75.29 - Teoría de Algoritmos I: Trabajo Práctico n. 3

Equipo Q:
Lavandeira, Lucas (#98042)
lucaslavandeira@gmail.com

Rozanec, Matias (#97404)
rozanecm@gmail.com

Sbruzzi, José (#97452)
jose.sbru@gmail.com

18.junio.2018

Facultad de Ingeniería, Universidad de Buenos Aires

Índice

I Un juego de batalla naval **3**

1. Instrucciones de ejecución **3**

2. Dinamico **3**
2.1. Explicación del algoritmo . . . 3
2.1.1. Calidad de heurística de Dinámico . . . 3
2.1.2. Complejidad temporal del algoritmo . . . 4
2.2. Condiciones para que Dinámico sea óptimo . . . 4
2.3. Planteo matemático de la hipótesis y demostración . . . 4
2.3.1. Conceptos necesarios para expresar la hipótesis y la prueba . . . 4
2.3.2. Hipótesis . . . 5
2.3.3. Demostración: caso base . . . 5
2.3.4. Planteo del caso inductivo . . . 5
2.3.5. Demostración del caso inductivo . . . 6

3. Greedo **7**
3.1. Explicación del algoritmo . . . 7
3.2. Calidad de heurística de Greedo . . . 7
3.3. Complejidad temporal del algoritmo Greedo . . . 8
3.4. Condiciones para que Greedo sea óptimo . . . 8

4. Posicionamiento inicial de barcos **9**

II Parte 2: Sabotaje! **10**
4.1. Análisis preliminar y consideraciones previas . . . 10
4.2. Algoritmo propuesto . . . 11
4.3. Extensión con varias fuentes y sumideros . . . 11
4.4. Implementación . . . 12

III Anexo: Correcciones de Sabotaje! **13**
4.5. Explicacion del algoritmo propuesto . . . 13
4.6. Naturaleza de heurística del algoritmo construído . . . 14
4.7. Complejidad de la ejecución del algoritmo . . . 14

IV Codigo fuente **16**

Parte I
Un juego de batalla naval

1. Instrucciones de ejecución

En el directorio root del proyecto, correr `npm i` para instalar las dependendicas necesarias. Para la ejecución propiamente dicha, correr el comando `npm start` seguido de los siguientes argumentos:

- ruta al archivo que contiene la información del juego
- selección de estrategia: `greedo` o `dinamico`
- cantidad de lanzaderas
- **opcional** `true` o `false` de acuerdo a si se desea utilizar posiciones iniciales de los barcos distintas a 0. Default: `false`.

2. Dinamico

2.1. Explicación del algoritmo

Algorithm 1: mejoresPartidas(t,d)

Data: Cantidad de turnos t que demoran las partidas retornadas y disparos d que se dispararán a continuación
Result: Partidas con menor cantidad de puntos obtenidas
$d \leftarrow$ siguiente disparo
$t \leftarrow$ turno actual
if $t = 0$ **then**
  return *partida inicial sin disparos*
else
  A $\leftarrow \bigcup_{d' \in D}$ mejoresPartidas$(t-1, d')$
  A $\leftarrow$ conDisparo(d,A)
  $p^* \leftarrow max_{a \in A} \{puntaje(a)\}$
  $A^* \leftarrow \{a \in A / puntaje(a) = p^*\}$
  return A^*

El algoritmo pretende devolver la lista de las mejores partidas posibles que se resuelvan en t turnos y que terminen con el disparo d, sin embargo, no lo logra. Devuelve las mejores partidas posibles al aplicarles el disparo d a cada una de las mejores partidas posibles en t-1 turnos.

En la implementación javascript, los hiperparámetros del algoritmo (es decir, el tablero, la cantidad de lanzaderas, la cantidad de barcos y los puntos de vida de cada uno) también se pasan como argumentos.

2.1.1. Calidad de heurística de Dinámico

Dinamico no conforma un algoritmo de resolución de la situación planteada, sino una heurística. Esto se demuestra por medio del siguiente contraejemplo.

Hay una sola lanzadera.													
V_i	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	t13
10	9	1	1	1	1	1	1	1	1	1	1	1	10
1	1	1	1	1	1	1	1	1	1	1	1	1	10

En la solución mínima, el barco de vida 10 recibe un disparo en el primer turno, aprovechándose de esta forma el gran daño que puede recibir; y el puntaje alcanzado es 5. Sin embargo, debido a que Dinámico es una heurística que aplica un criterio de greedy para determinar la mejor decisión por turno, determina que la mejor decisión para el primer turno es la que lleve a un mejor puntaje, es decir, disparar al barco de salud 1 primero y al de salud 10 después. Así, el puntaje mínimo alcanzado por Dinámico será 11.

2.1.2. Complejidad temporal del algoritmo

El algoritmo Dinamico es memoizado. Como sus únicos parámetros son t y d, su complejidad temporal será lineal respecto de los valores que puede tomar cada uno de sus parámetros. En memoria, se generará una tabla como la siguiente:

turno	disparo posible 1	disparo posible 2	disparo posible 3	...
t_0				
t_1				
t_2				
t_3				
t_4				
t_5				
...				

El costo añadido de computar la solución en cada çasilla"de la tabla es lineal al costo de obtener el puntaje de cada partida alternativa, y por lo tanto a la cantidad de partidas alternativas. Por otro lado, el costo de computar el puntaje de cada partida alternativa es lineal con la cantidad de barcos (ya que es necesario determinar si cada uno de ellos está vivo en la misma). Así, el costo del algoritmo memoizado para resolver $mejoresPartidas(t, d) = O(t * disparosposibles * costodecadacasillero) = O(t*disparosposibles*disparosposibles*costodeobtenerpuntajedeunapartida) = O(t * disparosposibles * disparosposibles * barcos) = O(t * disparosposibles^2 * barcos)$.

2.2. Condiciones para que Dinámico sea óptimo

Dinámico será óptimo cuando la mejor decisión posible en cada turno sea la que más barcos mate en ese turno. Esto sólo puede darse cuando las vulnerabilidades relacionadas a cada barco solamente ascienden a lo largo de la partida, cuando el tablero no es rotatorio", tal como fue el planteo dado, es necesario asegurar también que ninguna partida dura tantos turnos como columnas del tablero. A continuación se demuestra esta afirmación para un tablero infinito, sin repeticiones.

2.3. Planteo matemático de la hipótesis y demostración

2.3.1. Conceptos necesarios para expresar la hipótesis y la prueba

Sea $v(t, b)$ la vulnerabilidad del barco t en el turno b, con $t \geq 0$ y $1 \geq b \geq B$, siendo B la cantidad de barcos. Sea $d(t, b)$ una función tal que

$$d(t,b) = \begin{cases} 1 & \text{si se dispara al barco } b \text{ en el turno } t \\ 0 & \text{en otro caso} \end{cases}$$

Y sea D el conjunto de todos los disparos d que cumplen que:

$$\sum_{b=1}^{B} d(t,b) = L \forall t \geq 0$$

Sea $h_{d,v}(t, b)$ una función que representa la salud del barco b en el turno t, al usar los disparos d y las vulnerabilidades v:

$$h_{d,v}(t,b) = \begin{cases} h(t-1,b) - v(t,b) \cdot d(t,b) & \text{si } t > 0 \\ V_b & \text{si t=0} \end{cases}$$

Siendo V_b la salud inicial de cada barco. Sea $H_{d,v}(b,t)$ una función que indica si el barco b vive en el turno t, usando los disparos d y las vulnerabilidades v.

