Teoría de Algoritmos I

75.29 / 95.06

Trabajo Práctico 2

Amigo, Nicolás	105832
Nicolini, Franco	105632
Serrano, Maria Candela	105287
Singer, Joaquín	105854
Primerano Lomba, Franco Alejandro	106004

Profesores:
Víctor Podberezski
Lucas Ludueño
Kevin Untrojb
Jose Sbruzzi
Ernesto Alvarez

31 de Mayo de 2021

Índice general

Parte 1: Un juego amistoso de cartas

1.1. Ejecución

Al estar programado en python no se realiza una compilación del programa, simplemente se puede ejecutar mediante la linea: `python3 main.py <"archivo"` donde "archivo" debe ser reemplazado por el nombre del archivo de prueba a utilizar, adjuntos en el .zip entregado.

1.2. Relación de recurrencia y subproblemas

La idea detrás de nuestra ecuación es la siguiente: Para sumar el mayor puntaje posible, se maximiza entre las dos elecciones que el juego permite tomar, o sea, elegir la primera carta o la última. Luego de cada elección del primer jugador, el segundo, al estar usando la misma estrategia, se quedará con el mejor de los dos subproblemas que se pueden generar, minimizando la cantidad de puntos sumada por el primer jugador.

Cada subproblema será una instancia del juego de cartas, el cual se resolverá a partir de otros tres subproblemas. Estos nacen analizando lo que pasa luego de la elección del rival. Si el primer jugador elije la carta que se encuentra al inicio, a partir de ahora i, el segundo podrá elegir $i+1$ o la que se encuentra al final, a partir de ahora f. Si el primer jugador elije f, el segundo podrá elegir entre i y $f-1$. Por lo cual, en el siguiente turno del primer jugador, pueden aparecer estos tres subproblemas:

- $s(i+2, f)$
- $s(i+1, f-1)$
- $s(i, f-2)$

Siendo s una instancia del problema.

Entonces, sea v_k el valor de una carta en la posición k, la ecuación de recurrencia puede ser expresada como:

$$s(i,f) = \begin{cases} v_i & \text{si } f = i \\ max(v_i, v_f) & \text{si } i+1 = f \\ max(v_i + min(s(i+2,f), sub(i+1,f-1)), v_f + min(s(i,f-2), sub(i+1,f-1))) & \text{si } f > i+1 \end{cases}$$

1.3. Programación Dinámica

Para la resolución de este problema se utilizó programación dinámica.[2]

La programación dinámica consiste en la división de un problema en subproblemas. Una vez dividido en estos subproblemas, se buscan las soluciones óptimas de los mismos para encontrar la solución óptima global del problema inicial utilizándolos. Para evitar recalcular los subproblemas que aparecen repetidos se utiliza la memoización, la cual es una técnica que consiste en almacenar los resultados de los subproblemas calculados para evitar repetir su resolución cuando vuelva a requerirse, consiguiendo reducir significativamente la complejidad temporal de la solución.

El problema cumple con las propiedades que se necesitan para ser resuelto de esta manera: Contiene una subestructura óptima, es decir, contiene en su interior subproblemas cuyas soluciones óptimas llevan a una solución óptima global, y contiene subproblemas superpuestos, es decir, en la resolución de sus subproblemas vuelven a aparecer subproblemas previamente calculados.

1.4. Subestructura y problemas superpuestos

Los problemas a resolver aparecerán de forma repetida, como se puede ver en la Figura 1.1. Para resolver esto, habrá memorizar los resultados de los subproblemas y así evitar calcularlos de forma repetida, reduciendo la complejidad de la solución de forma drástica, pasando de una exponencial a una polinómica.

