Teoría de Algoritmos I - [75.29/95.06]

TRABAJO PRÁCTICO Nº 3

Grupo MinMax

Alumno	Padrón	Mail
del Mazo, Federico	100029	delmazofederico@gmail.com
Djeordjian, Esteban Pedro	100701	edjeordjian@gmail.com
Kristal, Juan Ignacio	99779	kristaljuanignacio@gmail.com
Raveszani, Nicole	101031	nraveszani@gmail.com

Índice

1. Batalla Naval **2**
1.1. Estrategias 2
1.1.1. Greedo 2
1.2. Juego 5
1.3. Condiciones para que la Estrategia Greedy Sea Óptima 6
1.4. Interfaz Gráfica 6
1.5. Estrategia para el Jugador A 6

2. Sabotage **7**
2.1. Algoritmo de Protección 7
2.2. Complejidad del Algoritmo 9
2.3. Flujo Máximo en la Red 10
2.4. Solución con Varias Fuentes y Sumideros 10

3. Anexo: Procedimientos para la Ejecución y Compilación **11**
3.1. Compilación 11
3.2. Ejecución 11
3.2.1 Batalla naval 11
3.2.1. Sabotage 11

1. Batalla Naval

1.1. Estrategias

La estrategia greedy consiste en hallar la solución al problema principal (la mejor partida) a través de una secuencia de soluciones de distintos subproblemas (en este caso, un conjunto de turnos), las cuales no tienen relación directa entre sí. Es por esto que para esta estrategia se iteran todos los turnos de la partida, sin pensar en la partida en su totalidad. Es decir, se busca localmente una solución conveniente. Se intenta contestar la pregunta: ¿para este turno, qué es lo que conviene?

Se implementaron dos ideas:

- Greedo: la versión más inteligente de ambas. Cada turno busca producir la menor cantidad de puntos posibles para el bando contrario. Es decir, en cada turno se busca matar la mayor cantidad de barcos posibles, o bien, producir el mayor daño.

- GreedoNaive: Este jugador utiliza dos 'grados' de técnicas Greedy. Se considera al mejor turno como el que más daño produce, sin más. Por lo tanto, dentro de cada turno se tiene un subproblema: cada lanzadera. Esto permite iterar las lanzaderas y llegar a que cada una dispare al barco que más vida le pueda sacar (de estar muerto, pasar al siguiente barco).

Un ejemplo de cómo difieren ambas versiones en elegir el turno. Suponiendo 3 barcos y dos lanzaderas, en el primer turno:

- Barco 1: 300 puntos de vida - Barco 2: 200 puntos de vida - Barco 3: 100 puntos de vida.
- Matriz de daños: 60 para el primer barco, 50 para el segundo, 50 para el tercero.

La versión Naive de Greedo diría que el mejor turno es el de disparar dos veces al primer barco, ya que genera 120 puntos de daño y siempre busca maximizar esto. Por otro lado, la versión más inteligente de Greedo disparará al tercer barco, ya que busca minimizar los puntos del rival y en este turno puede derribar a un barco.

Se destaca que esta estrategia no es óptima: el conjunto de turnos considerados óptimos no llega necesariamente a la solución óptima (ver 1.3).

Solo a modo de comparación, se implementó la estrategia Brutus, quien con fuerza bruta encuentra en cada turno todas las posibles combinaciones de disparos y elige el disparo óptimo.

Entre las dos ideas, Greedo es algo más inteligente, por lo tanto, se opta por considerarlo como la técnica greedy elegida para la partida. Greedo no debe tener impacto sobre la partida misma: no debe poder sacarle vida a los barcos, ya que esto es responsabilidad de la partida. Es por esto, que se realiza una copia[1] de la partida, donde se puede desarrollar la estrategia sin modificar la partida actual.

Hasta que la partida de Greedo no esté terminada, se juega cada turno. Dentro de estos turnos, lo que se hace es obtener el barco que se pueda matar en este turno con la menor cantidad posible de disparos, considerado el barco de mayor prioridad. De no haber ninguno para matar, se elige como barco de mayor prioridad aquel que más daño recibirá.

A continuación se presenta el pseudocódigo de Greedo:

[1] Haciendo uso del módulo `copy` de Python y su función `deepcopy`

```
def elegirTargetsDeLaPartida():
      turnos = []
      Mientras haya un barco vivo:
            turnos += elegirTargetsDelTurno()
            jugar el turno en una partida simulada
            pasar al siguiente turno
      return turnos

def elegirTargetsDelTurno():
      targets = []
      Por cada barco b:
            Calcular con cuántos lanzaderas lo mato en este turno

      Por cada lanzadera:
            target = Barco que menos lanzaderas necesita
            Si no puede ser destruído en este turno:
                  target = Barco que más daño recibiría
            Restar vida al barco equivalente a su casillero.
            Si el barco está muerto, quitarlo de los barcos considerados
            targets += target
      return targets
```

Para empezar a analizar la complejidad, se ve que la función para `elegirTargetsDeLaPartida()` itera en la medida en que haya un barco que no ha sido destruido. Es decir, se iteran T veces, siendo T la cantidad de turnos a jugar (indefinida).

