UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE INGENIERÍA
Año 2022- 1er Cuatrimestre

[75.29/95.06] TEORÍA DE ALGORITMOS

Trabajo Práctico N°3: “Redes de flujo y Problemas NP Completos”

Padrón	*Apellido y Nombre*	*Email*
93310	Cabrera, Jorge	jocabrera@fi.uba.ar
93042	Anez, Johana	janez@fi.uba.ar
92354	Serra, Diego	dserra@fi.uba.ar
107587	Gallino, Pedro	pgallino@fi.uba.ar
104126	Lamanna, Tobias	tlamanna@fi.uba.ar

Fecha 1er entrega: 18/05/2022

Código fuente:

Grupo N° 2

Parte 1

1. Explicar Min Cost Max Flow

El problema del camino mínimo de mayor flujo es un problema particular del problema general conocido como “Minimum cost flow”. Definimos:

- G un grafo con n nodos y m aristas.
- c costo
- s un nodo fuente de G
- t un nodo sumidero de G.
- r un valor de flujo

Queremos encontrar un flujo de s a t con valor r, con c costo mínimo. En particular, para el caso del mayor flujo, es el que nos interesa analizar.

Este problema no siempre tiene solución, ya que en el caso en que la sumatoria de los flujos que ingresan a los nodos y los que egresan sea distinta de cero, no habrá solución posible. Para que sea posible encontrar una solución, se deben cumplir 4 condiciones en el grafo:

- Todos los valores de capacidad, costo y flujo son enteros
- El grafo es dirigido
- Todos los costos son mayores o iguales a 0
- La diferencia entre el flujo que ingresa y egresa de todos los nodos de la red, es cero. De esta regla están excluidos los nodos fuente y sumidero.

Dadas estas condiciones, es posible aplicar varios algoritmos para resolver el problema, basándose en el grafo residual.

En el caso de *Cycle Cancelling Algorithm* se parte de un grafo con ciclos negativos y va aumentando el flujo en dichos ciclos. En cada iteración se mantienen flujos factibles que van mejorando progresivamente el flujo. Una vez se cancelan todos los ciclos con costo negativo, se encuentra el flujo máximo. Un problema de este algoritmo es que no indica un criterio a la hora de seleccionar los ciclos negativos del grafo, con lo cual, la complejidad temporal puede variar significativamente en función de cómo decidamos elegir esos ciclos negativos, hay formas de elegir correctamente los ciclos negativos, logrando que la complejidad del mismo tienda a la polinomial.

2. Explicar cómo funciona el algoritmo seleccionado. Incluir: pseudocódigo, análisis de complejidad espacial, temporal y optimalidad.

Encontrar entre los máximos flujos el de menor costo: minimum-cost maximum-flow problem. Este problema puede resolverse mediante el algoritmo llamado *Successive Shortest Paths*. Es muy similar al algoritmo "Edmonds-Karp".

Se tiene un grafo orientado "G", que tiene como mucho una arista entre cada par de vértices. Una fuente "s" y un sumidero "t". Cada arista tiene una capacidad "u", un costo por unidad de flujo "c", y un flujo "f".

Se define un grafo residual "Gr", tal como se hace en Ford-Fulkerson: Las aristas hacia "adelante" tendrán una capacidad residual "r" = u - f. Las aristas hacia atrás tendrán una capacidad f. Al comenzar con un flujo total 0, el grafo residual "Gr" será igual al grafo G con aristas hacia atrás con capacidad 0 y costos negados.

Los costos negados en las aristas que regresan corresponden a restar el costo de los pasajeros que antes viajaban y ahora ya no.

En cada iteración, se encuentra el camino más corto en el grafo residual de s a t. A diferencia del *Edmonds-Karp* aquí se busca el más corto en cuanto a precio, no a cantidad de aristas. Si no quedan más caminos de s hasta t, el algoritmo termina y se retorna el flujo máximo de menor costo. Si se encuentra un camino, se incrementa el flujo a través del mismo lo máximo que se pueda.

Como se desea utilizar Dijkstra para obtener los caminos menos costosos desde la fuente hasta el sumidero, los ciclos negativos representan un problema. Para evitarlos, se toma el costo más alto, y en el grafo residual se le suma ese valor a todas las aristas. De esta manera, el costo mínimo será de 0 y podrá aplicarse Dijkstra.

Luego, a la hora de calcular el costo de un determinado flujo, al multiplicar el precio por unidad, previamente se volverá a restar el monto añadido.

Pseudocódigo

```
augment(f , P)

    Sea b = bottleneck(P, f )
```

```
        Para cada eje e=(u, v) ∈ P
            Si e = (u, v) eje hacia adelante
                f(e)+=b en G
            Sino (si es eje para atrás)
                e' = (v, u)
                f(e')-=b en G
Retornar f
Successive shortest paths
    validar que todas las capacidades sean mayores a cero
        Inicialmente f(e) = 0 Para todo e en G
        Mientras haya un camino s-t en Gf
            se obtiene por Dijkstra el camino menos costoso P
            f' = augment(f , P)  (paso flujo por el camino menos costoso)
            f = f'
          Gf = Gf'
Retornar f
```

Tal como en "Edmons-Karp" hay un total de E*V iteraciones. En cada iteración, se calcula el camino mínimo de s a t. Para calcular estos caminos, se utiliza Dijkstra de complejidad O(E*logV). Por lo tanto, la complejidad total es O((V*E).(E * logV)). La cual es polinomial.