$$H_{d,v}(t,b) = \begin{cases} 1 & \text{si } h_{d,v}(t,b) > 0 \\ 0 & \text{en caso contrario} \end{cases}$$

Y sea también $H_{d,v}(t)$ tal que:

$$H_{d,v}(t) = \sum_{b=1}^{B} H_{d,v}(t,b)$$

Sea la función $G(d,v,t)$ que indica si en el turno t, el disparo d representa una formado por medio de la estrategia greedy, que consiste en elegir el disparo que destruya más barcos:

$$G(d,v,t) = \begin{cases} 1 & \text{si } h_{d,v}(t) = min_{d' \in D} \{H_{d',v}(t)\} \\ 0 & \text{si no} \end{cases}$$

Sea la función $G*(d,v,t)$ que representa si los disparos de todos los turnos hasta t fueron greedy:

$$G^*(d,v,t) = \begin{cases} G(d,v,t) \cdot G^*(d,v,t-1) & \text{si } T > 0 \\ 1 & \text{si } T = 0 \end{cases}$$

Sea la función $P_{d,v}(t)$ la cantidad de puntos acumulados hastael turno t al usar el disparo d y las vulnerabilidades v:

$$P_{d,v}(t) = \begin{cases} H_{d,v}(t) + P_{d,v}(t-1) & \text{si } t > 0 \\ 0 & \text{si } t = 0 \end{cases}$$

2.3.2. Hipótesis

Para cualquier T natural mayor a 0 se cumple que:
Dado v tal que:

$$v(t+1,b) \geq v(t,b) \forall 0 \leq t \leq T, 1 \leq b \leq B$$

Y dado d tal que:

$$G^*(d,v,T) = 1$$

Entonces:

$$P_{d,v}(T) = min_{d' \in D} \{P_{d',v}(T)\}$$

2.3.3. Demostración: caso base

Para $T = 0$ la hipótesis es verdadera porque $P_{d,v}(T) = 0 \forall d, v$.

2.3.4. Planteo del caso inductivo

Para $T > 0$, tenemos que se cumple para $T-1$ que:

Dado v tal que:

$$v(t+1,b) \geq v(t,b) \forall 0 \leq t \leq T-1, 1 \leq b \leq B$$

Y dado d tal que:

$$G^*(d,v,T-1) = 1$$

Entonces:

$$P_{d,v}(T-1) = min_{d' \in D} \{P_{d',v}(T-1)\}$$

A partir de esa proposición, queremos probar que teniendo:

Dado v tal que:

$$v(t+1,b) \geq v(t,b) \forall 0 \leq t \leq T, 1 \leq b \leq B$$

Y dado d tal que:

$$G^*(d,v,T) = 1$$

Podemos concluir que:

$$P_{d,v}(T) = min_{d' \in D} \{P_{d',v}(T)\}$$

2.3.5. Demostración del caso inductivo

Por la definición de P:

$$P_{d,v}(T) = H_{d,v}(T) + P_{d,v}(T-1)$$

Por la hipótesis inductiva, que afirma que $P_{d,v}(T-1) = min_{d^* \in D} \{P_{d^*,v}(T-1)\}$ podemos definir que:

$$P_{d,v}(T) = H_{d,v}(T) + min_{d^* \in D} \{P_{d^*,v}(T-1)\}$$

Debido a que:

$$G^*(d,v,T) = 1 \Rightarrow G(d,v,T) = 1 \Rightarrow H_{d,v}(T) = min_{d' \in D} \{H_{d',v}(T)\}$$

Podemos concluir, combinando estas últimas dos proposiciones, que:

$$P_{d,v}(T) = min_{d' \in D} \{H_{d',v}(T)\} + min_{d^* \in D} \{P_{d^*,v}(T-1)\}$$

Una manera alternativa de definir $P_{d,v}(t)$ es:

$$P_{d,v}(t) = \sum_{i=1}^{t} H_{d,v}(i)$$

Agregando esta definición a la proposición anterior tenemos que:

$$P_{d,v}(T) = min_{d' \in D} \{H_{d',v}(T)\} + min_{d^* \in D} \left\{ \sum_{i=1}^{T-1} H_{d^*,v}(i) \right\}$$

Debido a que los disparos d en cada uno de los turnos son independientes, y la minimización tiene en cuenta sólo la efectividad de cada turno:

$$P_{d,v}(T) = min_{d' \in D} \{H_{d',v}(T)\} + \sum_{i=1}^{T-1} min_{d^* \in D} \{H_{d^*,v}(i)\}$$

$$P_{d,v}(T) = \sum_{i=1}^{T} min_{d^* \in D} \{H_{d^*,v}(i)\}$$

Teniendo en cuenta la independencia de los disparos en cada uno de los turnos citada anteriormente, consecuencia de la condición impuesta sobre v, que implica que cualquier barco que podría haber sido destruído en cierto turno puede ser destruído en el siguiente, podemos concluir que:

$$P_{d,v}(T) = min_{d'' \in D} \left\{ \sum_{i=1}^{T} H_{d'',v}(i) \right\}$$

Lo cual concluye la demostración.

Algorithm 2: obtenerMejor(p)

Data: Partida inicial a mejorar p
Result: La mejor partida que puede obtener el algoritmo
$p \leftarrow$ Partida a mejorar
if $barcosVivos(p) = 0$ **then**
  return p
else
  $D' \leftarrow \{d \in D /\ d$ impacta barcos vivos de $p\ \}$
  $P \leftarrow$ conDisparo(D',p)
  $p^* \leftarrow$ una de las partidas con el menor puntaje posible de P
  return $obtenerMejor(p^*)$

3. Greedo

3.1. Explicación del algoritmo

En el caso del algoritmo Greedy, para elegir el mejor disparo se utilizó una heurística más efectiva que para la solución dinámica. Greedo evalúa según el mejor puntaje posible a cada paso. Este puntaje se obtiene suponiendo que en todos los turnos siguientes se dispone de tantas lanzaderas como barcos. Así, al intentar maximizar el mejor puntaje posible en vez del puntaje actual, Greedo no prioriza el disparo que más barcos mate sino el disparo que haga que la solución empeore lo menos posible.

Para ponerlo en términos más sencillos, Greedo se pregunta, para cada barco *Cuánto pierdo si no te disparo?* y evaluando eso define qué movimiento hacer a continuación.

3.2. Calidad de heurística de Greedo

A continuación se presenta un caso en el que el algoritmo Greedy falla.
Hay una sola lanzadera.

V_i	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	t13
15	5	5	10	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
10	1	0	10	1	1	1	1	1	1	1	1	1	1

En este caso, el algoritmo puede tomar dos decisiones distintas respecto del primer disparo: o bien dispararle al barco de salud 15 o bien al de salud 10. Ambas decisiones tienen la misma penalización, debido a que la misma está definida por el puntaje que se alcanzaría si se pudiera disparar 2 veces por turno y, al comenzar, ambos barcos serían dstruídos en la columna t3 según esta aproximación. La decisión desemboca en una de dos situaciones:
Disparando inicialmente al barco de vida 15. Hay una sola lanzadera.

V_i	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	t13
5	10	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
10	10	1	1	1	1	1	1	1	1	1	1

A partir de aquí, el algoritmo decide disparar al barco que ahora tiene vida 5, con lo cual, a partir de este turno, se alcanza un puntaje de 10 puntos. La alternativa es disparar al barco de vida 10, lo cual llevaría a un puntaje de 50 puntos.
Disparando inicialmente al barco de vida 10. Hay una sola lanzadera.