Resta ahora hallar la complejidad de `elegirTargetsDelTurno()`. Iterar los barcos para guardar sus atributos es $O(B)$ (con B la cantidad de barcos) y para cada barco solamente se averigua cuántas lanzaderas se necesitan para matarlo, que son una división $O(1)$. Luego, dentro de esta función se itera por cada lanzadera: $O(L)$ (con L la cantidad de lanzaderas). Dentro de cada una de estas lanzaderas la única función costosa es la de obtener el barco con mayor prioridad. Obtener el máximo de un diccionario es en tiempo lineal, y en el peor de los casos se debe obtener dos veces (barco que puedo matar, y luego barco que mayor daño recibe), esto es $O(2B)$

Entonces se tiene una complejidad de: $O(\ T\ (B\ + L(2B))\ ) = O(\ T\ L\ B\ )$

En un caso arbitrariamente malo para jugar, donde hay una lanzadera y todos los casilleros quitan un punto de vida, T equivaldría a la suma de los puntos de vida de todos los barcos, siendo esto una cota superior para la complejidad. Por lo tanto, se podría expresar la complejidad como: $O(\ P\ L\ B\ )$ siendo P la suma de la cantidad de puntos de vida de los barcos.

En el peor caso, donde hay una lanzadera y todas las filas contienen solo un casillero donde quitan 1 punto de vida mientras que el resto de la fila no quita puntos, y encima teniendo el casillero que daña sobre la misma columna en todas las filas, se debe jugar para cada barco $Pi * C$ turnos, siendo Pi los puntos de vida del barco i y siendo C la cantidad de columnas. Esto es porque tengo que esperar a dar una vuelta entera que equivale a C turnos para lograr sacarle 1 punto de vida, y por ende debo hacerlo Pi veces.

Con esta idea, la cantidad de turnos totales es $C * (Pi + ... + Pn) = P * C$, (manteniendo que P es la sumatoria de vidas de barcos) dejando así: $O(\ P\ C\ L\ B\ )$

Lo cual muestra que la complejidad del algoritmo puede llegar a depender de todas las variables manejadas por el juego (puntos de vida, cantidad de columnas, cantidad de lanzaderas y cantidad de barcos).

Un ejemplo ilustrado del peor caso:

```
100hp | 1     0     0     0
200hp | 1     0     0     0
300hp | 1     0     0     0
```

Acá, se deben jugar 600*4 turnos para lograr terminar la partida.

De la misma forma que Greedo, Greedo Naive no debe tener impacto sobre la partida misma: no debe poder sacarle vida a los barcos, ya que esto es responsabilidad de la partida.

Nuevamente, por la esencia de la técnica Greedy empleada y análogo a lo sucedido en Greedo, puedo iterar mis turnos. Lo primero que hago es ordenar los barcos según quien recibirá más daño en este turno, para dejar la decisión del siguiente subproblema más fácil de acceder. Esta vez, dentro de cada turno tengo un nuevo problema Greedy, el de las lanzaderas a elegir. Por ende, vuelvo a iterar, esta vez las lanzaderas. El target esta vez es simplemente el último barco de mi lista de barcos ordenados, que es el que más daño recibirá. En el caso de matar a un barco, se lo saca de la lista de los barcos considerados (para evitar disparar a un barco muerto).

```
def elegirTargetsDeLaPartida():
      turnos = []
      Mientras haya un barco vivo:
            turnos += elegirTargetsDelTurno()
            jugar el turno en una partida simulada
            pasar al siguiente turno
      return turnos

def elegirTargetsDelTurno():
      targets = []
      barcosOrdenados = sort(barcos según daño ascendentemente)
      Por cada lanzadera:
            target = barcosOrdenados[-1]
            Restar vida al barco equivalente a su casillero.
            Si el barco está muerto: barcosOrdenados.pop()
            targets += target
      return targets
```

Primero se itera por los turnos, es decir llamaré a `elegirTargetsDelTurno()` T veces. Luego, se ordena los barcos en $O(B \log B)$. Ahora, se itera por cada lanzadera, por ende itero L veces. Dentro de cada iteración las operaciones son $O(1)$ ya que no son más que asignaciones y el método pop de la lista, que al estar aplicado sobre el último elemento no es lineal (como si lo sería en caso de haber ordenado descendentemente y por ende tener que remover el primero).

Entonces se tiene una complejidad de: $O(T(B \log B + L))$ [2]

Como se puede ver, Greedo Naive es muy sencillo y se siente una versión preliminar de Greedo, después de todo es el mismo esqueleto solamente que a Greedo se le agregó la consideración (inteligente) de tener en cuenta el concepto de 'matar si puedo en este turno'. Es de notar que Greedo, en el caso de no poder matar barcos en el turno, actúa de la misma forma que lo haría Greedo Naive.

