da vueltas infinitas al ciclo, aunque su diagonal evidencia ciclo negativo (D[k][k]<0), la fila del source queda con números finitos si no se aplica un post-procesamiento.

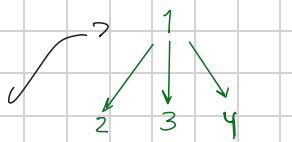
La fila del source en Floyd-Warshall no coincide con el vector de distancias de Bellman-Ford. Por tanto, Bellman-Ford NO es una "restricción por filas" de Floyd-Warshall.

Punto 3

3.2) Durante la ejecución del algoritmo, al aplicar la técnica de path compression en la operación smartFind, el arreglo height deja de representar la altura real de los árboles que conforman los conjuntos distintos. Esto ocurre porque smartFind aplana los árboles haciendo que los nodos apunten directamente a la raíz, pero sin actualizar el valor almacenado en height. Sin embargo, esto no afecta la eficiencia ni la corrección de la operación union, porque height solo se utiliza como una heurística para decidir qué árbol colocar debajo de cuál (ya que no es necesario que represente la altura exacta).

Diagrama Representativo:

```
//Antes del path Compression
//height[raíz] = 3   (altura real = 3
1 → 2 → 3 → 4

//Después del smartFind (usando Path Compression)
//height[raíz] sigue siendo 3, pero el árbol se aplana
```



//altura Real = 1

(Aunque height quedó desactualizado (ya no vale 3, sino 1), eso no afecta la eficiencia de union, porque la estructura ya se aplanó).

Debido a la combinación de unión by height y path compression, los árboles se mantienen muy planos, garantizando una complejidad amortizada casi constante, expresada como:

$$T_{union/find}(n) \approx O(\alpha(n))$$

donde $\alpha(n)$ es la función inversa, que para cualquier entrada práctica vale menor de 5. En otras palabras, aunque los valores de height ya no reflejen las alturas reales, la estructura continúa funcionando eficientemente y el tiempo de ejecución de union no se ve afectado.

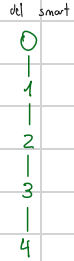
3.3) Si se implementa Kruskal usando badUnion (sin comparar alturas) junto con smartFind (path compression), los árboles que representan los conjuntos pueden volverse desbalanceados temporalmente, ya que las uniones no tienen en cuenta la altura de cada raíz.

Sin embargo, el path compression de smartFind aplana los árboles cada vez que se realiza una búsqueda, haciendo que las siguientes operaciones sean mucho más rápidas. Por tanto, aunque el peor caso aislado para una operación de find o union puede ser $O(n)$, el tiempo amortizado total del algoritmo sigue siendo $O(\alpha(n))$, es decir, prácticamente constante.

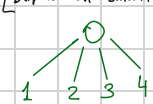
En consecuencia, el tiempo de ejecución total de Kruskal con badUnion y smartFind continúa dominado por el sort de las aristas, es decir: $T_{Kruskal} = O(E \log E)$ E siendo las aristas

Visualmente:

[Antes del smartFind]



[Después del smartFind]



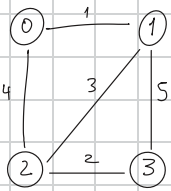
// En conclusión //

El smartFind compensa la ineficiencia de badUnion, logrando que el algoritmo conserve una complejidad amortizada casi constante en las operaciones de find y Union.

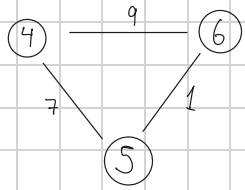
Punto 4

4.1) Grafo utilizado para la comparación entre kruscal y JPrim: (no conexo)

Componente A:



Componente B:



MST(A) = 6
MST(B) = 8

MSF: peso total = 14

Comparación Kruskal vs. JPrim:

→ Kruskal: funciona naturalmente en grafos no conexos porque ordena todas las aristas y va uniendo componentes mientras no forme ciclos. Devuelve un bosque generador mínimo (MSF). Complejidad: $O(E \log E)$ → domina el ordenamiento

→ J-Prim: Si lo aplicar "a lo bruto" desde un solo nodo fallaría en no conexos (se quedaría sin aristas en la PQ). Con el bucle implementado produce también un MSF. Complejidad $O(E \log V)$ → Domina la cola de prioridad, para grafos densos es similar a Kruskal.

Conclusión: Para grafos no conexos ambos devuelven un MSF, si JPrim se aplica por cada componente. La diferencia práctica es de implementación:
- Kruskal no necesita saber cuántas componentes hay
- Prim necesita un bucle externo (o "reinicio") para cada componente.

Foto resultado del código:

```
PS C:\Users\jfoch\DALGO\tarea4_dalgo> & 'C:\Program Files\Java\jdk
ser\workspaceStorage\acb597db8b82e27e80f186800ab14373\redhat.java\
Kruskal: peso=14  T=[(0-1:1), (5-6:1), (2-3:2), (1-2:3), (4-5:7)]
JPrim  : peso=14  T=[(0-1:1), (1-2:3), (2-3:2), (4-5:7), (5-6:1)]
PS C:\Users\jfoch\DALGO\tarea4_dalgo>
```

4.2) En el algoritmo JPrim original se inicializa con $B = \{0\}$ y $T = 0$, es decir, se comienza el árbol desde un nodo arbitrario (por defecto el nodo 0).

En la versión modificada de JPrim 1, se cambia inicialización por $B: 0$ y $T: \{e\}$, donde e es la arista más liviana del grafo. En este caso, el árbol comienza desde los dos nodos extremos de la arista más corta en lugar de un solo nodo.

Al ejecutar, ambos algoritmos arrojan el mismo resultado: las aristas del árbol y el peso total coinciden.

Foto resultado del código:

```
jPrim  : peso=14  T=[(0-1:1), (1-2:3), (2-3:2), (4-5:7), (5-6:1)]
jPrim1 : peso=14  T=[(0-1:1), (1-2:3), (2-3:2), (4-5:7), (5-6:1)]
PS C:\Users\jfoch\DALGO\tarea4_dalgo>
```

Esto sucede porque ambos algoritmos siguen el mismo principio avaro: en cada iteración eligen la arista más liviana que conecta un nodo dentro del árbol con otro fuera de él.

La única diferencia es el punto de inicio: JPrim empieza desde un nodo, JPrim 1 empieza desde una arista. Pero en ambos casos, el proceso de expansión y selección es idéntico, por lo que no cambia el resultado final.

Conclusión:

→ Los resultados idénticos en el código son correctos y esperados. La modificación solo altera el orden de crecimiento del árbol, no el conjunto final de aristas ni su peso total.

Diagrama JPrim:

$B = \{0\}$, $T = 0$

aristas mínimas } crece desde un nodo

Diagrama JPrim1:

$B: 0$, $T = \{e\}$ ← arista más corta (u, v)

u — v

se expanden al resto del grafo } Crece desde ambos extremos