3. Dar un ejemplo paso a paso de su funcionamiento.

Ejemplo:

Buenos Aires -> Nueva York 80, $5

Buenos Aires -> Rio de Janeiro 40, $2

Nueva York -> Río de Janeiro 25, $1

Rio de Janeiro -> Madrid 30, $2

Nueva York -> Madrid 40, $6

Rio de Janeiro -> Qatar 20, $10

Madrid -> Qatar 90, $5

París -> Qatar 10, $1

Nueva York -> París 35, $3

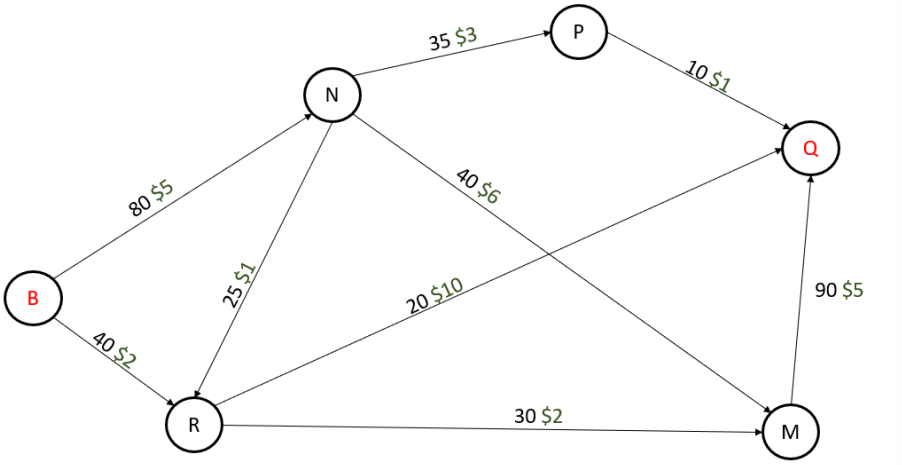

Figura 1: Grafo.

Se crea un grafo con B como fuente y Q como sumidero.

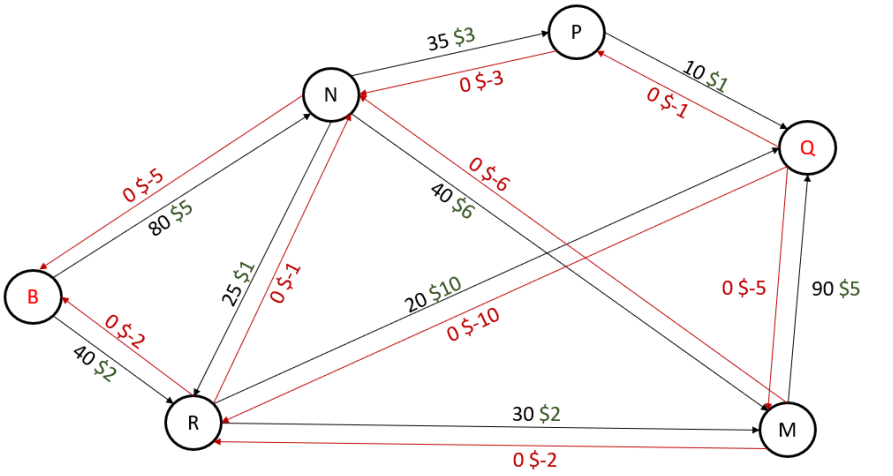

Figura 2: Grafo residual.

Se crea un grafo residual donde las aristas hacia adelante tienen capacidad u-f. Las aristas hacia atrás tienen capacidad f y costo negado.

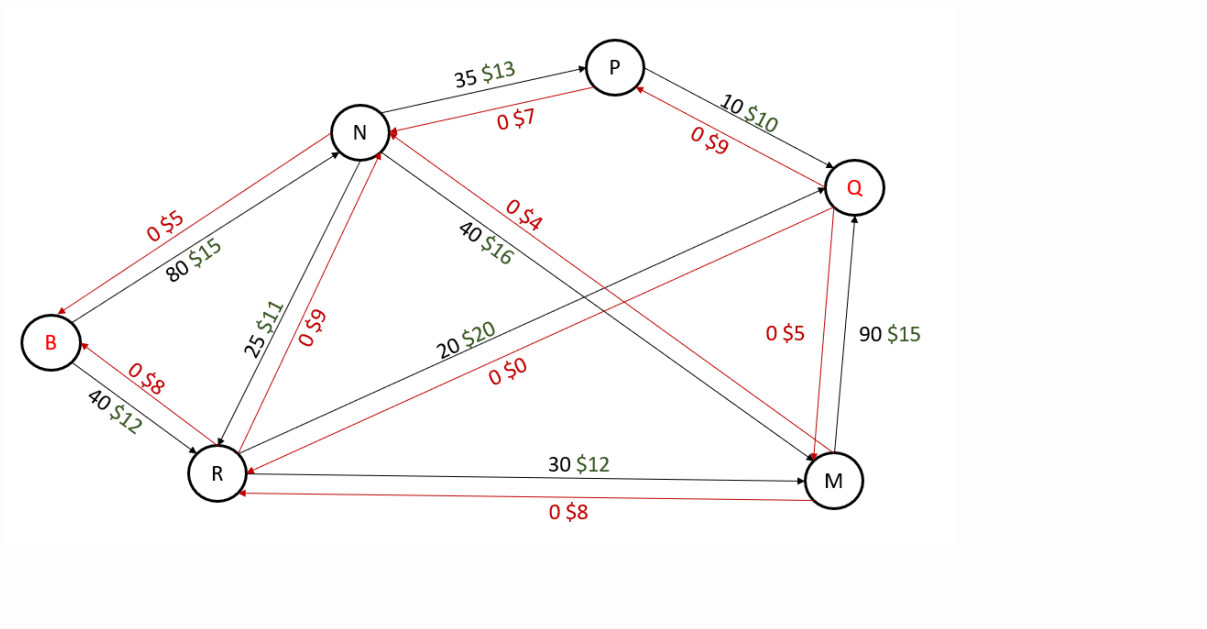

Figura 3: Grafo residual convertido a aristas positivas.

Como no es posible aplicar Dijkstra para grafos con ciclos negativos, se modifica el mismo para que todas las aristas sean positivas. Al cargar la información del problema, se guarda el costo más alto y se le suma a todas las aristas del residual.