La estrategia dinámica consiste en jugar nuevas partidas teniendo en cuenta partidas anteriores. Para este caso no sirve una estrategia donde se itera por turnos (como si se hacía en greedy) porque aunque se

[2] Esta complejidad no se puede acotar aún más y simplificar, ya que depende de 3 variables distintas, y si bien es linearítmico para los barcos y lineal para las lanzaderas (dentro de la multiplicación por los turnos), no se puede suponer una variable más 'pesada' que otra. Queda en función de las 3 variables.

pueda llegar a una sintaxis de programación dinámica, no se lograría una optimización por partidas. Lo que se busca es jugar la mejor partida, no el mejor turno. Por ende, se debe buscar cómo hacer partidas en base a otras. Pero como no hay forma de relacionar partidas enteras a partidas enteras (ya que si voy a calcular toda la partida vuelvo al problema original), hay que buscar forma de reducir partidas en partidas más pequeñas. Luego, teniendo en cuenta lo ya jugado, se incrementan las variables de estas partidas reducidas hasta llegar a la partida total, para resolver el problema original.

Una partida está determinada por las siguientes variables:

- L lanzaderas
- B barcos
- N columnas, teniendo en cuenta que el barco al pasar la columna N vuelve a la primera hasta ser derribado.

Por lo tanto, la cuestión a la que se arriba es ¿que variable reduzco para tener mis partidas parciales?

Se optó por reducir la partida original (el problema) a partidas de solo una lanzadera y un barco a la vez, y tomando el tablero como finito.

Dentro de cada una de estas partidas reducidas se encuentran todas las combinaciones posibles para destruir el barco de forma única y suficiente (sin disparar a un barco ya muerto). Esto determina cuántas partidas posibles, dentro de la partida reducida tiene cada barco para jugar. Lo que se nota es que el barco con menor cantidad de partidas posibles es el barco más difícil de matar.

Partiendo de las partidas posibles para el barco más difícil de matar, se agrega un barco, y entonces, se juega la partida teniendo en cuenta que las lanzaderas en las que ya se disparó no pueden ocuparse de nuevo.

Así, con todas las posibles partidas del barco que se tienen, y las posibles partidas de el barco que se agregó, se llega a un conjunto de partidas posibles a jugar para esos dos barcos. De estas, se elige la partida que menos lanzaderas vacías tenga, para evitar partidas largas y por ende que más puntos generen. La partida seleccionada como la mejor es la que pasa a ocupar el lugar de la partida que tenía anteriormente en el diccionario de programación dinámica (llamado resultados en la implementación).

Es así como se va mejorando siempre la lista de partidas posibles teniendo en cuenta las partidas ya jugadas, hasta llegar a un set de partidas posibles a jugar. De estas, se elige la que menos puntos produzca, y esa es la partida a devolver.

A continuación se presenta el pseudocódigo:

```
def elegirTargetsDeLaPartida():
    for barco in barcos:
       guardar posibles formas de ganar la partida reducida
       (partida reducida = 1 lanzadera, barco unico, matriz finita)

    heap = heap de barcos según dificultad de matar
    barco más difícil = heap.desencolar()
    resultados = posiblesPartidas(barco más difícil)

    while heap.no_esta_vacio():
         siguienteBarco = heap.desencolar()
         posiblesPartidasSiguienteBarco = posiblesPartidas(siguienteBarco)
         for partida in resultados:
              posiblesPartidas = []
              for partidaSiguiente in posiblesPartidasSiguienteBarco:
             posiblesPartidas +=
partida.jugarTeniendoEnCuentaOtraPartida(partidaSiguiente)
              resultados[partida] = mejorPartida(posiblesPartidas)
      return mejorPartida(resultados)

def mejorPartida():
```

```
   devuelve la partida que menos lanzaderas en nulo tenga (así se evitan huecos y
puntos ciegos y por ende se termine antes el juego)

def posiblesPartidas():
    for elemento in casilleros:
     guarda combinaciones posibles de números que su sumatoria sea mayor a la vida
del barco

def jugarTeniendoEnCuentaOtraPartida():
   Si no son compatibles las partidas: return False
   partidaNueva = []
   for turnoViejo, turnoActual in partidaVieja, partidaActual:
      partidaNueva += turnoViejo + turnoActual
   return partidaNueva
```

En el transcurso de la implementación de la solución propuesta se encontraron varias complicaciones, siendo la mayor la de encontrar las posibles combinaciones para la partida reducida. Idealmente, para mejorar la estrategia, se debería reducir las partidas a solo tener un barco. El número de lanzaderas y el tablero deberían quedar igual al problema original. Para lograr esto, se podría hacer una función que encuentre las combinaciones que maten a un barco teniendo en cuenta múltiples disparos sobre el mismo casillero y también que al llegar al final de la fila se vuelva a comenzar, lo cual llevaría a un diccionario de programación dinámica más extenso y moldeable. Esto es para evitar el caso donde un barco no se pueda destruir con la sumatoria de todos sus casilleros. El problema de esta implementación es que ya de por sí es muy costoso buscas todas las combinaciones de números que matan a un barco solo teniendo una lanzadera y sabiendo que la fila se termina. Si hubiera que agregar a esta función que tenga en cuenta que pasada la lista original de números haya que nuevamente volver a empezar, y que en las permutaciones se van a considerar múltiplos de cada número, la función pasaría a costar considerablemente más. Es por esto que se tuvo que sacrificar esta idea y tomar las partidas parciales como se hicieron.