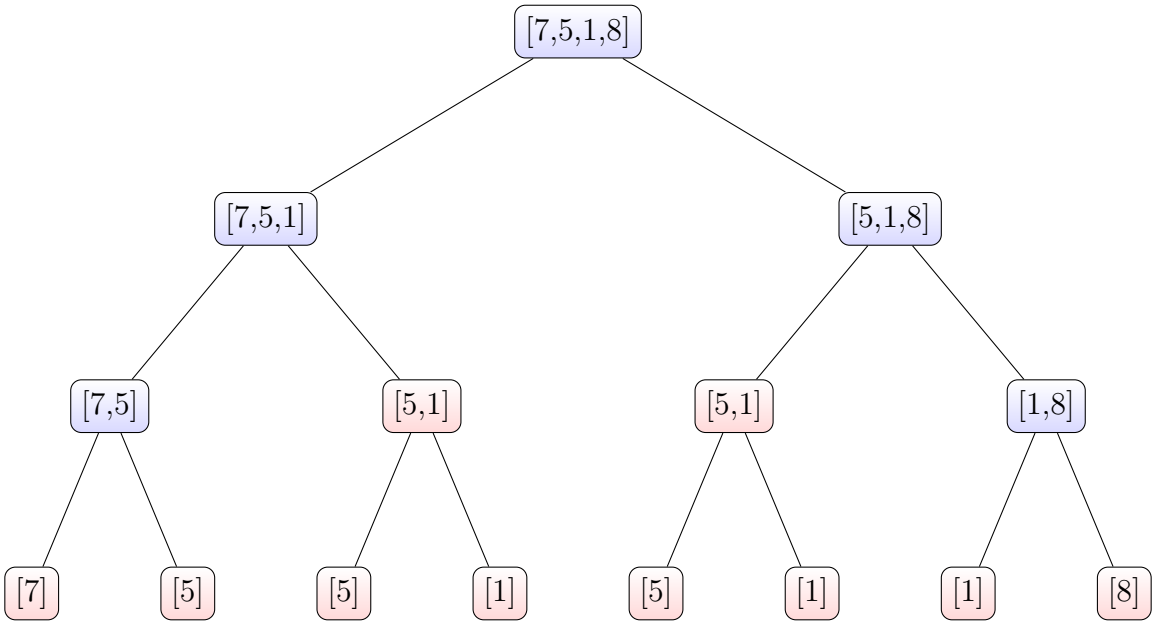

Figura 1.1: Árbol de subproblemas para una instancia $[7, 5, 1, 8]$

Cada suproblema (que no se corresponda con un caso base) se resuelve utilizando los subproblemas que se encuentran dos filas por debajo.

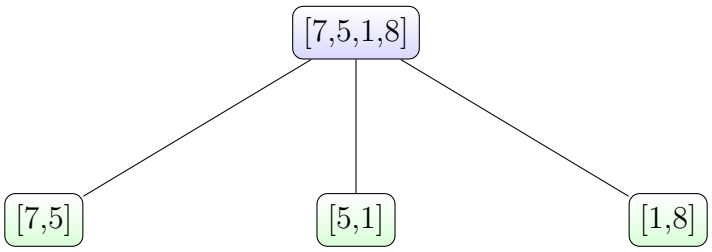

Figura 1.2: Subproblemas necesarios para resolver una instancia $[7, 5, 1, 8]$

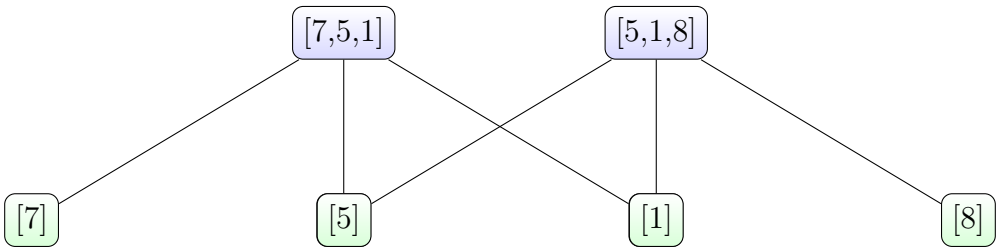

Figura 1.3: Subproblemas necesarios para resolver las instancias $[7, 5, 1]$ y $[5, 1, 8]$

1.5. Solución iterativa y análisis de la misma

Se utilizará una función que devolverá un diccionario con la mejor jugada posible dado un intervalo. De éste se obtendrá que carta eligió cada jugador en sus turnos y el puntaje obtenido.

La solución iterativa utiliza memorización, almacenando en un diccionario la posición de la mejor carta a seleccionar en un subproblema dado. Además almacenará en otro diccionario el mayor puntaje que puede alcanzar el jugador al que le toca elegir una carta, si ambos jugadores realizan elecciones óptimas durante el resto de la partida. Los dos primeros bucles cargaran en los diccionarios los casos donde la cantidad de cartas es una o dos, es decir los casos bases. Luego, se iterará entre los subproblemas, yendo desde abajo hacia arriba en la estructura, recorriéndola con dos bucles anidados: Uno irá aumentando la cantidad de cartas en cada subproblema y el otro la carta final del intervalo. Para calcular la posición de la primera carta, simplemente se resta la cantidad a la posición de la carta final. Cada subproblema resuelto será memorizado para poder ser utilizado por otros más grandes. En cada iteración se calculará cuál será la jugada óptima que debe realizar un jugador dado el subproblema actual, para ello seleccionará el caso en que maximiza su ganancia de puntos considerando las elecciones que tomará el otro, utilizando los resultados calculados en las iteraciones previas. Una vez calculada la mejor carta para cada intervalo se simulará el juego entre dos oponentes que utilizaran el diccionario para tomar la mejor carta dado el intervalo que les toque.