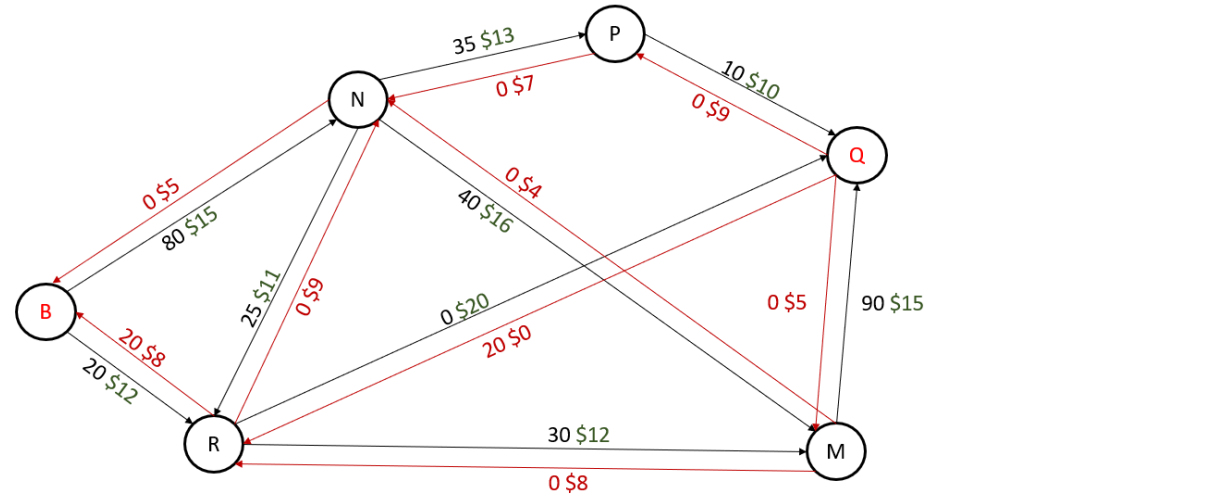

Figura 4: Camino de aumento B-R-Q

Mediante Dijkstra se obtiene el camino de menor costo de B a Q. Se obtiene el B-R-Q de costo $32. El bottleneck es 20. A las aristas correspondientes se les suma 20 o resta 20. Se tomó el camino más barato y se envió el mayor flujo posible.

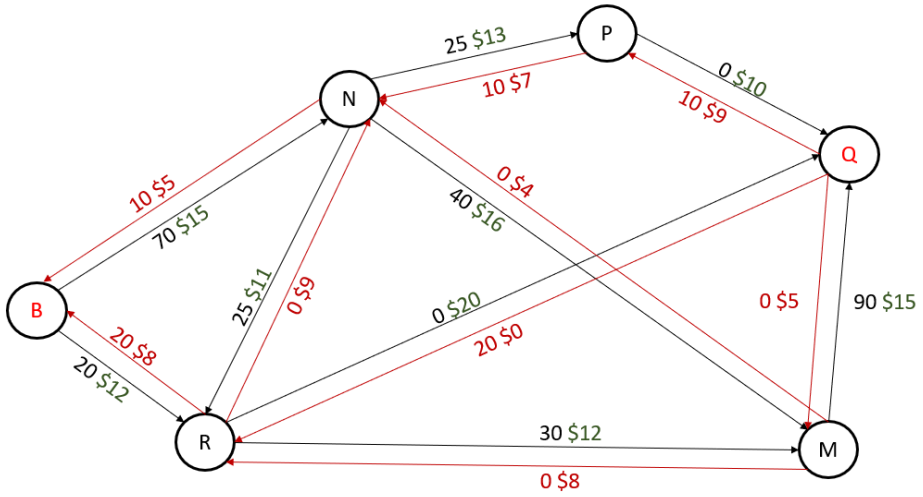

Figura 5: Camino de aumento B-N-P-Q.

Mediante Dijkstra se obtiene el camino de menor costo de B a Q. Se obtiene el B-N-P-Q de costo $38. El bottleneck es 10. A las aristas correspondientes se les suma 10 o resta 10. Se tomó el camino más barato y se envió el mayor flujo posible.

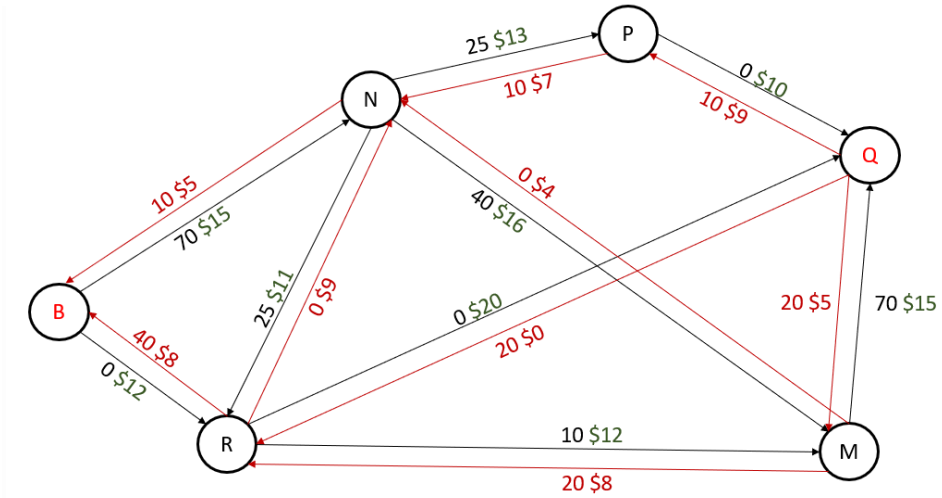

Figura 6: Camino de aumento B-R-M-Q.

Mediante Dijkstra se obtiene el camino de menor costo de B a Q. Se obtiene el B-R-M-Q de costo $39. El bottleneck es 20. A las aristas correspondientes se les suma 20 o resta 20. Se tomó el camino más barato y se envió el mayor flujo posible.