V_i	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	t13
10	10	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
9	10	1	1	1	1	1	1	1	1	1	1

A partir de aquí, el algoritmo decide disparar al barco que ahora tiene vida 10 (de lo contrarío obtendría un puntaje de 100), y alcanza una penalización de 9.

Así, vemos que el algoritmo puede alcanzar situaciones subóptimas cuando encuentra que debería destruir más barcos de los que le es posible en la misma columna. El algoritmo Baco, codificado de forma similar a Greedo, postpone estas decisiones, devolviendo a cada paso las mejores soluciones posibles. Se incluyó en el código del trabajo práctico.

3.3. Complejidad temporal del algoritmo Greedo

El algoritmo Greedo ejecuta una única llamada recursiva, que concluye cuando el algoritmo destruye todos los barcos. Así, tal llamada recursiva implica una complejidad temporal de $O(\text{ejecucion} \cdot \text{turnosMaximos})$.

El tiempo de ejecución del algoritmo en sí es lineal con el tiempo de ejecución del algoritmo que calcula el mejor puntaje posible de una partida y la cantidad de configuraciones de disparos posibles: $O(\text{calculo mejor puntaje posible} \cdot \text{disparos posibles} \cdot \text{turnosMaximos})$.

El cálculo del mejor puntaje posible es lineal con la cantidad de barcos y con el tiempo de ejecución del algoritmo que descubre la supervivencia mínima de cada barco:$O(\text{cálculo supervivencia mínima} \cdot \text{barcos} \cdot \text{disparos posibles} \cdot \text{turnosMaximos})$.

El algoritmo que descubre la supervivencia mínima de cada barco tiene una complejidad temporal que depende totalmente del problema: Se inicializa una variable con la salud del barco y se recorren las columnas avanzando y restando la vulnerabilidad correspondiente del acumulador hasta que el mismo se torne 0. Es posible escribir este algoritmo de forma que sea lineal con la cantidad de casilleros de la fila del barco. A continuación se detalla el pseudocódigo para lo mismo:

Algorithm 3: supervivencia(v,h)

$v \leftarrow$ array de vulnerabilidades de la fila
$h \leftarrow$ Salud inicial del barco analizado
$T \leftarrow$ Suma de todos los valores de v
$vueltas \leftarrow \lfloor h/T \rfloor$
$i \leftarrow vueltas \cdot len(v)$
$h \leftarrow h - vueltas \cdot T$
while $h > 0$ **do**
    $i \leftarrow i + 1$
    $h \leftarrow h - v_i$
return i

Por lo tanto, el algoritmo tiene la complejidad temporal:

$$O(\text{columnas del tablero} \cdot \text{barcos} \cdot \text{disparos posibles} \cdot \text{turnosMaximos})$$

3.4. Condiciones para que Greedo sea óptimo

Greedo es óptimo cuando no se da la necesidad de tomar decisiones alternativas, es decir, cuando hay una única solución mejor.

El menor puntaje estimado siempre será mayor al puntaje de la solución real, siempre y cuando la cantidad de barcos vivos sea mayor a la cantidad de lanzaderas. De lo contrario, estos puntajes serán iguales. Así, el juego nunca puede terminar con un puntaje menor al mínimo estimado. El algoritmo Baco descarta unicamente las decisiones que perjudican esta cota inferior. Sin embargo, la misma crece a lo largo de la ejecución del programa, hasta que coinciden la cota inferior y el puntaje encontrado. Debido a esto, es razonable pensar que Baco llega al óptimo. Esta afirmación no se demuestra sino que se deja como razonable.

Greedo, por otro lado, sí descarta decisiones que tienen la menor cota, con lo cual, puede descartar una rama"de decisiones que lleve a la solución óptima, la cual sí alcanzaría Baco. En el código se incluye el algoritmo Dinámico Jr., que aplica la heurística de Greedo al esquema de Dinamico. Es posible que ese algoritmo sí pueda alcanzar el óptimo en tiempo polinómico.

4. Posicionamiento inicial de barcos

La esencia del algoritmo se basa en analizar, por cada uno de los barcos, cuál será el casillero de inicio que le brindará la mayor cantidad de puntos. Esto se logra haciendo avanzar al barco un casillero hacia adelante y restándole en cada paso la cantidad de vida correspondiente, hasta que el barco es destruido. La posición inicial óptima será la que le brinde al barco la mayor cantidad de posiciones avanzadas. La complejidad para el cálculo inicial de cada uno de los barcos es entonces linealmente proporcional a la cantidad de columnas.

Parte II
Parte 2: Sabotaje!

4.1. Análisis preliminar y consideraciones previas

El problema planteado implica un conocimiento de redes de flujo. Una definición con la que trabajaremos es la siguiente: se entiende como red de flujo a un grafo dirigido con aristas pesadas. Llamamos al peso de cada arista la capacidad entre los dos vértices que conecta. Las redes de flujo cumplen las condiciones:

- Existe un único vértice con únicamente aristas salientes a él, sin aristas entrantes, llamado fuente.
- Existe un único vértice con únicamente aristas entrantes a él, sin aristas salientes, llamado sumidero.
- Todos los demás vértices cumplen la condición de que la capcidad entrante, es decir, la sumatoria de todas sus aristas que terminan en este vértice, es igual a la capcidad saliente, la sumatoria de todas las aristas con fuente en este vértice.

La última condición implica una conservación de la capacidad: en la red, todos los nodos intermedios entre la fuente y el sumidero tienen una capacidad neta nula, solo transportan capacidad entre las puntas, sin crear o destruir capacidad adicional.

Las redes de flujo modelan problemas en donde, valga la redundancia, fluye una cantidad apreciable de un bien sobre ciertas vías (los ejes o aristas), en donde cada vértice marca una intersección donde se puede redirigir la dirección del flujo. Las aplicaciones más comunes es en líquidos en cañerías en un problema de hidráulica, corriente eléctrica en un circuito, o el tránsito en un sistema de autovías, entre otros.

Definimos flujo de un eje como la cantidad de ese valor modelado (corriente, líquido, tránsito), que está circulando actualmente entre los nodos determinados. Este valor, por supuesto, debe ser menor a la capacidad total de ese eje, y también debe cumplir la condición de conservación: el flujo entrante a un nodo es igual al saliente.

El cuello de botella, *bottleneck*, de un camino entre fuente y sumidero es la capacidad mínima de todos los ejes del camino. Este valor es el flujo máximo que puede llegar a circular por ese camino, debido a la condición anterior de que el flujo no puede ser mayor a las capacidades de los ejes por el que pasa.

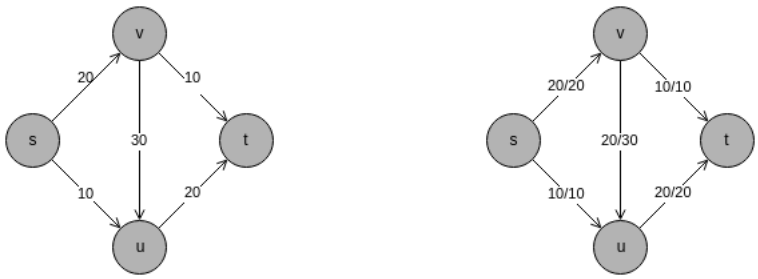

Figura 1: Red de flujo con capacidades totales y con flujo/capacidad total

Leyendo el enunciado se toman varias suposiciones. La más importante es que la red que tenemos que proteger es una red de flujo de manera tal que cumpla con la definición dada anteriormente. Lo segundo es la interpretación de "los saboteadores quieren hacer el máximo daño posible"del enunciado. Se entiende que los saboteadores actúan de manera completamente óptima: Primero sabotearán el eje más importante de la red (a definir más adelante), y luego, el segundo más importante, que será el eje más importante de la red de flujo resultante al tener inoperante

el primer eje. Dicho de otra manera, los saboteadores atacan como segundo eje al que sería más importante en una red de flujo que no contenga al primer eje saboteado.