Pseudocódigo:

```
1   cartas es una lista de largo n
2
3   fun orden_de_la_mejor_carta_posible (cartas)
4       sumaParcial = Diccionario vacio
5       posCalculadas = Diccionario vacio
6
7       i desde 0 a n-1:
8           posCalculadas[(i,i)] = cartas[i]
9
10      i desde 0 a n-2:
11          posCalculadas[(i,i+1)] = maximo(cartas[i], cartas[j])
12
13      dif de 2 a n-1:
14          j de 0 a dif:
15              i = j - dif
16              seleccionPrimera = cartas[i] + minimo(sumaParcial[(i+2,j)],
                    sumaParcial[(i+1,j-1)])
17              seleccionUltima = cartas[j] + minimo(sumaParcial[(i+1,j-1)],
                    sumaParcial[(i,j-2)])
18
19              Si seleccionPrimera > seleccionUltima :
20                  posCalculadas[(i,j)] = i
21                  sumaParcial[(i,j)] = seleccionPrimera
22              Sino:
23                  posCalculadas[(i,j)] = j
24                  sumaParcial[(i,j)] = seleccionUltima
25      devolver posCalculadas
26
27  fun puntaje(jugadaOptima, cartas)
28
29      turnoJugador1 = verdadero
30      lista1 = lista vacia
31      lista2 = lista vacia
32      i = 0
33      j = n- 1
34      De x a n:
35          Si turnoJugador1:
36              puntosJugador1 + cartas[jugadaOptima[(i,j)]]
37              lista1 + [cartas[jugadaOptima[(i,j)]]]
38          Sino:
39              puntosJugador2 + cartas[jugadaOptima[(i,j)]]
40              lista2 + [cartas[jugadaOptima[(i,j)]]]
41
42          Si jugadaOptima[(i,j)] = i :
43              i + 1
44          Sino:
45              j - 1
46          turnoJugador1 = !turnoJugador1
```

1.6. Ejemplo de funcionamiento del programa

El ejemplo utilizado es el propuesto en el enunciado, donde la fila de cartas esta dada por [7],[5],[1],[8].

Figura 1.4: Diagrama de subproblemas con diferencia igual a 0

Comenzando desde abajo hacia arriba, en la Figura 1.4 se puede observar la primera iteración del algoritmo, la cual es uno de los casos base, donde la elección que se guardará será seleccionar la última carta restante.

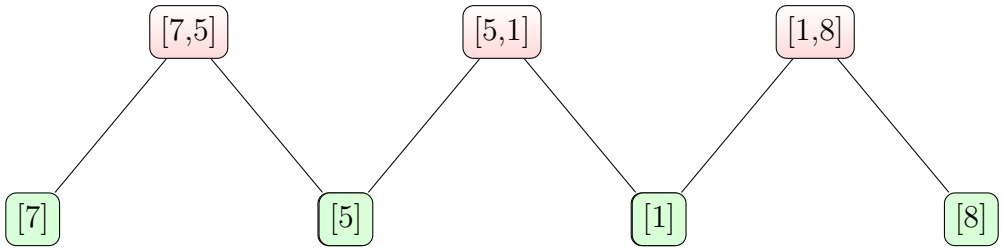

Figura 1.5: Diagrama de subproblemas con diferencia igual a 1

En la Figura 1.5 se presenta el segundo caso base, que consiste en tomar una elección cuando quedan dos cartas. En este caso, se memorizará como elección la que otorgue un mayor puntaje al jugador.

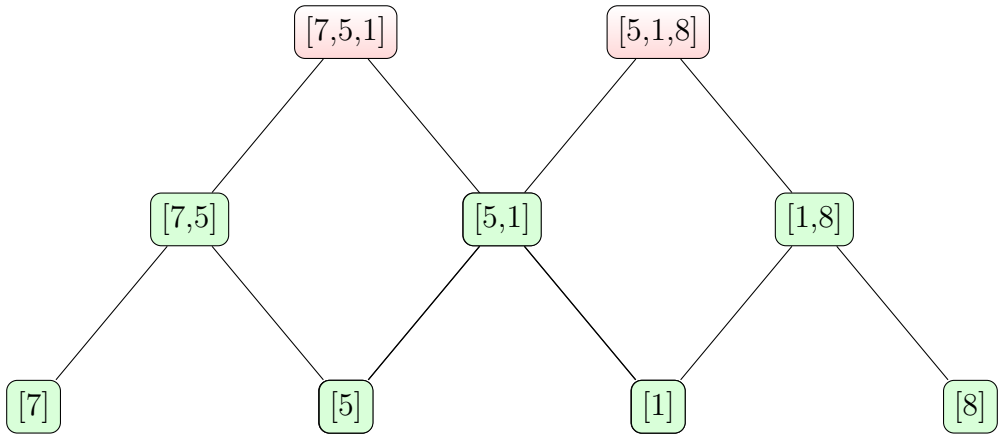

Figura 1.6: Diagrama de subproblemas con diferencia igual a 2

Luego, en los subproblemas que constan de tres cartas (mostrados en la Figura 1.6), se aplicará la ecuación de recurrencia que fue brindada anteriormente, pero como los resultados de los