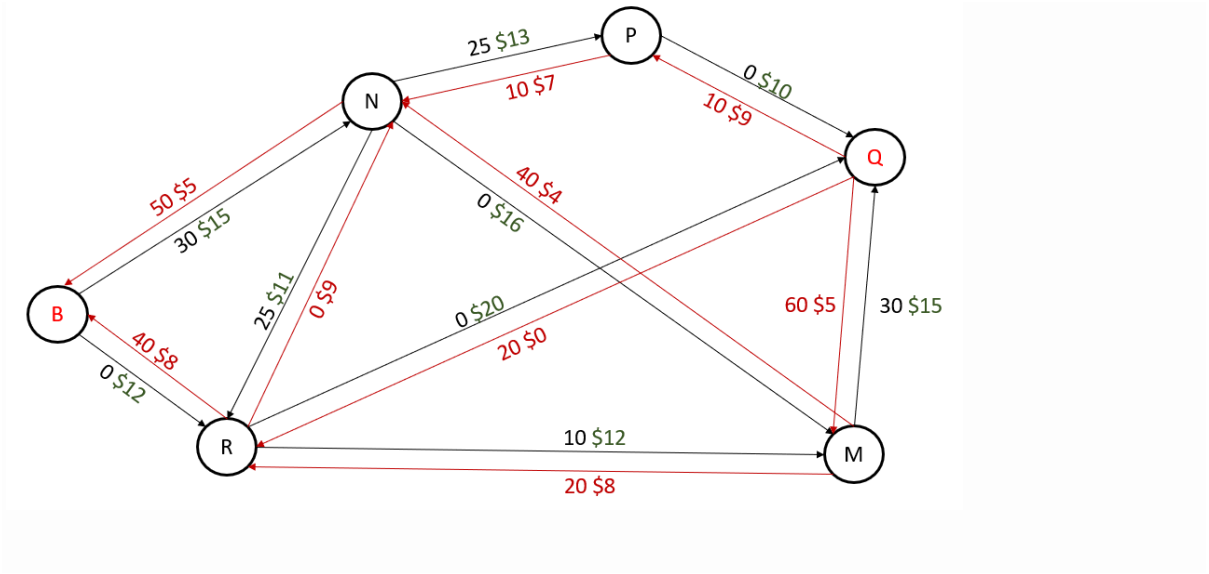

Figura 7: Camino de aumento B-N-M-Q.

Mediante Dijkstra se obtiene el camino de menor costo de B a Q. Se obtiene el B-N-M-Q de costo $46. El bottleneck es 40. A las aristas correspondientes se les suma 40 o resta 40. Se tomó el camino más barato y se envió el mayor flujo posible.

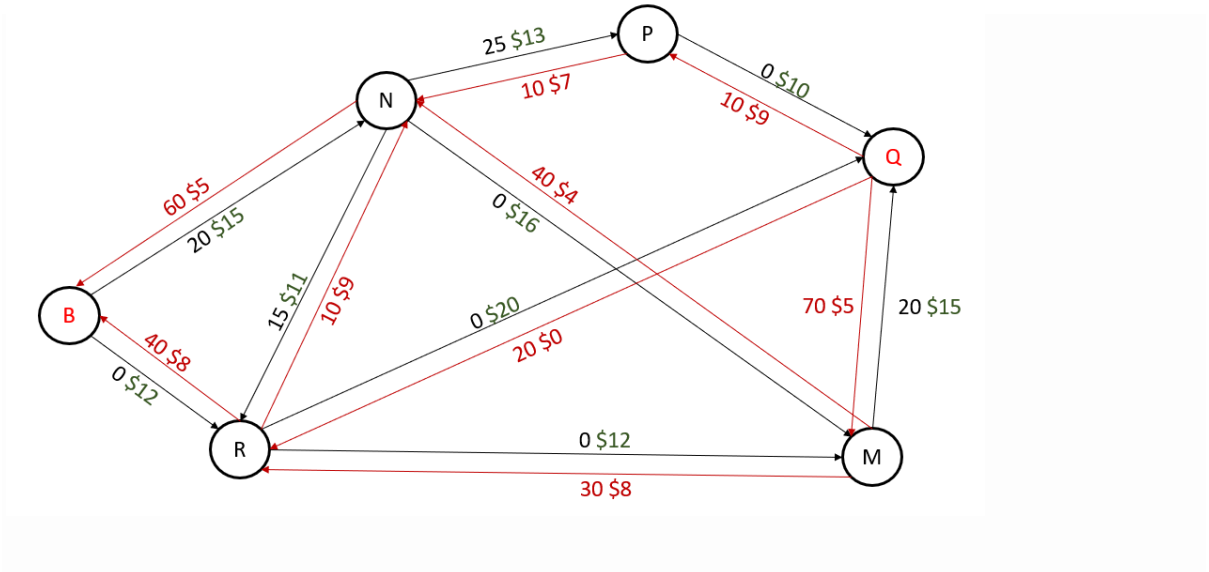

Figura 8: Camino de aumento B-N-R-M-Q.

Mediante Dijkstra se obtiene el camino de menor costo de B a Q. Se obtiene el B-N-R-M-Q de costo $53. El bottleneck es 10. A las aristas correspondientes se les suma 10 o resta 10. Se tomó el camino más barato y se envió el mayor flujo posible.

Aquí ya no hay caminos de B a Q posibles, por lo que el flujo final máximo de menor costo es de 100 personas por las aristas seleccionadas.

PARTE 2

1. Casting es NP-C

Para probar que Casting pertenece a NP-C se debe probar que pertenece a NP (es certificable en tiempo polinomial) y que pertenece a NP Hard.

Certificación polinomial de Casting (1er entrega):

Dada una lista de características K, una colección de listas NK (cada lista posee una características y las personas que la poseen) y un subconjunto de personas P se desea saber si P es solución del Casting con K características.