En cuanto a la complejidad del algoritmo, ya se puede ver que va a ser considerablemente grande al tener en cuenta se parte de la idea de jugar la partida de todas las formas justas posibles. Primero, se considera que la complejidad por iterar sobre los barcos es $O(B)$.

Para guardar las posibles formas de ganar una partida reducida, se deben calcular las distintas combinaciones de recorridos del barco sobre la fila de la grilla (considerando la fila sin volver a empezar sobre el inicio de la misma). Si la cantidad de columnas de la grilla es C, esa cantidad de combinaciones es $C!$ (por considerar el recorrido posible desde cada casillero hasta los siguientes). A su vez, se itera cada una de estas combinaciones (que en el peor de los casos tienen C elementos) para eliminar las combinaciones duplicadas (siendo ese remove $O(C)$) y otras operaciones de índices también acotadas por la cantidad de columnas. Es decir, hasta aquí se tiene una complejidad $O(B\,C!\,C^2)$

La creación del heap se hace en O(B log(B)), complejidad que se desprecia frente a lo anteriormente analizado. Al iterar ese heap ($O(B)$), en síntesis, se debe hacer verificaciones por cada partida posible, y por cada turno de esa partida, lo cual puede acotarse como $C!$ T.

Totalizando, se tiene una complejidad de: $O(B\,C!\,C^2 + B\,C!\,T)$

Otra estrategia que había surgido fue la posibilidad de, en lugar de usar la cantidad de combinaciones como criterio de dificultad, utilizar la cantidad de tiempo mínimo que tomaría derribar un barco (siendo el tiempo mínimo la cantidad de turnos mínima necesaria). Una vez logrado eso, se toma al barco más sencillo de derribar según la combinación más sencilla para hacerlo, y se repite hasta quedarse sin barcos (y por supuesto, se calcula para cada barco con las restricciones del anterior).

1.2. Juego

El juego consiste en una serie de partidas, cada una con un jugador diferente. Al finalizar las partidas, se decide el ganador del juego (el que logró que el Bando A genere la menor cantidad de puntos posibles). El juego puede ejecutarse especificando la cantidad de lanzaderas, y proporcionando un archivo de la grilla (que si no se encuentra, se genera con parámetros por defecto). Una vez que se carga el archivo de la grilla

con la información de los barcos y la matriz, y teniendo en cuenta la cantidad de lanzaderas especificadas, se agregan los jugadores al juego. Por cada jugador, los barcos se inicializan con su correspondiente nivel de vida inicial y se inicializa su partida asignando a los barcos en sus posiciones iniciales.

Para cada partida se le pide al jugador correspondiente la estrategia a utilizar, es decir, una lista de los disparos que debe realizar cada lanzadera en diferentes turnos para destruir a todos los barcos. Por cada una de las series de disparos a realizar en cada turno que ha indicado el jugador, la partida daña a los barcos que correspondan a los disparos dependiendo de su casillero actual (en la clase Partida). Al final de cada turno, se avanzan todos los barcos en un casillero sobre su propia fila. Esto se repite hasta que se completa la lista indicada por el jugador, luego de lo cual deben haberse destruido todos los barcos, y pasar al siguiente jugador para que haga lo propio. Cada partida guarda la cantidad de puntos que hizo el bando contrario (según cuánto haya hecho resistir sus barcos a lo largo de los turnos) y finalmente, se indica el ganador según quién tuvo más puntos.

1.3. Condiciones para que la Estrategia Greedy Sea Óptima

Basta dar un contraejemplo para mostrar que Greedo no se comparta de forma óptima. Dada una partida con una sóla lanzadera y la siguiente grilla (donde la primera columna representa la vida del barco) :

```
1000    100    100    100    100    1000
50        1      1     40     60       1
10        2      8      1      1       1
```

se tiene que Greedo empezará por atacar al primer barco, porque es al que más daño le puede hacer sin poder matar a ninguno. Pero en realidad, lo óptimo hubiera sido atacar al barco con 10 puntos de vida, ya que se lo podría haber destruído en el siguiente turno (y lo importante es minimizar la cantidad de barcos vivos tan rápido como sea posible para minimizar la cantidad de puntos de A). A su vez, en los dos turnos siguientes se destruiría al barco con 50 puntos de vida, y finalmente, en un sólo turno se podría al barco con más vida.

Lo anterior muestra que no por el hecho de haber una sola lanzadera Greedo es óptimo. Pero, para los casos triviales de cualquier tablero en donde en cada turno se pueda destruir de un tiro por lanzadera a cada barco, Greedo es una estrategia óptima, ya que el algoritmo verifica que la cantidad de lanzaderas requeridas para destruir un barco en un turno dado son viables para eliminarlo efectivamente en ese turno. Un ejemplo de esto es la siguiente grilla:

```
1000   1000   1000   1000   1000
50     1000   1000   1000   1000
10     1000   1000   1000   1000
```

Allí se destruirán tantos barcos por turno como lanzaderas halla, por lo que no se pueden minimizar más los puntos de A.