subproblemas ya fueron calculados, alcanzará con reemplazar los valores, obteniendo:

$$sub(0,2) = max(7 + min(1,5), 1 + min(7,5)) = max(8,6) = 8$$

Como $8 > 6$, se elije la primera carta (7).

$$sub(1,3) = max(5 + min(8,1), 8 + min(5,1)) = max(6,9) = 9$$

Como $6 < 9$, se elije la última carta (8).

Se almacenan ambos resultados.

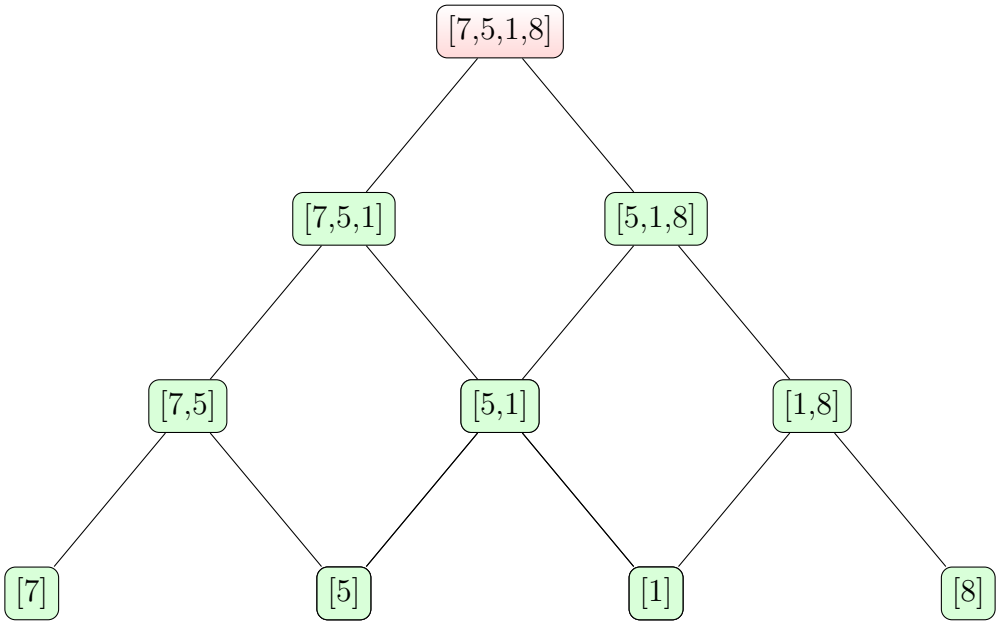

Figura 1.7: Diagrama de subproblemas con diferencia igual a 3

Para el último paso, señalado en la figura 1.7, se aplica el mismo procedimiento que en el anterior, aunque ahora solo queda un último subproblema (que es el problema que se está queriendo resolver).

$$sub(0,3) = max(7 + min(8,5), 8 + min(7,5)) = max(12,13) = 13$$

Como $12 < 13$, se elije la última carta (8).

1.7. Complejidad temporal de la solución

En primer lugar, la función "orden_de_la_mejor_carta_posible" realiza dos ciclos de longitud N (cantidad de cartas) donde establece los casos bases, para ello utilizan las funciones nativas de python "max", "min" y "len"[1] cuya complejidad es $O(1)$ y los bucles $O(N)$. Luego realizará dos bucles anidados con una complejidad $O(N^2)$.

En la función "imprimir_puntaje" se realiza un bucle de N iteraciones, con complejidad $O(N)$.

Finalmente, si se considera que el acceso a diccionarios[1] es $O(1)$ la complejidad temporal resultante será $O(N^2)$, en caso contrario será $O(N^3)$.

1.8. Complejidad espacial

Para solucionar el problema, se utilizan dos diccionarios, uno para almacenar el índice de la mejor carta dado un intervalo y otro para almacenar la puntuación máxima posible a obtener (suponiendo un juego óptimo por ambos jugadores), para el jugador al que le toque elegir primero, en un subproblema con el intervalo dado. La complejidad espacial de estos será en principio $O(2N^2)$.