Para esto se puede iterar P y por cada persona Pi iterar NK viendo si Pi pertenece a NKj, si Pi se encuentra en NKj se puede remover Kj de K, en caso de intentar remover un Kj no presente en K la solución no es válida (ya que habría dos personas con la misma característica). Al terminar de iterar P, K debería estar vacío indicando que todas las características se encuentran representadas, en dicho caso la solución P es válida; si aún hay elementos en K la solución es inválida.
Como la verificación requiere dos iteraciones anidadas, una en P y otra en NK (siendo las operaciones a realizar dentro de la iteración O(n) en el peor caso) la verificación es polinómica, y por lo tanto, Casting pertenece a NP.

Reducción polinómica de Exact Cover en Casting

Para probar que Casting pertenece a NP Hard se puede tomar un problema perteneciente a NP Hard y reducirlo polinomicamente a Casting, esto probará que Casting es al menos tan difícil como un problema NP Hard.
En este caso se reducirá Exact Cover perteneciente a NP-C en Casting.

Descripción Exact Cover:
Dada una colección S de subconjuntos de un set X : {Xi, i = 1, 2, …, t} se desea determinar si existe una subcolección S* de S que verifique que cada Xi está contenido en exactamente un subconjunto de S*.

Entonces para reducir Exact Cover en Casting se necesitará transformar en tiempo polinómico la colección S y el set X en una lista de caracteristicas K, una lista de personas P y una colección de listas KP (cada lista corresponde a un Ki y posee elementos de P).

Para armar la lista K se tomarán los elementos de X, cada elemento de X pasará a ser un elemento de K, esto se puede hacer en tiempo lineal.

Luego se armará el conjunto P a partir de los elementos de S: por cada elemento Si de S se agregara el índice i a P, en otras palabras, P será el conjunto de índices que representan los subconjuntos de S. También se puede construir en tiempo lineal.
Y por último se creará una colección de listas LX y por cada elemento Xj en X se creará una lista LXj. Por cada lista LXj se recorrerá S, y si Xj se encuentra en Si se agregará el índice i a la lista LXj. Finalmente el conjunto de listas LX será el KP de Casting. El armado de LX se puede realizar en tiempo polinomial dado que por cada elemento en X se crea una lista y por cada lista se recorre S.

Casting resolverá el problema con los conjuntos K, P y KP armados y devolverá un subconjunto de P, al que se llamará P*. Como P fue construido con los índices de los elementos de S, P* será un conjunto de índices. Para construir la solución de Exact Cover se deben tomar los elementos de S correspondientes a los índices de P*.

Ejemplo

Se tiene una instancia de Exact Cover con:

- un conjunto X = {X1, X2, …, X7}
- una colección de subconjuntos S = [{X1, X2, X6}, {X1}, {X2, X5, X7}, {X3, X4, X6}]

y se desea conocer la solución del problema mediante una reducción polinómica a Casting.

Para reducir el problema a Casting se construyen:

- una colección K con los elemento de X, K = {X1, X2, …, X7}
- un conjunto P a partir de los índices de los elementos de S, P = {1, 2, 3, 4}
- una colección de listas LX, donde cada lista corresponde a un elemento Xj y los subconjuntos Si que lo contienen.
  LX = [X1:{1, 2}, X2:{1, 3}, X3:{4}, X4:{4}, X5:{3}, X6:{1, 4}, X7:{3}]

A partir de dichos elementos Casting devolverá P* = {4, 3, 2} dado que es el único subconjunto de P que representa todas las características de K sin que haya dos o más personas que compartan características de K.
Para obtener la solución de Exact Cover a partir de P* se deben tomar los elementos de S correspondientes a los índices 2, 3, y 4:

S* = [{X1}, {X2, X5, X7}, {X3, X4, X6}]

Al reducir Exact Cover en Casting se prueba que Casting es al menos tan difícil como Exact Cover, y al ya haber probado que Casting pertenece a NP se puede concluir que Casting pertenece a NP-C.

2. Exact Cover es NP-C

Dada una colección S de subconjuntos de un set X : {Xi, i = 1, 2, …, t} se desea determinar si existe una subcolección S* de S que verifique que cada Xi está contenido en exactamente un subconjunto de S*.

Así las siguientes condiciones deben satisfacerse:

- La intersección de dos subconjuntos cualesquiera de S*, debería ser vacía.
- La unión de todos los subconjuntos de S* es X.

Certificación polinomial de Exact Cover:

La certificación debe verificar que una colección S* dada cubre todos los elementos del set X una y solo una vez. Esto se puede hacer recorriendo cada subconjunto de S y guardando los elementos que contienen de manera única en una estructura M, para luego recorrer cada subconjunto de S* e ir eliminando los elementos de M a medida que se encuentran en los subconjuntos de S* (si se desea eliminar un elemento que no se encuentra en M la solución S* provista no es válida). Si al finalizar la estructura M se encuentra vacía entonces S* es la solución del problema. Como se puede observar se recorre cada subconjunto perteneciente a S o S* una única vez, siendo la certificación polinomial y probando que Exact Cover pertenece a NP.

Reducción polinomial de 3SAT a Exact Cove:

Para probar que Exact Cover es NP-C se debe probar que también pertenece a NP Hard. Para hacer esto se tomará el problema 3SAT perteneciente a NP Hard y se lo reducirá polinomialmente a Exact Cover.

Primero una descripción de 3SAT:

Se tiene una expresión booleana formada por un conjunto de cláusulas conjuntas C, donde cada cláusula es una disyunción lógica de 3 literales booleanos pertenecientes a L. Se desea saber si es posible satisfacer la expresión booleana, o lo que es lo mismo, si existe una asignación de valores para L tal que la expresión booleana devuelve TRUE.

Para reducir 3SAT a Exact Cover se debe crear, a partir de la expresión booleana, una colección S y un set X tal que hay una cobertura exacta de X en S si y sólo si la expresión booleana se puede satisfacer.

Se construirá el set X con los literales de L, las cláusulas de C y elementos Pi,j para cada ocurrencia del literal Lj en la cláusula Ci.