No está de más aclarar también que la solución implementada es óptima y termina con el orden determinado únicamente cuando las capacidades de los ejes son números enteros.

4.2. Algoritmo propuesto

El problema se resume en determinar las dos aristas más importantes de la red. Es sencillo de ver que si encontramos el flujo máximo que puede pasar por nuestra red, las aristas más importantes serán las que mayor flujo pasen por ellas. El algoritmo a proponer se resume a este concepto, hallamos el flujo máximo de la red, y luego iteramos sobre todas las aristas de la red y detectamos las de mayor flujo. El flujo máximo se calcula por el algoritmo de Ford Fulkerson, con la búsqueda de caminos fuente-sumidero usando DFS.

El algoritmo implementado es óptimo gracias a la garantía de que el flujo máximo encontrado es óptimo, Ford Fulkerson determina la mayor cantidad de flujo pasable por la red. Sabiendo que los flujos son óptimos, el resultado de iterar sobre ellos devolverá las aristas más importantes también será óptimo.

Algorithm 4: Pseudocódigo del algoritmo propuesto

```
Data: G: red de flujo a determinar vulnerabilidades, n: cantidad de ejes a proteger
Result: vul: lista de ejes a proteger
R ← FordFulkerson(G); // Grafo residual
vul ← [];
while vul.length < n do
    foreach e ∈ E(R) do
        if vul.length < n then
            vul.append(e);
        else if e.flow > min(vul) then
            vul.remove(min(vul));
            vul.append(e);
return vul;
```

La complejidad del algoritmo está acotada por Ford Fulkerson, de orden $O(mC)$, siendo m la cantidad de vértices de la red, y C la capacidad saliente de la fuente. El hallado de los ejes más importantes de la red es una búsqueda lineal sobre las mismas, $O(n) = O(2m) = O(m)$. Se podría ver como una reducción sobre Ford Fulkerson: se está usando el algoritmo de flujo máximo como una caja negra, y simplemente iterando sobre sus resultados (transformación de valores de salida) para determinar la solución a nuestro problema.

La complejidad es calculada de esta manera gracias a la suposición de que las capacidades son valores enteros. El algoritmo de Ford Fulkerson utiliza esta suposición para dar su propia cota de complejidad. De ser las capacidades un valor fraccionario, ni siquiera se podría garantizar la terminación del algoritmo, debido a que en el peor caso Ford Fulkerson podría llegar a aumentar su flujo de a cantidades arbitrariamente pequeñas hasta llegar al máximo.

Analizando los flujos máximos antes y después del sabotaje podemos apreciar que el flujo no necesariamente difiere en la cantidad exacta de los flujos de las aristas saboteadas. Si borramos esas aristas, el flujo puede ser redirigido y tomar una dirección con capacidad disponible que antes no hacía porque simplemente no era conveniente.

4.3. Extensión con varias fuentes y sumideros

Se plantea la situación del mismo problema, pero ahora con una red que tenga varios nodos fuente y sumidero. Se puede ver de manera sencilla que una red con varias fuentes y sumideros es una composición de varias redes individuales de una única fuente y único sumidero.

Comencemos con una red de 1 fuente, 1 sumidero. Esta red ya fue definida anteriormente, en particular restaltamos la propiedad de conservación del flujo en cada nodo interior. Agregar una fuente y un sumidero nuevo implicaría agregar esos dos nodos, más los ejes que transporten flujos entre ellos, agregando también nodos adicionales si se quiere, pero también podrían utilizarse los mismos nodos de la red original. Agregar un eje acá puede implicar simplemente aumentar la capacidad máxima del eje entre dos nodos, no necesariamente haya que agregar una nueva arista al grafo que representa la red.

De esta manera se puede ver que para cualquier fuente de flujo existe al menos un camino desde una fuente a algún sumidero. La idea principal para resolver este caso, entonces, es descomponer a la red de múltiples fuentes - sumideros en caminos o subgrafos en donde haya una correspondencia 1:1, y luego aplicar el algoritmo de flujo máximo sobre cada uno, y sumar los resultados. Esta suma de flujos es el flujo máximo de la red original propuesta. Desde allí basta con detectar los dos ejes de mayor flujo.

La descomposición puede resultar difícil de plantear como un algoritmo pero basta modificar la Implementación actual que busca caminos entre la única fuente y el único sumidero a buscar caminos desde *cualquier* fuente hasta *cualquier* sumidero, que es equivalente a la descomposición, por lo menos para el propósito que se estaría usando en este caso. El orden del algoritmo no es modificado, sigue estando acotado por la cantidad de nodos, y la capacidad mínima de la red, debido al uso de Ford-Fulkerson internamente.

4.4. Implementación

Se implementó el algoritmo propuesto (junto con Ford Fulkerson) en Nodejs. Para correr el código, asegurarse de contar con una versión de Node >= 8.0, correr *npm i*, y luego *npm start*.

Parte III
Anexo: Correcciones de Sabotaje!

La suposición que se toma es que KAOS quiere disminuir el flujo lo más posible.

Algorithm 5: Pseudocódigo del algoritmo propuesto

Data: G: red de flujo a determinar vulnerabilidades, n: cantidad de ejes a proteger
Result: lista de ejes a proteger

$R \leftarrow$ FordFulkerson(G) // Grafo residual
//las pertenecientes al corte mínimo
$aristasCorte \leftarrow$ obtenerAristasCorte(R)
$primeraVictima \leftarrow e \in aristasCorte$ / $capacidad_e = max_{e \in aristasCorte}$ { $capacidad_e$ }
$primerFlujo \leftarrow$ flujo(R)− $capacidad_{primeraVictima}$
$ataques \leftarrow \{(primerFlujo, atacada)\}$
$victima \leftarrow primeraVictima$