Para la parte de imprimir las cartas elegidas por cada jugador, se optó por usar dos listas que tendrán una complejidad espacial de $O(\frac{N}{2})$ resultando un total de $O(N)$ entre ambas.

Finalmente, la complejidad espacial utilizada es $O(N^2)$.

1.9. Pruebas del programa

Para probar el programa se eligieron los archivos `primer_prueba.txt` y `segunda_prueba.txt`.

El output esperado (con las convenciones elegidas para elecciones igual de buenas) ingresando la primera prueba es:

```
1 Jugador 1:
2 Cartas elegidas: [87, 105, 107, 126, 82, 25, 88, 52, 17, 21, 81, 137, 83, 47, 99,
      4, 129, 25, 117, 68, 145, 129, 55, 139, 120]
3 Puntos sumados: 2088
4
5 Jugador 2:
6 Cartas elegidas: [15, 49, 29, 77, 67, 112, 36, 32, 100, 54, 85, 100, 71, 48, 81,
      117, 93, 77, 142, 138, 7, 36, 14, 35, 104]
7 Puntos sumados: 1719
```

El output esperado (con las convenciones elegidas para elecciones igual de buenas) ingresando la segunda prueba es:

```
1 Jugador 1:
2 Cartas elegidas: [143, 44, 10, 139, 8, 143, 34, 30, 131, 8, 147, 29, 45, 96, 28,
      113, 14, 3, 24, 129, 29, 137, 146, 98, 46]
3 Puntos sumados: 1774
4
5 Jugador 2:
6 Cartas elegidas: [109, 35, 72, 38, 36, 135, 9, 112, 84, 115, 111, 110, 2, 11, 2,
      96, 64, 77, 75, 12, 69, 27, 64, 117, 4]
7 Puntos sumados: 1586
```

Parte 2: La red de espías

2.1. Redes de flujo

Una red de flujo[3] es un grafo dirigido que tiene dos vértices especiales: uno llamado fuente o *source*, el cual no tiene ejes entrantes y un sumidero o *sink*, el cual no tiene ejes salientes. Los vértices sobrantes son llamados vértices internos, estos vértices cuentan con una condición de conservación, la cual dice que todo el flujo entrante a través de los ejes incidentes a ese vértice tiene que ser igual a todo el flujo saliente por todos los ejes que salen de ese eje particular. Esta condición está dada por la ecuación ($v \in V$ donde V son todos los vértices):

$$\sum_{e\ in\ V} f(e) \ = \sum_{e\ out\ V} f(e)$$

Una red de flujos puede ser utilizada para modelar diversos sistemas donde haya algo que viaje entre nodos, como fluidos, el tráfico y sistemas similares. Dentro de las redes de flujo se pueden encontrar varias definiciones, las que se utilizan en la resolución a presentar son las siguientes:

- Dado una red de flujo G y un flujo f en G, se define el **grafo residual** Gf (de G con respecto a f) a los mismos vértices de G los cuales contienen:
  - Ejes hacia adelante: Para cada $e = (u, v) \in E$ en el que $f(e) < Ce$. Donde Ce es la capacidad de transporte de ese eje y $f(e)$ es el flujo. Se incluye en Gf con capacidad $Ce - f(e)$.
  - Ejes hacia atrás: Para cada $e = (u, v) \in E$ en el que $f(e) > 0$. Se incluye $e' = (v, u)$ con capacidad $f(e)$.
- El **corte de grafo** esta dado por la división del grafo en dos mitades: A, que contiene a la fuente, y B, que contiene al sumidero. Hallar el corte mínimo de un grafo es hallar el máximo flujo que puede pasar por el este.
- Un **grafo de nivel** G_L es un grafo tal que:
  - Sus vértices son los vértices del grafo original separados en capas, de modo que la primer capa contiene a todos los vértices adyacentes a la fuente, la segunda capa contiene a todos los vértices aún no incluidos que sean adyacentes a los de la primera capa.
  - Sus aristas son aquellas del grafo original que en el grafo de nivel vayan de una capa a una siguiente, descartando las que no lo cumplan.

- Se llama **flujo bloqueante** a un flujo en el grafo de nivel tal que cualquier camino de la fuente al sumidero contiene una arista saturada; o equivalentemente, que la fuente y el sumidero están desconectados en el grafo residual.