Para construir los subconjuntos de S se crearán dos conjuntos por cada literal Lj, un conjunto "verdadero" contendrá a Lj y a cada elemento P donde Lj esté negado. El otro conjunto "falso" contendrá al elemento Lj y a cada elemento P donde Lj se encuentre sin negar. También por cada elemento Pi,j crearemos un conjunto {Ci, Pi,j}; y por último por cada elemento Pi,j crearemos el conjunto {Pi,j}

Como los conjuntos verdadero/falso son los únicos conteniendo a las variables Lj, por cada valor de j uno y solo uno deberá ser seleccionado por Exact Cover. También la solución debe, para cada Ci, contener algún elemento {Ci, Pi,j}, si Lj es un literal no negado en la cláusula Ci entonces el conjunto "falso" de Lj no puede estar en la solución, ya que también contendrá a Pi,j.

Para obtener la solución de 3SAT a partir de los conjuntos retornados por Exact Cover se deben asignar los valores a los literales según el conjunto correspondiente a ese literal devuelto. Si la solución contiene al conjunto "verdadero" de Lj se le asignará True, si contiene al conjunto "falso" se le asignará False. Recorrer el conjunto solución y compararlo con los conjuntos Lj "verdadero" y Lj "falso" puede resolverse en tiempo polinomial.

Habiendo reducido 3SAT a Exact Cover podemos afirmar que Exact Cover es al menos tan difícil como 3SAT, al ser 3SAT NP-C Exact Cover es al menos NP Hard, y al poder certificarse en tiempo polinomial entonces Exact Cover es NP-C.

3. Qué ocurriría si Exact Cover perteneciera a P

Definición NP completo: NP-C es un conjunto de problemas que cumplen la condición de pertenecer al conjunto NP y al conjunto NP Hard, esto significa que los problemas NP-C son aquellos que cumplen:

- Dada una solución del problema dicha solución se puede verificar en tiempo polinomial.
- Cualquier problema en NP se puede reducir polinomialmente al problema en cuestión.

El problema Exact Cover pertenece a la clase NP-C dado que cumple las dos condiciones mencionadas, pero no pertenece a la clase P dado que no se conoce ningún algoritmo que pueda resolver en tiempo polinomial.

Si se encontrase algún algoritmo capaz de resolver Exact Cover en tiempo polinomial Exact Cover pasaría a pertenecer a la clase P, pero al ser NP-C cualquier problema en NP podría reducirse polinomialmente a Exact Cover y resolver polinomialmente, lo que significa que cualquier problema en NP podría resolverse polinomialmente, en otras palabras, el conjunto NP pasaría a ser igual al conjunto P.

4. Complejidad de un problema reducible a Exact Cover

Que un problema A pueda reducirse polinomialmente a un problema B indique que la dificultad de B es al menos tan difícil como A. Conociendo la dificultad de B conocemos la cota superior de la dificultad de A.

Si X es reducible polinomialmente a Exact Cover, como sabemos que Exact Cover es NP-C entonces X será a lo sumo NP-C. No se puede definir si X pertenece a P, NP (exclusivamente) o NP-C. Si se puede definir que no es NP Hard exclusivamente, dado que puede reducirse a un problema NP.

5. Relación entre clases de complejidad

Las clases de complejidad son conjuntos de problemas computacionales cuya complejidad basada en recursos computacionales está relacionada.

Clase P: Esta clase contiene a los problemas para los que se conoce un algoritmo que puede resolver el problema en tiempo polinomial, sin importar el orden del exponente.

Clase NP: Su definición no se basa en hallar la solución a los problemas, sino en verificar que cierta solución dada es válida. Los problemas pertenecientes a NP son aquellos que pueden certificarse en tiempo polinomial, o lo que es lo mismo, puede verificarse que una solución es válida o no en tiempo polinomial.

Clase NP Hard: La definición de esta clase no se basa ni en la solución ni en la certificación del problema, sino en la posibilidad de reducir otros problemas en estos. Un problema X es NP Hard cuando cualquier problema de la clase NP puede reducirse polinomialmente en X.

Clase NP-C: Esta clase agrupa a los problemas que son tanto NP como NP Hard. Un problema X pertenece a NP-C si puede certificarse polinomialmente y si cualquier problema en NP puede reducirse polinomialmente a X.

Así no se puede establecer una relación clara entre problemas P, NP y NP-C dado que las definiciones de estas clases se basan en distintos criterios: Resolver, certificar o reducir. De esto y de la imposibilidad de encontrar soluciones polinómicas para los problemas NP nace el dilema P vs NP.

Dado que todos los problemas P pueden resolverse en tiempo polinomial también pueden certificarse en tiempo polinomial. Entonces el conjunto P es un subconjunto del conjunto NP, pero al ser la definición de NP abierta respecto a la complejidad algorítmica para hallar la

solución de los problemas, los problemas de NP podrían estar incluidos en P. Un problema en NP para el que se halla una solución polinómica no deja de pertenecer a NP, sino que también pasa a pertenecer a P.

En particular si se encuentra una solución polinómica para un problema X dicho problema pasa a pertenecer a P, sin embargo, si el problema X pertenece a NP-C las consecuencias son aún mayores. Dada NP-C está incluido en NP y cualquier problema NP es reducible (en tiempo polinomial) en cualquier problema NP-C, cualquier problema NP-C es reducible en cualquier otro problema NP-C. Entonces si algún problema de NP-C pasará a pertenecer a P, todos los problemas pertenecientes a NP podrían resolverse en tiempo polinomial, dado que serían reducibles en tiempo polinomial a un problema que puede resolverse en tiempo polinomial.

