Universidad de Buenos Aires

Facultad de Ingeniería

75.29 Teoría de Algoritmos

Trabajo Práctico 3, Team Tardos

Integrantes

- 95099, Martín Andújar
- 96525, Khalil Alejandro Moriello
- 96252, Seva Gavrilov

Primer cuatrimestre de 2018

Índice

1. Parte 1: Juego de batalla naval

1.1. Consigna

Podemos modelizar el juego como:

- El juego se realizar por turnos.
- El bando "A" tiene X barcos. Cada uno de ellos tiene Vi puntos de vida.
- El bando "B" tiene Y lanzaderas de misiles. Cada una tiene un alcance global (es decir que llega a cualquier lugar que desee.
- El tablero de juego tiene forma de grilla. Con igual cantidad de filas que barcos iniciales. Las columnas son ilimitadas (tantas como se necesiten en el juego). A efectos programáticos se puede considerar que al llegar al final de la grilla se "teletransporta" al inicio de la misma
- Cada barco se ubica al inicio de una fila y se desplaza por turno en 1 posición (a la columna siguiente).
- Cada lanzadera de misiles puede disparar 1 misil por turno
- El daño que puede realizar un misil está únicamente ligado a la grilla en la que se encuentra el barco en ese momento.
- Los valores de daño están predefinidos en la grilla al inicio del juego, no se modifican y son conocidos por "A"
- Por cada turno que pasa el equipo "B" recibe un punto por cada barco que siga en el juego
- Los barcos permanecen en el juego y avanzan siempre que conserven puntos de vida.
- Los puntos de vida y la cantidad de daño son valores enteros positivos.
- El objetivo de "B" es minimizar la cantidad de puntos logrados por "A"

Tenemos 2 amigos que usaran el juego. "Greedo" y "Dinámico". Jugarán partidas entre ellos alternando entre el bando "A" y "B". Siempre el juego se realizará de a pares conservando la misma disposición de tablero y fichas (lanzaderas y barcos).

- Diseñar para "Greedo" una estrategia greedy y para "Dinamico" una estrategia mediante programación dinámica. Presentar el pseudocódigo. Analizar sus complejidades
- Programar el juego permitiendo tomar diferentes grillas, lanzaderas y barcos. debe poder usarse cualquiera de las dos estrategias realizadas en el punto 1.
- Determinar si la solución greedy es óptima o si no lo es bajo qué condiciones puede serlo.
- (opcional) Realice una interfaz gráfica para visualizar la evolución del juego
- Imagine que el jugador "A" tiene la posibilidad de seleccionar para cada barco la columna inicial donde se ubicará su barco. Diseñe un algoritmo que le permita aumentar la cantidad a lograr. Analice su complejidad

Grilla:

Se deberá contar con un archivo con la definición del tablero de juego. El mismo deberá tener el siguiente formato:

- Un renglón por fila de grilla
- El primero número de cada fila es la vida del barco que se movera en ella.
- Seguido de números enteros separados por espacio para cada daño de recibir un misil en la columna

Ejemplo:

```
800 20 40 10 120
300 80 30 90 0
```

El número de filas determinará la cantidad de barcos del jugador "A". Por parámetro del programa se le debe pasar la grilla a utilizar, la estrategia y la cantidad de lanzaderas disponibles.

La salida del programa debe mostrar por cada turno:

- Número de turno
- Cantidad de barcos disponibles
- Vida restante de cada barco y daño potencial según celda en la que se encuentra
- Objetivos seleccionados por las lanzaderas.
- Puntos acumulados

A la finalización debe indicar

- Turnos totales
- Puntos acumulados.

1.2. Resolución del ejercicio

1.2.1. Supuestos

Se tomaron los siguientes supuestos:

- En un turno, primero disparan las lanzaderas, luego se mueven los barcos.
- Solo hay que desarrollar una estrategia para los disparos de las lanzaderas. Los barcos siempre avanzan un paso cada turno.

1.2.2. Planteo del juego

La resolución de nuestro juego se puede encontrar en el archivo `batalla_naval.py` que define la lógica del juego (definida en la clase `Juego`)y las clases y estructuras de datos que se van a utilizar. El juego necesita recibir como parámetros la ruta del archivo con información sobre el tablero, la cantidad de lanzaderas a utilizar y una instancia de `Estrategia`. `Estrategia` es una clase de la que se espera que esté definido el método `siguiente_turno`, que dado el tablero, cantidad de lanzaderas, información sobre barcos y número de turno actual devuelva una lista de barcos, a los que el juego deberá disparar en ese turno.