2.2. Presentación del algoritmo

Dada una red de flujo donde cada vértice es un agente, la capacidad de cada arista es 1, la fuente es la agencia emisora de mensajes y el sumidero es el agente destinatario, se describe el siguiente algoritmo para resolver el problema:

El primer paso del algoritmo es hacer un pre-procesamiento sobre la red pasada, esto es necesario para que se cumpla la condición de que un agente solo pueda entregar un único mensaje durante una emergencia. Este consiste en convertir un vértice V_i en dos vértices V_{ie} y V_{is} donde e y s denotan entrada y salida de tal manera que V_{ie} contenga todos las aristas entrantes de V_i, y V_{is} contenga todas las salientes. Finalmente se agrega una arista conectando los dos vértices, yendo de V_{ie} a V_{is}. Este paso se aplica en los casos donde tanto el grado de salida como el de entrada son mayores a 1, como el que se muestra en la Figura 2.1, con grado de entrada y salida 3:

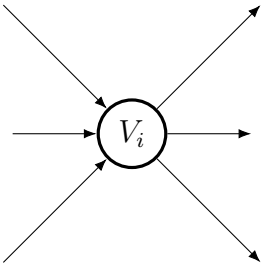

Figura 2.1: Nodo que es necesario pre-procesar

En la Figura 2.2 se puede observar como queda V_i luego de aplicar este paso.

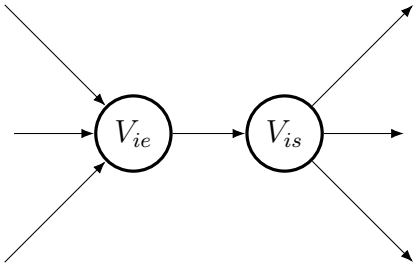

Figura 2.2: Resultado del pre-procesamiento

Este paso tendrá una complejidad de $O(V^2)$ ya que se deberá copiar el grafo, y dicho proceso es $O(V^2)$. Añadir elementos no aumenta la complejidad porque la cantidad de elementos nuevos que se incluyen será menor a V.

A partir de aquí se empleará algún algoritmo ya conocido para hallar el flujo máximo, se podría utilizar Edmonds-Karp, en este caso utilizaremos una leve mejora del mismo y citaremos las fuentes que justifican su funcionamiento y complejidad.

Se parte del algoritmo de Edmonds-Karp[4] visto en la materia, y se tendrá en cuenta que las redes de flujo en la mayoría de los casos presentan mayor cantidad de aristas que de vértices. Ahora bien, si se utiliza DFS en lugar de BFS, encontrar un camino de aumento tomará $O(V)$ en vez de $O(E)$.[5]

El algoritmo de Dinic consiste en los siguientes pasos:

1. Crear un grafo residual, que empieza siendo igual al grafo original
2. Realizar un DFS empezando desde la fuente y formar un grafo de nivel.
3. Buscar un camino con las aristas (pasando de un nivel a uno sucesivo). Cuando se alcance el sumidero aumentar el flujo en ese camino(sobre el grafo residual) y empezar de nuevo en la fuente creando un nuevo grafo de nivel.
4. Si llegara a un vértice sin aristas salientes, borrar ese vértice y todas las aristas entrantes.
5. Detenerse cuando la fuente no tiene aristas salientes.

El costo total del DFS será $O(E \cdot V)$ ya que cada V pasos adelante, resulta en al menos un aumento de flujo o borrado de vértice, pero podrán ocurrir como mucho E veces. El costo total de borrados es $O(E)$. El total costo de aumento de flujo es $O(E \cdot V)$ porque se incrementa como máximo $O(E)$ veces y cada una será en el peor de los casos $O(V)$.

Se puede computar el grafo de nivel en $O(E)$ con un BFS, y como mucho se necesitarán V flujos de bloqueo. Por lo tanto, la complejidad del algoritmo de Dinic es $O(E \cdot V^2)$, una pequeña mejora al algoritmo de Edmonds-Karp.[6]

Como el numero total de aristas que pertenecen al corte mínimo (en este problema en particular) será igual al flujo máximo de la red, se guarda el máximo obtenido del algoritmo de flujo y se usará para calcular la cantidad de aristas que se deberían eliminar. El numero de aristas que se deben eliminar será dado por la formula:

$$n = \lceil r \cdot k \rceil \tag{2.1}$$

Donde n es el número de aristas a cortar, r es el porcentaje (en fracción) por el que queremos disminuir la cantidad de mensajes totales y k es la cantidad máxima de mensajes que se puede mandar. Se necesitaran esta cantidad de cortes ya que todas las aristas tienen igual capacidad (1).

En este caso n también sera la cantidad mínima de vértices (o agentes) a eliminar para reducir el flujo por una cantidad r. Esto se debe a que no existirá un vértice (salvo el sumidero y la fuente)

que este en contacto con mas de una arista del corte mínimo, debido a las condiciones del problema y como está armada nuestra red.