Referencias consultadas

Computers and Intractability: A Guide to the Theory of NP-Completeness - M.R. Garey; D.S. Johnson

Reducibility among combinatorial problems - Richard M. Karp

Automata Theory CS411-2015S-18 Complexity Theory II: Class NP – David Galles

https://www.topcoder.com/thrive/articles/Minimum%20Cost%20Flow%20Part%20Two:%20Algorithms

https://cp-algorithms.com/graph/min_cost_flow.html#undirected-graphs-multigraphs

https://cp-algorithms.com/graph/dijkstra.html#implementation

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.23.4422&rep=rep1&type=pdf

http://www.cs.uu.nl/docs/vakken/an/an-mincostflow-2016.pdf

/http://perso.ens-lyon.fr/eric.thierry/Graphes2010/amaury-pouly.pdf

Correcciones

Parte 1

1) Explicar Min Cost Max Flow

El problema del camino mínimo de mayor flujo es un problema particular del problema general conocido como “Minimum cost flow”. Definimos:

- G un grafo con n nodos y m aristas.
- c costo
- s un nodo fuente de G
- t un nodo sumidero de G.
- r un valor de flujo

Queremos encontrar un flujo de s a t con valor r, con c costo mínimo. En particular, para el caso del mayor flujo, es el que nos interesa analizar.

Este problema no siempre tiene solución, ya que en el caso en que la sumatoria de los flujos que ingresan a los nodos y los que egresan sea distinta de cero, no habrá solución posible. Para que sea posible encontrar una solución, se deben cumplir 3 condiciones en el grafo:

- El grafo es dirigido
- Todos los costos son mayores o iguales a 0
- La diferencia entre el flujo que ingresa y egresa de todos los nodos de la red, es cero. De esta regla están excluidos los nodos fuente y sumidero.

Dadas estas condiciones, es posible aplicar varios algoritmos para resolver el problema, basándose en el grafo residual. Además los algoritmos que se usan hoy en día para resolver este problema también requieren que todos los valores de capacidad, costo y flujo sean enteros.

En el caso de ***Cycle Cancelling Algorithm*** se parte de un grafo con ciclos negativos y va aumentando el flujo en dichos ciclos. En cada iteración se mantienen flujos factibles que van mejorando progresivamente el flujo. Una vez que se cancelan todos los ciclos con costo negativo, se encuentra el flujo máximo. Un problema de este algoritmo es que no indica un criterio a la hora de seleccionar los ciclos negativos del grafo, con lo cual, la complejidad

temporal puede variar significativamente en función de cómo decidamos elegir esos ciclos negativos, hay formas de elegir correctamente los ciclos negativos, logrando que la complejidad del mismo tienda a la polinomial.

El algoritmo que elegimos es ***Successive Shortest Paths****, este* busca el camino de flujo máximo de menor costo, y se puede usar cuando no hay ciclos negativos en nuestro grafo y también hace uso del grafo residual.

Lo que se hace es ir encontrando todos los caminos posibles que hay desde la fuente al sumidero, y de todos estos elige el de menor costo con algún algoritmo, puede usarse Dijkstra, luego se encuentra el cuello de botella para ese camino y se actualiza el grafo residual. Se itera sobre el grafo residual aplicando el mismo método, hasta que ya no haya caminos.

2) Explicar cómo funciona el algoritmo seleccionado. Incluir: pseudocódigo, análisis de complejidad espacial, temporal y optimalidad.

Complejidad

En cuanto a la complejidad temporal, es pseudo polinomial.

$$O(V * E) * O(E * logV) = O(E^2 * VlogV)$$

Como en *Edmons-Karp* hay un total de E*V iteraciones.

En cada iteración, se calcula el camino mínimo de s a t. Para calcular estos caminos, se utiliza Dijkstra de complejidad O(E*logV).

La cual es pseudo polinomial. Si acotamos al término a E<V2 resulta

$$O(V^3) * O(V^2 \log V) = O(V^5 log(V))$$

En cuanto a la complejidad espacial teórica, tenemos en cuenta el espacio de un grafo residual

$$O(V + 2E) = O(V + E)$$

Optimalidad

Justificación optimalidad de Ford-Fulkerson para luego justificar Successive-shortest-paths.

1- En cada instancia intermedia, el valor de los flujos y capacidades residuales son números enteros.

2- En cada instancia el valor del flujo aumenta. Cada camino s-t sale de s y no regresa, por lo que el primer eje tomado es hacia adelante y aumenta el F(e) en bottleneck (P,f). Sea f el flujo en G, P camino simple s-t en el grafo residual. Entonces v(f') = v(f) + bottleneck(P,f). bottleneck > 0.

3- El algoritmo terminará en un número finito de iteraciones. Si se llama C la suma de todas las capacidades de los ejes que salen de la fuente s. v(f) ≤ C. En cada iteración el v(f) crece, en el peor de los casos en 1. Por lo que en C iteraciones como máximo se terminará la ejecución.

Su complejidad es O(|E|C). La cantidad de vértices es menor a los ejes. El residual tiene a lo sumo el doble de los ejes del original. Buscar caminos de aumento (BFS o DFS) es O(V+E) que puede tomarse como O(E). Modificar el grafo para aumentar el flujo tomará a lo sumo O(V) y construir el residual tomará O(E). El proceso se iterará a lo sumo C veces.

4- Sea f un flujo s-t y (A,B) cualquier corte, entonces v(f) = f out(A) - f in(A).

v(f) = f out(S), f in(S) = 0 => v(f) = f out(S) - f in(S). Como todos los vértices de A (menos s) son internos: f out(v) - f in(v) = 0. Entonces:

$$v(f) = \sum_{v \in A} (f\ out(v) - f\ in(v)).$$

Si los ejes de un vértice en A solo se conectan con otro en A entonces su f out(e) y f in(e) se cancelan. Solo quedará el aporte de los ejes que entran de B y que salen a B

.

$$\sum_{v \in A} f\,out(v) \;-\; f\,in(v) \;=\; \sum_{e\ out\ of\ A} f(e) \;-\; \sum_{e\ into\ A} f(e) \;=\; f\,out(A) \;-\; f\,in(A)$$

5- Sea f un flujo s-t, (A,B) un corte s-t. Entonces v(f)≤ c(A,B)

$$v(f) \;=\; f\,out(A) \;-\; f\,in(A) \;\leq\; f\,out(A) \;=\; \sum_{e\ out\ of\ A} f(e) \;\leq\; \sum_{e\ out\ of\ A} Ce \;=\; c(A,B).$$

6- Si f es un flujo s-t, tal que no hay un camino s-t en el grafo residual Gf, entonces existe un corte s-t (A*, B*) en G, en el que v(f) = c(A*, B*). Por lo tanto f tiene el máximo valor de cualquier flujo en G y (A*, B*) tiene la mínima capacidad de cualquier corte s-t en G.