1.4. Interfaz Gráfica

La interfaz gráfica del juego es por consola de comandos (la misma que se utiliza para inicializar el juego).

Al iniciarse el juego (en el main) se inicializa su correspondiente vista y se imprime el título del juego. Por cada jugador que juega una partida se tiene una vista de la partida que muestra la cantidad de lanzaderas, el jugador y la cantidad de barcos iniciales, adicionalmente al número de turno y una representación de la grilla para cada turno. Al terminar la partida del jugador, se muestra un resumen del fin de la partida, y la vista del juego se encarga de informar el cambio de jugador, así como indicar el ganador al final. Por defecto, se requiere que el usuario pase de turno con la tecla enter.

1.5. Estrategia para el Jugador A

La estrategia para el jugador A consiste en ir iterando por cada elemento de las filas de los barcos, y posicionando al barco en la columna donde pueda sobrevivir más, es decir, dejado al inicio de la hilera más larga que sume menos que la vida del barco. Correspondiente, se consigue el mayor de los subconjuntos de

filas que sumen menor o igual a la vida del barco. Se destaca que los daños de cada casillero se multiplican por la cantidad de lanzaderas, ya que se tiene en cuenta que el jugador B va a atacar con todas sus lanzaderas al mismo barco. En caso de morir en todo casillero, simplemente se coloca sobre el menor de estos.

Como el algoritmo implica elegir la mejor columna para cada barco, se tiene la complejidad de iterar por cada uno de los barcos (O(B)). Para elegir la mejor columna se itera por sobre todas las columnas del tablero (O(C)) realizando dentro de esta iteración operaciones de tiempo constante, por lo que el algoritmo implementado para posicionar los barcos para el jugador A es de orden O(BC) . Como el algoritmo no toma en cuenta la infinidad del tablero, los subsets calculados terminan a lo sumo en el último elemento de la columna. En el caso de que un barco no puede ser eliminado incluso disparando la máxima cantidad de veces en todas las columnas, el algoritmo coloca al barco en la primera posición, por lo que no agrega complejidad al algoritmo. Si se hubiera tomado en cuenta la infinidad del tablero entonces se habrían calculado C cálculos para la mejor columna lo cual hubiera llevado a un orden de complejidad más elevado.

2. Sabotage

2.1. Algoritmo de Protección

El problema se puede interpretar como la posibilidad de quitar una arista en una red de flujo, donde las capacidades de las aristas representan la "cantidad de información" que se puede transportar por esa vía. K.A.O.S. aprovechará su conocimiento de la red para eliminar la arista que más haga descender el flujo máximo que la red puede transportar, maximizando así el daño de su sabotaje. Por otra parte, C.O.N.T.R.O.L. sólo puede evitar que dos aristas particulares sean eliminadas de la red. Así, la idea del algoritmo debe ser identificar las dos aristas "más importantes" para maximizar el flujo de la red, haciendo que la eliminación de cualquier otra arista generé un descenso en la capacidad de flujo máximo que sea menor que la que se hubiera tenido por la eliminación de cualquiera de las otras dos aristas protegidas.

Por empezar, se debe determinar cómo se considera la "relevancia" de las aristas de la red, de forma que C.O.N.T.R.O.L. pueda proteger las dos más importantes. Una arista es importante si constituye un cuello de botella (en adelante, bottleneck) para un camino dado de la red, ya que cualquier otra arista que integre dicho camino tiene su flujo acotado por la capacidad de ese bottleneck. Ahora, como el flujo de la red en última instancia es la suma de los cuellos de botella encontrados para los distintos caminos elegidos, se tiene que las dos aristas con más relevancia a priori son los dos mayores cuellos de botella encontrados en los distintos caminos.

Pero lo anterior plantea el inconveniente de asegurarnos que esos cuellos son máximos, es decir, que no se pueden encontrar caminos alternativos en donde se obtiene un flujo mayor. De esa forma, se ve que es útil aplicar el algoritmo de Ford Fulkerson para resolver esto: si se puede obtener el flujo máximo de la red, se puede identificar los máximos cuellos de botella, y así, tener las dos aristas más relevantes para maximizar el flujo.

Por otra parte, se deben considerar las aristas que desconectan la red (en adelante, aristas-corte). Se dice que una arista desconecta la red si su sola remoción divide al grafo en dos regiones, donde en cada una se encuentra o bien la fuente de la red o bien el sumidero, por lo que eliminar una de ellas hace que el flujo máximo de la red sea cero, lo cual las convierte en candidatas primarias a ser protegidas. Para conocerlas, en este caso sí puede realizarse un algoritmo de fuerza bruta donde se verifica si luego de quitar una arista la red, la misma quedó desconectada (que es polinomial con respecto de la cantidad de aristas, no como la solución por fuerza bruta del problema planteado en el enunciado, que es de un orden $O(A^2c)$ con $O(Ac)$ la complejidad del algoritmo de Ford Fulkerson), siendo que no necesariamente un arista-corte es un bottleneck (Fig. 1):