$flujoNuevo \leftarrow 0$
while $flujoNuevo <$ *max*{ *f* / *(f,v)* $\in$ *ataques* } **do**
  $R \leftarrow R$.cambiarCapacidad($victima$,∞)
  //cambiarCapacidad continúa la ejecución de Ford Fulkerson
  aristasCorte $\leftarrow$ obtenerAristasCorte(R)
  $victima \leftarrow e \in aristasCorte$ / $capacidad_e = max_{e \in aristasCorte}$ { $capacidad_e$ }
  $flujoNuevo \leftarrow$ flujo(R) − $capacidad_{victima}$
  $ataques \leftarrow ataques \cup$ { (flujo,victima) }
  if $|ataques| > 2$ **then**
    ataques $\leftarrow$ los dos (f,v) $\in$ *ataques* con menor f

return los dos (f,v) $\in$ *ataques* con menor f

4.5. Explicacion del algoritmo propuesto

La fundamentación principal del algoritmo es el llamado *max flow-min cut theorem*, enunciado en "Algorithm Design" de Kleinberg y Tardos como la afirmación 7.9. Este teorema establece que el flujo máximo es igual a la suma de las capacidades de las aristas del corte mínimo. Así, removiendo una arista de tal corte nos aseguramos que la capacidad del corte mínimo disminuye, y por lo tanto también el flujo máximo de la red.
Se elige remover, en cada caso, la arista del corte mínimo con mayor capacidad. Sin embargo, remover esta arista no asegura que el flujo decaiga lo más posible, ya que podría existir un corte que -sin ser el mínimo- pueda tener todavía menor capacidad al remover la arista de mayor capacidad que posee. El algoritmo construído intenta apalear esta situación averiguando el menor corte mínimo que sea mayor al analizado: imposibilita que el corte analizado sea mínimo asignando una capacidad infinita a la arista atacada, y luego continúa la ejecución de Ford-Fulkerson, para luego encontrar el siguiente corte mínimo. Se especula que el siguiente corte mínimo no sea demasiado mayor al encontrado antes, y se vuelve a proceder como antes, con la esperanza de que la nueva remoción sea mejor -o no demasiado peor- respecto de la anterior.
Se procede de esta manera de forma de obtener al menos dos soluciones. Sin embargo, si se encuentra que la siguiente solución es mejor, se continúan procesando más soluciones, con la esperanza de mejorar más a cada iteración.

4.6. Naturaleza de heurística del algoritmo construído

El algoritmo construído no descubre las aristas que causan la mayor disminución del flujo, como muestra el siguiente ejemplo:

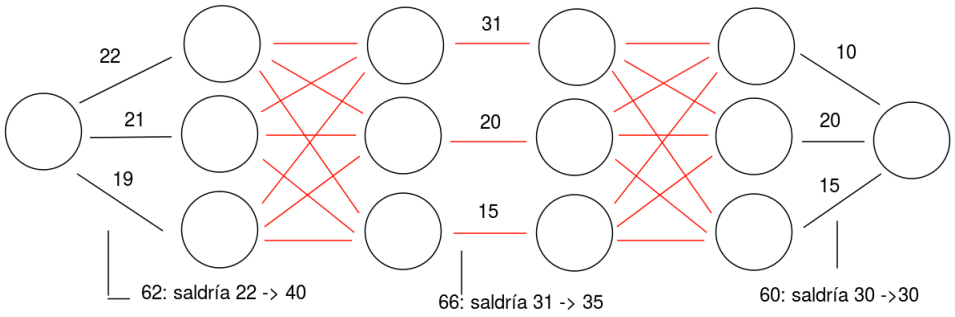

Figura 2: Contraejemplo

En este caso, el algoritmo primero eliminaría el vértice de valor 30, y luego, en vez a de priorizar el corte de capacidad 66, que llevaría a un flujo máximo de 35 al remover la arista correspondiente, prioriza el corte de capacidad 62, que lleva a uno de 40.

4.7. Complejidad de la ejecución del algoritmo

En el caso de que el algoritmo encuentre cada vez mejores aristas para remover, el ciclo terminará con la solución de flujo infinito, es decir, hará infinitas las capacidades de todas las aristas. Como el infinito es mayor que cualquier mínimo corte que se encuentre, esta es la última situación analizada posible. En este caso, itera tantas veces como aristas tenga el grafo.

Kleinberg-Tardos demuestran que en cada ciclo del algoritmo Ford Fulkerson, el flujo aumenta, y que aumenta como mínimo en una unidad (afirmaciones 7.2 y 7.3), así, establecen que el ciclo del algoritmo en el que se aumentan los flujos se ejecutará al menos C veces, siendo C una cota del flujo máximo. Kleinberg y Tardos acotan el flujo máximo con C igual a la suma de las capacidades salientes de la fuente.
Aquí pretendemos extender la cota temporal del algoritmo Ford-Fulkerson a redes de flujo con grafos con capacidades infinitas. En este caso, el flujo máximo o bien es infinito (en el caso de que exista un augmenting path conformado de aristas que sólo tienen capacidad infinita), o bien es un entero (si no existe tal augmenting path). Si el flujo máximo es un entero, debe ser menor o igual a la suma de todas las capacidades no infinitas de las aristas del grafo, ya que el corte mínimo debe estar conformado por aristas de capacidad entera (como mucho, todas). Si el flujo es infinito, es natural pensar que se encontrará en como mucho un path augmentation más que cualquier flujo entero. Por lo tanto, el algoritmo Ford-Fulkerson con capacidades infinitas tomará como mucho $K + 1$ iteraciones, siendo K la suma de las capacidades no infinitas.
Ahora pretendemos acotar temporalmente el algoritmo propuesto. Una primera cota es $O(m^{2K})$, es decir, la ejecución de Ford-Fulkerson m veces. Sin embargo, como el algoritmo no se inicia de nuevo, sino que se continúa en cada ciclo, la cantidad de veces que se ejecutarán path augmentations será $m + K$, con lo cual tendremos una cota de $O(m * (m + K))$. Como $K \geq m$ (debido a que partimos de un grafo con capacidades finitas y enteras), tendremos la cota $O(m * K)$.

Afirmación La cantidad de veces que se ejecutan *path augmentations* en nuestro algoritmo será como mucho $m + K + 1$.

Demostración En cada ejecución del ciclo exterior de nuestro algoritmo, el flujo no disminuye, es decir, o bien aumenta, o bien permanece constante. Sea el aumento de flujo en el ciclo i δ_i, siendo $\delta_i \geq 0$, con δ_0 el flujo inicial. En nuestro algoritmo no volvemos a empezar la ejecución

de Ford-Fulkerson para cada arista removida del grafo sino que se continúa la ejecución previa. Así, a partir de las afirmaciones 7.2 y 7.3 de Kleinberg y Tardos podemos concluir que en cada ejecución de Ford-Fulkerson se ejecutan, como mucho, δ_i ciclos, es decir, se aumentan δ_i caminos. En algunas iteraciones del ciclo exterior podría no obtenerse ningún incremento respecto del flujo anterior, con lo cual la cantidad de veces que se buscan augmenting paths en la iteración i es $\max\{1,\delta_i\}$. Sea Δ_i la suma de δ_j con j desde 0 hasta i. Así, Δ_i es el flujo máximo obtenido en la iteración i. Así, la suma de $\max\{1,\delta_j\}$ desde 0 hasta $i \leq \Delta_i + i \ \forall i$. Sea M la iteración final del algoritmo en el peor caso. En ella, se alcanzará un flujo infinito. Así, en la iteración $M-1$ se tendrá el último flujo finito. Tal como discutido previamente, $\Delta_{M-1} \leq \sum c_e \ \forall e \in E / c_e \neq \infty$, teniendo c_e los valores correspondientes a la iteración $M-1$. Como en la iteración $M-1$ se habrá cambiado el valor de $M-1$ capacidades a infinito, entonces la suma de las capacidades finitas en el paso $M-1$ será menor que la suma de las capacidades de la red inicial K. Por lo tanto, $\Delta_{M-1} \leq K$. En conclusión, la cantidad de veces que se buscan augmenting paths hasta la ejecución M-1 es la suma de $\max\{1,\delta_j\}$ desde 0 hasta $M-1 \leq \Delta_{M-1} + M - 1 \leq K + M - 1$. Como el algoritmo se detiene, en el peor caso, cuando las m aristas tuvieron su capacidad cambiada a infinito, tenemos que $M \leq m$, con lo cual $K + M - 1 \leq K + m - 1$. Además, en el paso M se busca el último augmenting path: aquel que va de s a t, con capacidad infinita en todas sus aristas. Así, se buscarán, como mucho $k + m - 1 + 1$ augmneting paths, llegando así al máximo de $k + m$ augmenting paths.