Se puede analizar esto en los 2 casos posibles de nodo en nuestra red donde este podría estar en contacto con mas de una arista del corte mínimo.

Cuando el grado de entrada de un vértice es mayor a 1:

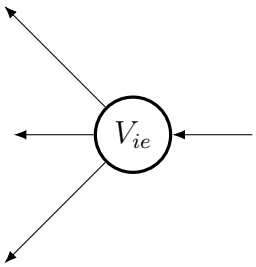

Figura 2.3: Vértice con grado de entrada mayor a 1 en la red residual

En el caso de la Figura 2.3 es imposible ya que recordemos que todas las aristas tienen peso 1 y en este ejemplo sale de V_i un flujo de 3.

Cuando el grado de salida de un vértice es mayor a 1:

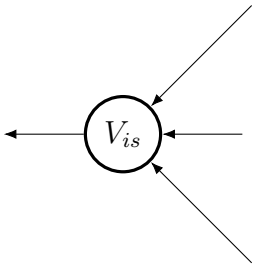

Figura 2.4: Vértice con grado de salida mayor a 1 en la red residual

También podemos ver que la Figura 2.4 es igualmente imposible si consideramos como funciona el algoritmo que calcula la red residual. Luego de realizarse el primer camino a través de V_{is} el algoritmo ya no encontraría otro camino que atraviese ese nodo y por lo tanto no cambiaría de sentido los otras aristas.

2.3. Pseudocódigo del algoritmo

```
1     g es una red que empieza en la agencia y termina en el agente deseado
2
3     fun agentes minimos(g):
4         g' es una copia de g
5
6         por cada vertice v de g:
7             si v tiene grado de salida y el grado de entrada mayores a 1:
8                 Ve, Vs nuevos vertices
9                 en g' incluyo Ve
10                en g' incluyo Vs
11
12                por cada vertice que apunta a v:
13                    en g' incluyo (vertice apunta a Ve)
14                por cada vertice que es apuntado por v:
15                    en g' incluyo (Vs apunta a vertice)
16                en g' incluyo (Ve apunta a Vs)
17                de g' elimino a v
18
19        flujoMaximo = AlgoritmoDeDinic(g')
20        devolver techo (flujoMaximo * 0.3)
```

En el caso de que se quiera un algoritmo mas general donde se puede reducir por cualquier cantidad que se desee se pueden hacer estos pequeños cambios:

```
1     fun agentes minimo(g, porcentaje):
2         ...
3         devolver techo (flujoMaximo * porcentaje)
```

2.4. Complejidad temporal y espacial de la solución

La primer parte consiste realizar la transformación mencionada en la presentación sobre el grafo. Esto se realizará en complejidad $O(V^2)$.

Luego sobre este grafo se aplicará el conocido algoritmo de Dinic para flujo máximo, cuya complejidad es $O(V^2 \cdot E)$.

Finalmente se utilizará el flujo obtenido por este algoritmo para realizar una operación $O(1)$. Por lo tanto el algoritmo tiene una complejidad temporal para resolver el problema de $O(V^2 \cdot E)$.

Para la complejidad espacial se tendrá en cuenta que a partir del grafo original se creará uno nuevo, que en el peor de los casos contendrá el doble de vértices. A partir de este se crearán grafos de nivel en el algoritmo de Dinic, con la misma complejidad espacial. Utilizando una matriz de adyacencia se tendría una complejidad espacial de $O(V^2)$.

Ambas complejidades son polinomiales, ya que ambas cuentan con un solo parámetro y dependen únicamente de los datos de entrada V y E, no como en el algoritmo Ford-Fulkerson original, que dependía del flujo máximo o bien de la capacidad de las aristas de la fuente.