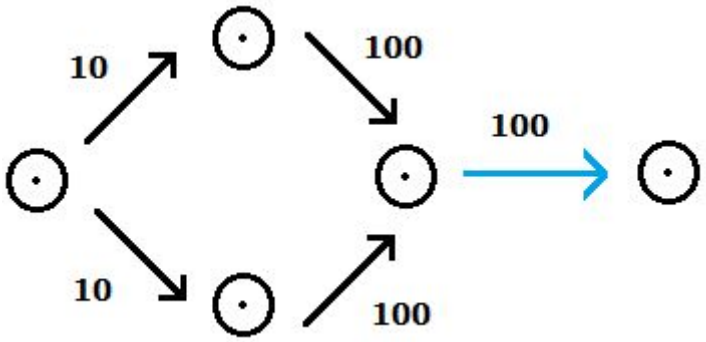

Fig. 1: La flecha azul muestra un ejemplo de arista-corte que no es bottleneck.

Entonces, sabiendo que se pueden conocer las aristas-corte, primero se intentarán proteger las mismas, protegiendo luego los bottlenecks en orden de capacidad (Fig. 2).

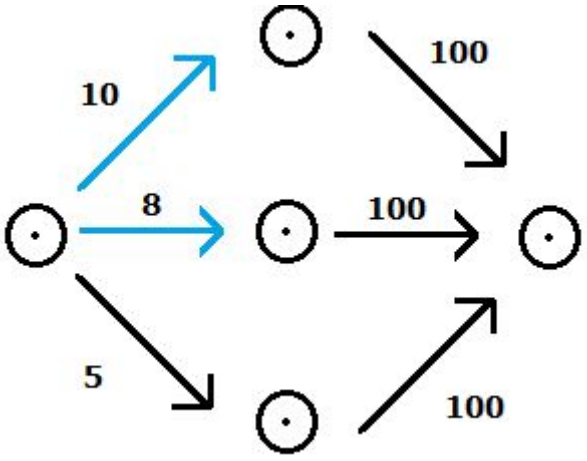

Fig. 2: Cómo sólo se pueden proteger dos aristas, se protegen las dos aristas en color azul.

Independientemente de los casos en donde el algoritmo no puede evitar el sabotaje (ejemplo en Fig. 3), un problema que plantea la estrategia presentada es que se podría proteger el bottleneck más importante, y aún así los saboteadores podrían desconectar una arista que impida aprovecharlo, y en ese caso el daño realizado será el mismo. Diremos que el algoritmo no es óptimo, porque no considera todos los casos borde en donde utilizar una estrategia más específica que la anteriormente descrita hubiera conducido a una mejor protección (ejemplo en Fig. 4). Se debe tener la consideración adicional de que el algoritmo de Ford Fulkerson tampoco elige la arista más adecuada para los caminos de circulación en todos los casos, por lo que eso también contribuye a la falta de optimización de la solución planteada.

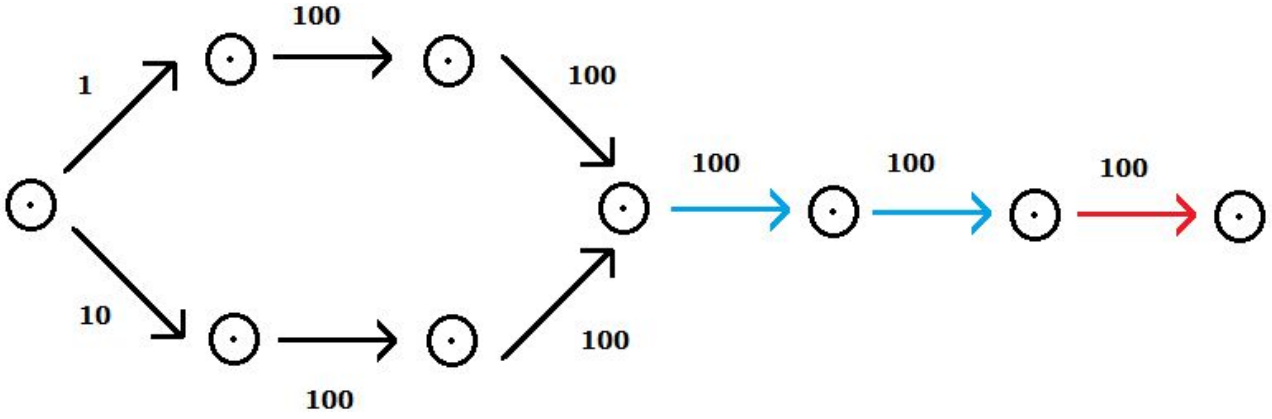

Fig. 3: Aunque se protegan las aristas en azul, romper la arista en rojo desconecta el grafo.