Parte IV
Codigo fuente

Listing 1: Battleships: barco

```
function barco(vida,posicion=0){
        let o={}
        o.vive=function(){
                return vida >0;
        }
        o.conDanio=function(danio){
                return barco(vida-danio(posicion),posicion)
        }
        o.obtenerSalud=function(){
                return vida
        }
        o.obtenerPosicion=function(){
                return posicion
        }
        o.desplazado=function(dx){
                return barco(vida,posicion+dx)
        }
        o.obtenerDesplazamiento=function(){
                return posicion
        }
        o.supervivenciaMinima=function(danio){
                let desplazado=o.conDanio(danio).desplazado(1)
                if(desplazado.vive()){
                        return 1 + desplazado.supervivenciaMinima(danio)
                }else{
                        return 0
                }
        }
        return o
}
module.exports=barco
```

Listing 2: Battleships: dinamico.js

```
const disparo = require("./disparo")
const partida = require("./partida")
//caso general
function mejoresPartidasGeneral(turno,disparoSiguiente,lanzaderas,
    vulnerabilidades,barcos){
    //obtener todas las alternativas posibles
    let alternativas =
    disparo.posibles(lanzaderas,barcos.length)
    .map(d=>
        mejoresPartidasMemoizada(turno-1,d,lanzaderas,vulnerabilidades,barcos)
    )
    .reduce(
        (x,y)=>
            x.concat(y),
        []
    )
    //evaluar puntaje al agregar el disparo
    .map((a)=>
            a.conDanios(vulnerabilidades(turno),disparoSiguiente)
    )

    //quedarme con las alternativas no superadas
    let mejorPuntaje=0
    let mejores=[]
    for (let a of alternativas){
        if(mejores.length==0 || mejorPuntaje>a.obtenerPuntaje()){
            mejores=[a]
            mejorPuntaje=a.obtenerPuntaje()
        }else if(mejorPuntaje==a.obtenerPuntaje()){
            mejores.push(a)
        }
    }
    return mejores
}

```

```
//caso base
function mejoresPartidasConBase(turno,disparo,lanzaderas,
    vulnerabilidades,barcos){
    if(turno==0){
        return [partida(barcos,0).conDanios(vulnerabilidades(turno),disparo)]
    }else{
        return mejoresPartidasGeneral(turno,disparo,lanzaderas,
            vulnerabilidades,barcos)
    }
}

//versión memoizada
var map=new Map()
function mejoresPartidasMemoizada(turno,disparo,lanzaderas,
    vulnerabilidades,barcos){
    let llamada=[turno,disparo.descripcion(),lanzaderas,vulnerabilidades.
      descripcion(),barcos.map((b)=>b.obtenerSalud())].toString()
    if(map.has(llamada)){
        return map.get(llamada)
    }else{
        let resultado=mejoresPartidasConBase(turno,disparo,lanzaderas,
            vulnerabilidades,barcos)
        map.set(llamada,resultado)
        return resultado
    }

}


//interfaz
function dinamico(lanzaderas,vulnerabilidades,barcos) {
    let t=0;
    let partidas=[]
    let partidasAnteriores=[]
    let turnos=0
    while (partidas.length==0 ||
        !partidas.some((p)=>p.obtenerBarcosVivos()==0)){
        partidas=mejoresPartidasMemoizada(turnos,disparo.vacio(barcos.length),
            lanzaderas,vulnerabilidades,barcos)
            turnos+=1
    }

    return partidas.filter((p)=>p.obtenerBarcosVivos()==0)

}


module.exports=dinamico
```

Listing 3: Battleships: disparo.js

```
function disparo(arrayDeDisparos){
    let o={}
    function conInicial(v){
        return disparo([v].concat(arrayDeDisparos))
    }
    o.conDisparoInicial=function(){
        return conInicial(true)
    }
    o.sinDisparoInicial=function(){
        return conInicial(false)
    }
    o.dispara=function(v){
        return arrayDeDisparos[v]
    }
    o.descripcion=function(){
        return arrayDeDisparos.concat([])
    }
    return o;
}

function disparoIgual(barcos,disparan){
    let disparos=Array.apply(null,Array(barcos)).map((d,i)=>disparan)

        return disparo(disparos)
}

```

```
function disparoNulo(){
        return disparo([])
}

function disparoVacio(barcos){
        return disparoIgual(barcos,false)
}

function disparoCompleto(barcos){
        return disparoIgual(barcos,true)
}

function disparosPosibles(lanzaderas,barcos){
    if(barcos==0){
        return [disparoNulo()]
    }
    if(lanzaderas==0){
        return [disparoVacio(barcos)]
    }
    if(lanzaderas==barcos){
        return [disparoCompleto(barcos)]
    }

    let disparosDisparoAhora=disparosPosibles(lanzaderas-1,barcos-1)
    disparosDisparoAhora=disparosDisparoAhora.map((d)=>d.conDisparoInicial())

    let disparosNoDisparoAhora=disparosPosibles(lanzaderas,barcos-1)
    disparosNoDisparoAhora=disparosNoDisparoAhora.map((d)=>d.
        sinDisparoInicial())

    return disparosDisparoAhora.concat(disparosNoDisparoAhora)
}


module.exports.nulo=disparoNulo
module.exports.vacio=disparoVacio
module.exports.completo=disparoCompleto
module.exports.posibles=disparosPosibles
```

Listing 4: Battleships: greedo.js

```
const partida=require("./partida")
const disparo=require("./disparo")

/**
 * Construye la solución hacia adelante,
 * minimizando una función de pérdida a cada paso
 * @param {number} lanzaderas
 * @param {function} vulnerabilidades
 * @param {array} barcos
 */
function greedo(lanzaderas,vulnerabilidades,barcos){
    let inicial=partida(barcos,0)
    return obtenerMejorDesde(inicial,0,vulnerabilidades,
        disparo.posibles(lanzaderas,barcos.length))
}


function obtenerMejorDesde(partida,turno,vulnerabilidades,disparosPosibles){

    if(partida.obtenerBarcosVivos()==0){
        return partida
    }

    let mejorPartida=disparosPosibles
    .filter((d)=>partida.impactaVivos(d))
    .map((d)=>partida.conDanios(vulnerabilidades(turno),d))
    .map((p)=>{
        return {
            puntaje:p.mejorPuntajePosible(vulnerabilidades(turno)),
            partida:p
        }
    })
    .reduce((a,b)=>{
        if(a.puntaje <= b.puntaje){
            return a
        }else{
```

```
            return b
        }
    }).partida

    return obtenerMejorDesde(mejorPartida,turno+1,vulnerabilidades,
        disparosPosibles)
}

module.exports=greedo
```