Todos los ejes de A* a B* están saturados y todos los ejes de B* a A* están sin utilizar, por lo tanto:

$$v(f) \;=\; f\,out(A\,^*) \;-\; f\,in(A\,^*) \;=\; \sum_{e\ out\ of\ A^*} f(e) \;-\; \sum_{e\ into\ A^*} f(e) \;=$$

$$=\; \sum_{e\ out\ of\ A^*} Ce \;-\; 0 \;=\; C(A\,^*, B\,^*)$$

Dado que el algoritmo termina cuando no hay caminos s-t en el Gf, y dado 6) el flujo retornado por el algoritmo es el flujo máximo.

Al ser Successive-Shortest-Paths tan parecido al algoritmo Edmonds-Karp, que a su vez es similar a Ford-Fulkerson, su optimalidad es evidente.

Edmonds-karp, se diferencia de Ford-Fulkerson a la hora de seleccionar el camino de aumento. Mientras que en Ford-Fulkerson es aleatoria, en Edmonds-Karp se selecciona el camino más corto. Teniendo en cuenta que Ford-Fulkerson es óptimo, Edmonds-Karp también debe serlo, ya que el criterio de elección de caminos no presenta cambios en la

obtención del flujo máximo. El criterio de elección no modifica su optimalidad porque en definitiva ambos hacen lo mismo, buscar caminos s-t.

Siguiendo esta lógica, Successive-Shortest-Paths también es similar a los algoritmos anteriores. El único cambio, es el criterio de elección de caminos. Por lo que Successive-Shortest-Paths es óptimo para el problema de flujo máximo. Solo queda justificar, si este flujo máximo es el de mínimo costo. Todo indica que sí, debido a que el flujo máximo se obtiene utilizando los caminos de aumento del menor costo. De esta manera, se consumen los caminos menos costosos progresivamente. Por lo tanto, si se prefirieron los caminos menos costosos, y se obtuvo el flujo máximo, este debe ser el flujo máximo de menor costo.

Cabe aclarar que para utilizar Dijkstra durante el algoritmo, previamente se modifica el grafo para descartar ciclos negativos, por lo que el algoritmo encuentra los caminos s-t..

4) Programar el algoritmo.

Instrucciones para ejecutar el código:

Hay 3 scripts en el root del proyecto

- run-example.sh : Ejecuta el código para un caso en particular (ya tiene una entrada de ejemplo)
- run-example-v.sh: Ejecuta en modo verboso
- run-test.sh: ejecuta todos los test.

Ejecución:

En el directorio de los fuentes de la solución ejecutar:

Para modo normal: _> python3 ./main.py -f “path_to_file”

Para modo verbose: _> python3 ./main.py -v True -f “path_to_file”

5) Responder justificando: ¿La complejidad de su algoritmo es igual a la presentada en forma teórica?

No, no son las mismas, a continuación detallamos las complejidades prácticas obtenidas.

Complejidad temporal

$$O(V * E * V) + O(E) * (O(E * logV) + O(E) + O(E)) = O(V^2 E) + O(2E^2 + Elog(V)$$

El primer término corresponde a armar al grafo residual, se recorre el grafo, por cada vértice y luego por cada arista del vértice que a lo sumo, puede tener tantas aristas.

Luego el segundo término corresponde al ciclo de recorrer todos los caminos desde la fuente al sumidero. Dentro de este hay 3 sub términos. El primero es al aplicar Dijkstra. El segundo es el de encontrar el cuello de botella (en el cual se recorren todas las aristas del camino) y el último subtermino es el de actualizar el grafo residual residual.

Complejidad espacial

Se usaron 4 estructuras:

Diccionario de grafo residual cuyas claves son los nodos y como valores tiene aquellos nodos a los que llega. (A lo sumo O(V*V))

Lista de nodos (O(V))

Lista de aristas (O(E))

Una lista para encontrar el camino entre 2 nodos (A lo sumo O(E))

Con lo cual el Orden espacial es

$$O(V * V) + O(V) + O(E) + O(E) = O(V^2) + 2O(E) + O(V) = O(V * V) + 2O(E)$$

Parte 2

Certificación polinomial de Casting (Corrección reentrega):

El algoritmo que resuelve casting recibe una lista de características K, una lista de personas N y una colección de listas NK (cada lista NKj posee una características Kj y las personas que la poseen). Entonces, la certificación de casting debería recibir la misma entrada y adicionalmente una lista P de personas, subconjunto de N, que sea solución para el K, N y NK dados.

Para verificar que P es solución, primero hay que verificar que P es subconjunto de N. Para esto por cada persona en P se puede recorrer N, verificando que exista en N. Esto se hace en tiempo polinomial O(P*N).

También se debe verificar que todas las características de K, se encuentran representadas por el conjunto de personas en P, sin repetirse características en dos o más personas.
Para realizar esto se puede iterar P y por cada persona en P iterar las listas de NK, si la persona Pi pertenece a la lista NKj que está siendo iterada entonces se debe remover la característica Kj de K correspondiente a NKj.
Si al finalizar, la lista de caracteristicas K se encuentra vacía, entonces todas las características se encuentran representadas por la solución P.
Si en algún momento de la iteración se intentó remover una característica Kj no presente en K, entonces dicha característica se encuentra repetida en dos personas y en ese caso P no es solución válida.
Esta verificación requiere iterar P y NK de manera anidada, y adicionalmente K cuando se desea remover una característica (pero dado que NK y K poseen el mismo tamaño de entrada se puede ignorar) por lo que su complejidad temporal es polinómica.

Finalmente dado que Casting se puede certificar en tiempo polinomial pertenece a NP.