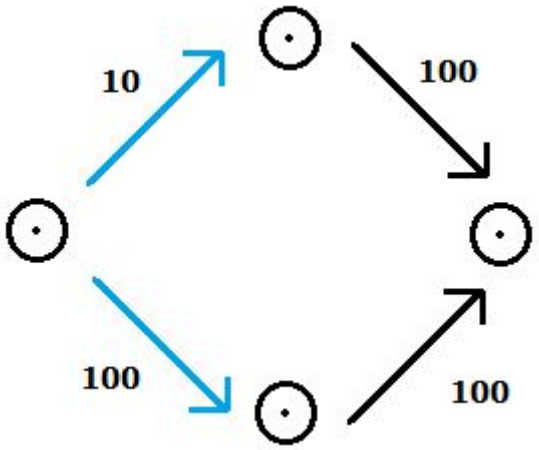

Fig. 4. Ejemplo de uno de los casos donde el algoritmo no se comporta de forma óptima.

A partir de las ideas anteriores se diseña el algoritmo:

- primero se ejecuta el algoritmo de Ford Fulkerson para obtener los bottlenecks encontrados: los mismos se ordenan por capacidad decrecientemente;
- se hallan las aristas-corte de la red;
- se intentan proteger todas las aristas-corte posibles;
- luego, se intentan proteger todos los cuellos de botella posibles.

A continuación se presenta el pseudocódigo.

```
def proteger2Aristas(red):

  cantidadDeVigilancias = 2
  aristasProtegidas = []
  flujoMax, bottleNecks =  FordFulkerson(red)
  aristasCorte =  red.obtenerAristasDeCorte()

  for a in aristasDeCorte:
     if( una arista de corte está entre los cuellos de botella)
            cuellosDeBotella.remove(a)

  cuellosDeBotella.ordenarDescendientementePorCapacidad
  agregar a las aristas protegidas tantos aristas-corte como sea posible
  agregar a las aristas protegidas tantosbottlenecks como sea posible
  devolver aristasProtegidas
```

2.2. Complejidad del Algoritmo

Se sabe que la complejidad del algoritmo de Ford Fulkerson en sí mismo es de $O(cA)$, donde c es la suma de todas las capacidades de las aristas, y A es la cantidad de aristas. A esto se le debe sumar la complejidad de guardar los bottlenecks de cada iteración, que en el peor caso, $O(A)$, por lo que no modifican asintóticamente la complejidad del algoritmo de Ford Fulkerson.

Obtener las aristas de corte consiste en aplicar el correspondiente algoritmo de fuerza bruta, esto es, por cada arista de cada camino, verificar si quitándola se desconecta la red. Si por cada arista ($O(A)$) se la debe eliminar de la red ($O(A)$), y luego verificar que la red siga siendo conexa (que es verificar si existe un camino en profundidad de la fuente al sumidero, $O(V+A)$) y luego volver a agregar la arista (O(1)) se tiene que la complejidad del algoritmo de fuerza bruta es: $O(A\,(A+V+A))$. Considerando que por ser una red de transporte, cada vértice tiene al menos una arista incidente (excepto la fuente), se puede acotar la cantidad de vértices con la cantidad de aristas, y entonces se tiene una complejidad de: $O(A^2)$

Verificar si hay aristas-corte que también son bottlenecks implica por cada arista-corte (en el peor caso, $O(A)$) verificar si esa arista está entre los bottlenecks ($O(A)$), y en tal caso eliminar esa arista del arreglo de bottlenecks (en el peor caso, $O(A)$). Entonces se tiene una complejidad de O(A(A +A)) = $O(A^2)$.

El ordenamiento de los bottlenecks se puede lograr en $O(Alog(A))$. Por otra parte, lo de agregar tantas aristas como sea posible consiste simplemente en iterar el arreglo correspondiente ($O(A)$) y en la medida en que haya ”vigilancias” disponibles , se agrega la arista al arreglo (O(1)) que se terminará devolviendo.

En síntesis, se tiene una complejidad de: $O(cA + A^2 + A^2 + Alog(A) + A) = O(cA + A^2)$

Incluso en el caso más benévolo en donde todas las aristas tienen una capacidad mínima de 1, A está acotado superiormente por c, por lo que eliminar los bottlenecks repetidos en las aristas-corte y buscar estás últimas (A^2) no agrega más complejidad de la que ya tiene el algoritmo de Ford Fulkerson. Entonces se tiene una complejidad de: $O(Ac)$, por lo que el algoritmo propuesto no modificaría asintóticamente la complejidad del algoritmo de Ford Fulkerson.

Ahora, sea X el problema del enunciado. Sea Y el problema que resuelve el algoritmo de Ford Fulkerson, es decir, el de devolver el flujo máximo para una red de flujo dada. Si al algoritmo de Ford Fulkerson se le realizan modificaciones para que pueda devolver además los bottlenecks, entonces se tiene la implementación FordFulkerson ya utilizada. FordFulkerson resuelve Y, porque devuelve el flujo máximo además de los bottlenecks, y las modificaciones que se le realizan al algoritmo, como ya se mostró, son de de orden polinómico (con respecto de la cantidad de aristas). El resto de las operaciones que se describen en la función proteger2Aristas también son polinómicas como ya se analizó. Luego, como proteger2Aristas resuelve el problema Y, por llamar a FordFulkerson, e involucra una serie de llamadas polinómicas para además resolver X, por lo que se puede afirmar que Y <= X (es decir, Y es una reducción polinomial X), por lo que el problema de hallar el máximo flujo de una red de flujo es una reducción del problema planteado en el enunciado (entendiendo la solución de X como no óptima). Si se quisiera conocer la solución de Y, bastaría con hacer que la función de proteger2Aristas devolviera también el flujo máximo de la red.