Listing 5: Battleships: main.js

```
const FILE_ARGV = 2;
const STRATEGY_ARGV = 3;
const CANTIDAD_LANZADERAS = 4;
const POSITION_SHIPS = 5;
const readline = require('readline');
const fs = require('fs');
const greedo = require("./greedo.js")
const dinamico = require("./dinamico.js")
const posicionador = require("./posicionadorGreedy.js");
const barco = require("./barco.js");
const vulnerabilidades = require("./vulnerabilidades.js");
const math = require('mathjs')

/* ships' points */
var ships = [];
var shipsPositions = [];
var numOfShips = 0;
var board = [];
var output;

main();
function main(){
  if(process.argv.length <= 2){
    console.log("Por favor, indicar como argumento: \n"
        + "\tarchivo de informacion de juego\n");
    return;
  }

        readFile(process.argv[FILE_ARGV]);

  if(process.argv[POSITION_SHIPS] == "true"){
    posicionadosGreedy(board, ships);
  }

  if(process.argv[STRATEGY_ARGV] == "greedo"){
    console.log("greedo chosen");
    output = greedo(process.argv[CANTIDAD_LANZADERAS],
        vulnerabilidades(board),ships);
  } else if (process.argv[STRATEGY_ARGV] == "dinamico"){
    console.log("dinamico chosen");
    output = dinamico(process.argv[CANTIDAD_LANZADERAS],
        vulnerabilidades(board),ships);
  }
  printOutput(output.obtenerHistorial());
}

function readFile(path){
  let data = fs.readFileSync(path, 'utf-8');

  let lines = data.split('\n');
  lines = lines.filter((line) => line.length);

  lines.forEach((line) => parseInputLine(line));
  board = math.transpose(board);
}

function parseInputLine(line, grafo) {
  let values = line.split(' ');
  var ship = new barco(values.shift());
  ships.push(ship);
  board.push(values);
}

function printOutput(out){
  for(i = 0; i < out.length; ++i){
```

```
    turno = i + 1;
    console.log("Turno: " + turno);
    console.log("salud barcos: " + out[i].ba);
    console.log("posicion barcos: " + out[i].pos);
    console.log("puntaje: " + out[i].pun);
    console.log("descripcion disparo: " + out[i].disp + "\n");
  }
}
```

Listing 6: Battleships: posicionadorGreedy.js

```
function posicionadosGreedy(vulnerabilidades,barcos){
    return barcos.map((b,i)=>{
        let desp = mejorDesplazamientoGreedy(b,vulnerabilidades(0)(i))
        return b.desplazado(desp)
    })
}

function mejorDesplazamientoGreedy(barco,danios){
    let mejorSupervivencia=0
    let mejorPosicion=-1
    for(let i=0;i<danios.tamanio();i+=1){
        let supervivenciaI=barco.desplazado(i).supervivenciaMinima(danios)
        if(supervivenciaI>mejorSupervivencia || mejorPosicion==-1){
            mejorSupervivencia=supervivenciaI
            mejorPosicion=i
        }
    }

    return mejorPosicion
}

module.exports=posicionadosGreedy
```

Listing 7: Battleships: vulnerabilidades.js

```
function ocultadorDeVulnerabilidades(arrayDeArrays){
  let fDeTurno = function deTurno(turno){
    return function deBarco(barco){
      let fConDesplazamiento= function conDesplazamiento(desplazamiento){
        let filaElegida=arrayDeArrays[(turno+desplazamiento) %
          arrayDeArrays.length]
          return filaElegida[barco % filaElegida.length]
      }
      fConDesplazamiento.tamanio=function(){
        return arrayDeArrays.length
      }
      return fConDesplazamiento
    }
  }
  fDeTurno.descripcion=function(){
    return arrayDeArrays
  }
  return fDeTurno
}

module.exports=ocultadorDeVulnerabilidades
```

Listing 8: Sabotaje!: grafo.js

```
class Grafo {

    constructor() {
        this.aristas = {};
    }

    agregarArista(nodo1, nodo2, peso=1) {
        if (!this.aristas[nodo1]) {
            this.aristas[nodo1] = [];
        }

        if (!this.aristas[nodo2]) {
            this.aristas[nodo2] = [];
        }

```

```
16          this.aristas[nodo1].push({'destino': nodo2, 'peso': peso});
17      }
18
19      borrarArista(desde, hasta) {
20          if (!this.aristas.hasOwnProperty(desde)) {
21              throw EvalError("No existe nodo " + desde);
22          }
23
24          for (let i = 0; i < this.aristas[desde].length; i++) {
25              const arista = this.aristas[desde][i];
26              if (arista['destino'] === hasta) {
27                  this.aristas[desde].splice(i, 1);
28                  return;
29              }
30          }
31
32          throw EvalError("no existe arista " + desde + "-" + hasta);
33      }
34
35      adyacentes(nodo) {
36          if (this.aristas[nodo]) {
37              return this.aristas[nodo].map((nodo) => nodo.destino);
38          }
39          return [];
40      }
41
42      camino(desde, hasta) {
43          let padres = {}
44
45          this._fill_parents(desde, padres);
46
47          // Si no hay adyacentes, no hay camino
48          if (Object.keys(padres).length === 0) {
49              return [];
50          }
51
52          // No llegamos nunca hasta el nodo final, no hay camino
53          if (padres[hasta] === undefined) {
54              return [];
55          }
56          padres[desde] = null;
57          let result = [hasta];
58          let actual = hasta;
59          while (result[0] !== desde) {
60              actual = padres[actual];
61              result.unshift(actual)
62          }
63          return result;
64      }
65
66      _fill_parents(nodo, padres, cut=[]) {
67          let adyacentes = this.adyacentes(nodo);
68          for(let i = 0; i < adyacentes.length; i++) {
69              const element = adyacentes[i];
70              if (this.peso(nodo, element) === 0) {
71                  cut.push({desde: nodo, hasta: element});
72                  continue;
73              }
74
75              if (!padres.hasOwnProperty(element)) {
76                  padres[element] = nodo;
77                  this._fill_parents(element, padres, cut);
78              }
79          }
80      }
81
82      getCut(startNode) {
83          // roto roto roto muy roto todo fixme
84          let padres = {};
85          padres[startNode] = null;
86          let cut = [];
87          this._inCut(startNode, padres, cut);
88          return cut;
89      }
90
91      _inCut(nodo, padres, cut) {
92          let adyacentes = this.adyacentes(nodo);
93          for(let i = 0; i < adyacentes.length; i++) {
```

```
            const element = adyacentes[i];
            if (!padres.hasOwnProperty(element)) {
                if (this.peso(nodo, element) > 0) {
                    padres[element] = nodo;
                    this._inCut(element, padres, cut);
                } else {
                    cut.push({desde: nodo , hasta: element});
                }
            }
        }
    }

    peso(desde, hasta) {
        if (!this.aristas.hasOwnProperty(desde)) {
            throw EvalError("No existe nodo " + desde);
        }

        for(const arista of this.aristas[desde]) {
            if (arista['destino'] === hasta) {
                return arista['peso'];
            }
        }
        throw EvalError("no existe arista " + desde + "-" + hasta);
    }

    existeArista(desde, hasta) {
        if (!this.aristas.hasOwnProperty(desde)) {
            return false;
        }

        for(const arista of this.aristas[desde]) {
            if (arista['destino'] === hasta) {
                return true;
            }
        }
        return false;
    }
    actualizarPeso(desde, hasta, peso) {
        if (!this.aristas.hasOwnProperty(desde)) {
            throw EvalError("No existe nodo " + desde);
        }

        for(const arista of this.aristas[desde]) {
            if (arista['destino'] === hasta) {
                arista['peso'] = peso;
                return;
            }
        }

        throw EvalError("No existe arista " + desde + "-" + hasta);
    }
}

const _Grafo = Grafo;
export { _Grafo as Grafo };
```