2.5. Ejemplo paso a paso

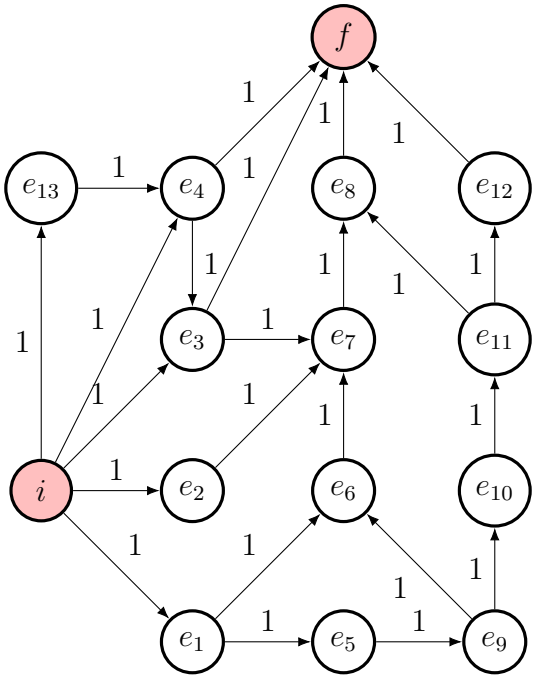

Figura 2.5: Una instancia del problema en forma de red de flujo

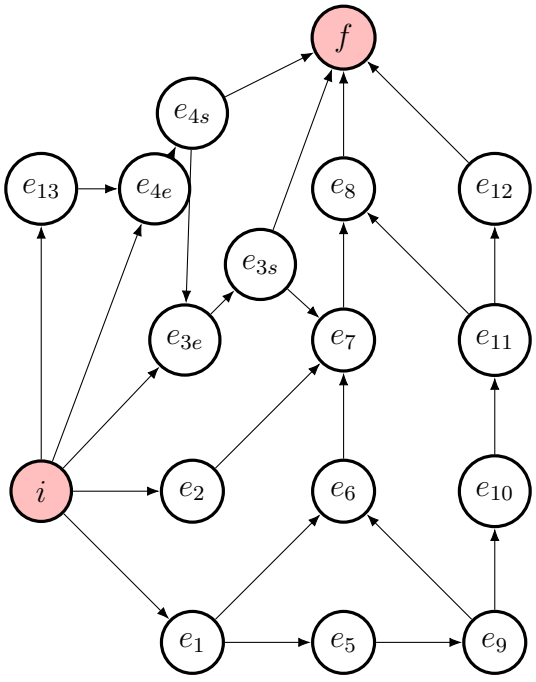

Figura 2.6: Resultado del pre-procesamiento

Dado el grafo dirigido de la Figura 2.5, donde f es el agente destinatario e i la agencia emisora, se aplicará el algoritmo brindado en la sección anterior. Se dará por sobreentendido que todas las aristas tienen capacidad 1.

Como se puede observar en la Figura 2.6, se transforma el grafo aplicando el pre-procesamiento del algoritmo.

Luego se busca el flujo máximo mediante el algoritmo de Dinic. Dando por resultado el grafo con la distribución de flujo de la Figura 2.7.

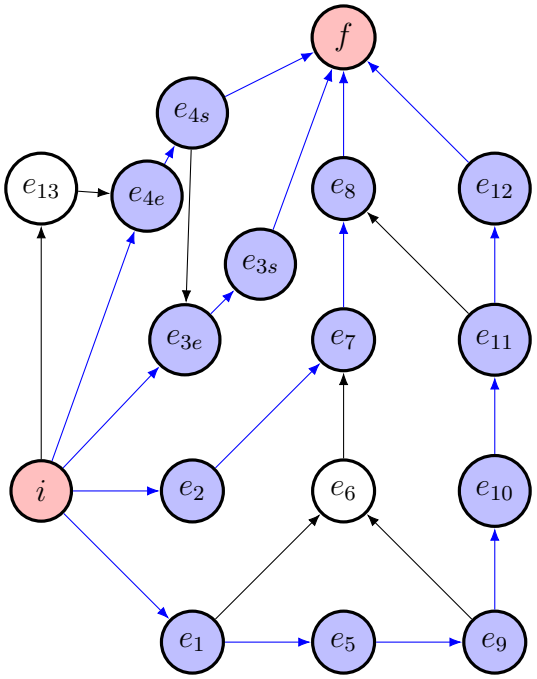

Figura 2.7: Red después de aplicar Dinic

El flujo máximo resultante es 4, por lo que aplicando la fórmula $4 \cdot 0{,}3 = 1{,}2$ y redondeando el resultado hacia arriba, se sabe que la cantidad de espías a neutralizar es 2.

Bibliografía

[1] Python TimeComplexity
https://wiki.python.org/moin/TimeComplexity

[2] Programación dinámica

J. Kleinberg, E. Tardos, Algorithm Design, Addison Wesley (2006).

Capítulo 6

[3] Redes de flujo

J. Kleinberg, E. Tardos, Algorithm Design, Addison Wesley (2006).

Capítulo 7

[4] FIUBA. Teoría de algoritmos 1 . Módulo 6: Edmonds-Karp: Paso a paso

Victor Daniel Podberezski

https://www.youtube.com/watch?v=oslmrRVfgU8

[5] Georgia Tech, College of Computing. Algoritmo de Dinic
https://www.cc.gatech.edu/~rpeng/18434_S15/blockingFlows.pdf

[6] Proof of correctness. Dinic's algorithm
https://cp-algorithms.com/graph/dinic.html