2.3. Flujo Máximo en la Red

El algoritmo de flujo máximo fue programado como parte de la solución anterior, como ya se describió, a partir de una mínima variante del algoritmo de Ford Fulkerson. Para verificar el flujo antes y después de remover alguna de las aristas protegidas, se carga la red a partir del nombre del archivo especificado (por defecto, “redsecreta.map”), y con proteger2Aristas se obtienen las aristas a proteger. Luego, por cada una, se eliminan temporalmente de la red, se computa el flujo máximo para el nuevo estado de la misma, y luego se vuelve a agregar la arista borrada originalmente.

Existe un cierto sesgo al utilizar el flujo máximo de la red sin cada una de las aristas protegidas como un indicador del nivel de vigilancia (o protección) que se tiene. Por ejemplo, en el grafo de la fig. 3, se tiene que si no se protegen cualquiera de las dos aristas seleccionadas (en azul) el flujo máximo de la red será cero. Esto a priori podría ser un indicador de que la protección es la mejor posible (en cierta forma, de hecho lo es), pero sin embargo, en el caso de un sabotaje el flujo máximo de la red también sería cero (arista en rojo), por lo que la lectura de este indicador no da una idea completa de que tan buena es la vigilancia realizada.

2.4. Solución con Varias Fuentes y Sumideros

Sea R1 una red de transporte con varias aristas y sumideros. Si a cada una de las fuentes de R1 del se la hace incidente de una arista proveniente de un vértice, al que se llamará "fuente auxiliar" (con aristas de capacidad "infinita" para que no afecten el flujo que puede transportar la red) y a cada uno de los sumideros se lo hace adyacente a un vértice, que se llamará "sumidero auxiliar" (nuevamente, a partir de aristas con capacidad tal que mantenga inalterado el flujo máximo de la red) en una nueva red R2, luego de lo cual se llama a un algoritmo como proteger2Aristas (que halla las aristas a proteger en una red con una sola fuente y un solo sumidero) se tiene que se pueden encontrar las aristas a proteger de R1, ya que ninguna de las aristas agregadas adicionalmente deberá ser protegida: sus capacidades son mayores a todas las existentes,

por lo que no serán bottlenecks, y a su vez todas se está agregando a caminos existentes en la red -de la fuente al sumidero- por lo que no serán aristas-corte. Es decir, como las aristas a proteger para R2 son las mismas que para R1, con un algoritmo como el definido anteriormente se puede hallar una solución al problema planteada, aunque esta no será óptima. En cualquier caso, si se encontrará una solución óptima para proteger R2, por lo indicado anteriormente, también existiría para R1.

3. Anexo: Procedimientos para la Ejecución y Compilación

3.1. Compilación

Como el trabajo fue realizado en Python, no debe ser compilado. La versión de Python usada fue 3.6.1, para la arquitectura 64 bits (el código podría llegar a ser compatible con versiones previas de Python 3, pero se recomienda utilizar la indicada).

3.2. Ejecución

La ejecución del juego 'Battleship' consiste en solamente llamar al archivo main. Lo primero que hace el archivo es confirmar hay un mapa de coordenadas en el directorio. Si no existe, lo crea (con los parámetros por defecto), haciendo uso del módulo CrearGrilla. Luego de crear un Juego, le agrega los jugadores correspondientes, y para cada jugador juega una partida. Al finalizar todas las partidas, la vista imprime al ganador. Hay distintas opciones a la hora de ejecutar el programa:

```
$ python3 main.py // Si no existe, crea un archivo con los parámetros por defecto

$ python3 main.py --no-input // Desahibilita el input del usuario

$ python3 main.py --overwrite // Crea el archivo aunque ya exista

$ python3 main.py --lanzaderas 3// Siendo las lanzaderas el único parámetro que
no se puede tomar del archivo del mapa, se pueden especificar acá

$python3 main.py --set-posiciones-iniciales// Siendo las lanzaderas el único
parámetro que no se puede tomar del archivo del mapa, se pueden especificar acá
```

Por otro lado, para crear el mapa, si se quiere hacer por separado.

```
$ python3 CrearGrilla.py 5 10 500 1000 300 // Crea un archivo 'grilla.coords' con
5 filas, 10 columnas, barcos entre 500 y 1000 de vida, y casilleros de hasta 300 de
daño
```

Para la segunda parte, basta ejecutar el archivo Main.py en la carpeta Sabotage (con el comando "`python Main.py`") para visualizar el flujo máximo de la red (por defecto, la que se encuentra en el archivo redsecreta.map, en el mismo directorio que el Main.py) las aristas que se proponen vigilar, y la consecuencia a nivel flujo de no vigilarlas