Listing 9: Sabotaje!: network.js

```
import { Grafo } from "./grafo";


function bottleneck(grafo, camino) {
    let bottleneck = Number.MAX_SAFE_INTEGER;
    for (let i = 0; i < camino.length - 1; i++) {
        const current = camino[i];
        const next = camino[i + 1];
        const peso = grafo.peso(current, next);
        bottleneck = peso < bottleneck ? peso : bottleneck;
    }
    return bottleneck;
}

function init_residual_graph(grafo) {
    let residual = new Grafo();
    for(let nodo in grafo.aristas) {
        for(let arista of grafo.aristas[nodo]) {
            residual.agregarArista(nodo, arista['destino'], arista['peso']);
```

```
            if (grafo.existeArista(arista['destino'], nodo)) {
                continue;
            }

            residual.agregarArista(arista['destino'], nodo, 0, true);
        }
    }
    return residual;
}

function update_residual_graph(grafo, camino) {
    let max_flow = bottleneck(grafo, camino);
    for (let i = 0; i < camino.length - 1; i++) {
        const current = camino[i];
        const next = camino[i + 1];
        const peso = grafo.peso(current, next);
        if(peso < max_flow) {
            throw EvalError("Flujo a reducir es mayor que el actual");
        }

        grafo.actualizarPeso(current, next, peso - max_flow);
        const peso_inverso = grafo.peso(next, current);
        grafo.actualizarPeso(next, current, peso_inverso + max_flow);
    }
}

function maxFlow(grafo, inicial, final) {
    let residual = init_residual_graph(grafo);
    let camino = residual.camino(inicial, final);
    let flow = 0;
    while (camino.length) {
        flow += bottleneck(residual, camino);
        update_residual_graph(residual, camino);
        camino = residual.camino(inicial, final);
    }
    return {'flow': flow, 'grafo_residual': residual};
}

function getMaxUsedCapacities(grafo, residual, cantidad) {
  let max_used_capacities = [];
  while(max_used_capacities.length < cantidad) {
    for (let nodo in grafo.aristas) {
      for (let arista of grafo.aristas[nodo]) {
        let used_capacity = residual.peso(arista['destino'], nodo);
        let remaining_capacity = residual.peso(nodo, arista['destino']);

        const min =
          Math.min(...max_used_capacities.map((x) => x['capacidad_usada']));
        if (max_used_capacities.length < cantidad || used_capacity > min) {
          if (max_used_capacities.length == cantidad) {
            max_used_capacities =  max_used_capacities.filter((x) =>
                x['capacidad_usada'] !== min);
          }
          max_used_capacities.push({
            'capacidad': used_capacity + remaining_capacity, '
              capacidad_usada': used_capacity,
            'capacidad_restante': remaining_capacity,
            'fuente': nodo,
            'destino': arista['destino']
          });
        }
      }
    }
  }
  return max_used_capacities;
}

export {bottleneck, init_residual_graph, update_residual_graph, maxFlow,
  getMaxUsedCapacities};
```

Listing 10: Sabotaje!: read_input.js

```
import { readFileSync } from "fs";
import { Grafo } from "./grafo";

function readInputFile(path) {
    let data = readFileSync(path, 'utf-8');
```

```
6
7       let lines = data.split('\n');
8       lines = lines.filter((line) => line.length);
9
10      let grafo = new Grafo();
11      lines.forEach((line) => parseInputLine(line, grafo));
12      return grafo;
13  }
14
15  function parseInputLine(line, grafo) {
16      let values = line.split(' ');
17      if (values.length != 3) {
18          throw EvalError(line);
19      }
20      let nodo1 = values[0];
21      let nodo2 = values[1];
22      let peso = parseInt(values[2]);
23      grafo.agregarArista(nodo1, nodo2, peso);
24  }
25  export { readInputFile };
```

Listing 11: Sabotaje!: vulnerabilidades.js

```
1   import { readInputFile } from "./read_input";
2   import { maxFlow, getMaxUsedCapacities } from "./network";
3
4   const FUENTE = '0';
5   const SUMIDERO = '1';
6   const CANTIDAD_EJES_A_VIGILAR = 2;
7   const INFINITO = 999999;
8
9   function encontrar_vulnerabilidades() {
10    let g = readInputFile("redsecreta.map");
11
12    let max_flow = maxFlow(g, FUENTE, SUMIDERO);
13    if (!max_flow['flow']) {
14      console.log("No hay flujo posible en el grafo");
15      return;
16    }
17    const residual = max_flow['grafo_residual'];
18
19    console.log("Flujo máximo antes de sabotaje: ", max_flow['flow']);
20
21    let max_cap = getMaxUsedCapacities(g, residual, CANTIDAD_EJES_A_VIGILAR);
22    const str = max_cap.
23      map((x) => [x['fuente'] + " -> " + x['destino']] +
24          ' (' + x['capacidad_usada'] + ')').
25          reduce((x, y) => x + ", " + y);
26
27    console.log("Aristas más relevantes", str);
28
29    max_cap.forEach((x) => g.borrarArista(x['fuente'], x['destino']));
30
31    max_flow = maxFlow(g, FUENTE, SUMIDERO);
32    console.log("Flujo máximo después del sabotaje: ", max_flow['flow']);
33  }
34
35  function vulnerabilidades2() {
36      let g = readInputFile("redsecreta.map");
37
38      let max_flow = maxFlow(g, FUENTE, SUMIDERO);
39      const originalFlow = max_flow['flow']
40      if (!originalFlow) {
41          console.log("No hay flujo posible en el grafo");
42          return;
43      }
44
45      let residual = max_flow['grafo_residual'];
46      let victima = getVictima(residual);
47      let ataques = [{flujo: max_flow['flow'] - victima.cap, atacada: victima}];
48      let firstLoop = true;
49      while (true) {
50          g.actualizarPeso(victima.desde, victima.hasta, INFINITO);
51          max_flow = maxFlow(g, FUENTE, SUMIDERO);
52          residual = max_flow['grafo_residual'];
53          victima = getVictima(residual);
54          let nuevoFlujo = max_flow['flow'] - victima.cap;
```

```
        ataques.push({flujo: nuevoFlujo, atacada: victima});

        if (ataques.length > 2) {
            let minFlujo = ataques
                .map((x) => x.flujo)
                .reduce((x, y) => x < y ? x : y);

            for (let i = 0; i < ataques.length; i++) {
                if (ataques[i].flujo === minFlujo) {
                    ataques.splice(i, 1);
                    break;
                }
            }
        }
        let maxFlujo = ataques
            .map((x) => x.flujo)
            .reduce((x, y) => x > y ? x : y);

        if (!firstLoop && nuevoFlujo >= maxFlujo) {
            break;
        }

        firstLoop = false;
    }

    console.log(ataques);
    return ataques;
}

function getVictima(residual, ataques) {
  const cut = residual.getCut(FUENTE);
  if (!cut.length) {
    return;
  }
  let maxArista = cut
    .map((x) => {
      x.cap = residual.peso(x.desde, x.hasta) +
        residual.peso(x.hasta, x.desde);
      return x;
    })
  .filter((x) => {  // Filtro ataques que ya estén incluidos anteriormente
    for(let ataque in ataques) {
      if (ataque.desde == x.desde && ataque.hasta == x.hasta) {
        return false;
      }
    }
    if (x.cap === INFINITO) {
      return false;
    }
    return true;
  })
  .reduce((x, y) =>  (x.cap > y.cap) ? x : y);

  return maxArista;
}

vulnerabilidades2();
```