La lógica principal del juego es la siguiente:

1) Se parsea el archivo con el tablero especificado, los valores se guardan en estructuras internas que se van a explicar más adelante.
2) Se entra al loop principal, en el que: 2.1) Se le pide a la estrategia a qué barcos disparar en el turno actual 2.2) Se aplica a los barcos el daño de los misiles disparados 2.3) Se verifica cuantos barcos quedaron vivos. Si ninguno, se sale del loop principal y el juego termina. En el otro caso, se suma la cantidad de barcos vivos a los puntos acumulados. 2.4) Finalmente los barcos se mueven una posición para adelante, y se vuelve al comienzo del loop

1.2.3. Estructuras de datos

A continuación se explicarán las estructuras de datos utilizadas. Estas estructuras van a ser usadas por el juego, tanto para mantener la información sobre el estado del juego, como para proveérselas a la estrategia. La estrategia a su vez tendrá que usarlas para calcular los barcos a impactar en un turno.

Supongamos que el contenido del archivo de tablero es el del ejemplo del enunciado:

```
800 20 40 10 120
300 80 30 90 0
```

El juego va a guardar el tablero en un diccionario `tablero`, en el que las claves son los números de los barcos (que siempre se van a enumerar de 0 a *n-1*, donde *n* es la cantidad de filas en el archivo), y los valores son listas con daño potencial de cada posición. De esta forma, por ejemplo, para el barco 0 en la posición 0 el daño potencial es de 20. Esta estructura se mantiene estática en el transcurso del juego.

```
{
  0: [20, 40, 10, 120],
  1: [80, 30, 90, 0]
}
```

La información de los barcos se va a guardar en un diccionario `barcos`, que como claves tendrá los números de los barcos, y como valores, diccionarios con la información de cada barco. Ese diccionario de información tendrá la posición actual de cada barco, y su vida actual.

```
{
  0: {'posicion': 0, 'vida': 200},
  1: {'posicion': 0, 'vida': 300}
}
```

Para dar un ejemplo, supongamos que tenemos 2 lanzaderas. En el primer turno, la primer lanzadera le dispara al barco 0, y el segundo al barco 1. Al principio del turno 2, el diccionario de barcos tendrá el siguiente contenido:

```
{
  0: {'posicion': 1, 'vida': 180},
  1: {'posicion': 1, 'vida': 220}
}
```

1.3. Estrategia Greedy

1.3.1. Algoritmo elegido

Como base para la estrategia Greedy se utilizó la siguiente idea: en cada turno se quiere hundir la mayor cantidad de barcos, para minimizar los puntos logrados por el contrincante. Si no se puede hundir más barcos, pero quedan disparos a hacer, se selecciona el disparo que infiere mayor daño.

En pseudocódigo la solución elegida se puede describir de la siguiente forma:

```
def siguiente_turno(tablero, cantidad_lanzaderas, barcos):
    disparos = []
    disparos_pendientes = cantidad_lanzaderas

    while disparos_pendientes:

        barco_a_hundir = null
        barco_a_disparar = null
        for barco in barcos:
            if barco.vida < 0:
                continue
            disparos_necesarios = barco.vida / danio_potencial(barco)
            if disparos_necesarios < disparos_pendientes:
                if es_mejor_hundir(barco, barco_a_hundir):
                    barco_a_hundir := barco
            if es_mejor_disparar(barco, barco_a_disparar):
                barco_a_disparar := barco

        if barco_a_hundir:
            numero_de_disparos := aplicar_disparos(barco_a_hundir)
            disparos_pendientes := disparos_pendientes + disparos_pendientes
        elif barco_a_disparar:
            aplicar_disparos(barco_a_disparar)
            disparos_pendientes := disparos_pendientes - 1
        else:
            break

    return disparos
```

En el pseudocódigo se hace uso de algunas funciones auxiliares, que no están presentes en el código real, y se usan para simplificar la lógica del pseudocódigo. Son las siguientes:

- `es_mejor_hundir(barco, barco_a_hundir)`: toma la decisión de si es mejor hundir `barco` que `barco_a_hundir`, actualmente seleccionado para hundir. En esta decisión se decide por `barco` si:
  - `barco_a_hundir` no está definido
  - Si la cantidad de disparos que toma hundir `barco` es menor que `barco_a_hundir`
  - Si la cantidad de disparos es la misma, pero el daño total es mayor.
- `es_mejor_disparar(barco, barco_a_disparar)`: decide si es mejor dispararle al `barco` que al `barco_a_disparar`, actualmente seleccionado para disparar. Se decide por `barco` si `barco_a_disparar` no está definido, o si el daño que se le va a impactar es menor.
- `aplicar_disparos(barco_a_disparar)` es la función que le quita vida a los barcos, antes de volver a realizar el loop, y devuelve la cantidad de disparos que tomó. En el código real se mantiene una estructura con la vida actual de los barcos, y se actualiza en este momento.

Un chequeo más que se hace es si el daño potencial a un barco es igual 0, en cuyo caso se ignora ese barco. Si al final de una iteración externa no se eligió ningún barco para hundir, ni para disparar, es porque o bien no quedan más barcos con vida, o solo es posible hacer 0 daño con el disparo. En ambos casos el resto de los disparos se hace al barco 0 (ya que no importa a qué barcos disparar).

1.3.2. Análisis de complejidad

Todas las estructuras usadas tienen costo de acceso y de actualización de $O(1)$, asique no van a alterar el orden total. Supongamos que tenemos n barcos y m lanzaderas.

El loop principal se hace a lo sumo m veces, ya que en cada iteración se hace al menos un disparo (como es visible en el código). En cada iteración se recorren a lo sumo n barcos y se hace una serie de operaciones de acceso y/o actualización. Entonces, el orden total del algoritmo es:

$$O(nm)$$

1.3.3. ¿Cuando es óptimo?

Hay diversas condiciones, bajo las que este algoritmo puede ser el óptimo. Algunas de las que encontramos son:

- Hay un solo barco (cualquier algoritmo que dispare a un barco con vidas es óptimo en este caso), o hay una sola posición posible para cada barco.
- El daño a impactar a cada barco es el mismo en cada turno, y la vida inicial es la misma. En este caso el algoritmo siempre va a hundir los barcos uno por uno, en orden.
- Los daños posibles están repartidos de tal manera que en cada turno el algoritmo greedy siempre encuentra barcos a hundir (todos los disparos se utilizan para hundir barcos). En este caso, como el algoritmo siempre elige el barco a hundir en menos disparos, esta acción es la óptima.

1.4. Estrategia Dinámica

1.4.1. Resolución teórica

Para la estrategia dinámica se plantearon varios algoritmos. El que mejores resultados dio tiene la siguiente idea como base: dada la posición actual de los barcos, el mejor conjunto de disparos es el que minimiza la cantidad de puntos logrados en la posición actual y el de los disparos consecutivos (alterando la posición por los disparos elegidos). Entonces, los disparos elegidos por este algoritmo van a ser los óptimos, porque serán los que minimizan exactamente lo pedido en el enunciado.

Esta idea se puede plantear de la siguiente forma: supongamos que n es nuestra posición actual, d es un conjunto de disparos (con tantos elementos, cuantas lanzaderas tengamos), y $f(n, d)$ es una función que dada la posición n y disparos d cuantos puntos va a obtener el jugador del equipo contrario. Supongamos que D son todos los disparos que podemos hacer y $n + 1$ es la posición en la que vamos a quedar si impactamos disparos d_i. Podemos plantear la siguiente recurrencia:

$$OPT(n, d) = f(n, d) + min(OPT(n + 1, d_i) \forall d_i \in D)$$

Necesitamos agregar una condición de corte: si un set de disparos resulta en que el contrincante no gane puntos, es el set de disparos óptimo, es decir, si $f(n, d) = 0$ entonces $OPT(n, d) = 0$.

La llamada inicial (la del primer turno) de esta forma sería:

$$PrimerTurno(n) = min(OPT(n, d_i) \forall d_i \in D)$$

1.4.2. Resolución práctica

Hay una serie de pasos que queda definir antes de pasar el planteo teórico a código:

1) **¿Qué es la posición?**

La posición no puede ser simplemente la posición actual de cada barco, depende también de vida de cada barco en cada momento dado. Se va a necesitar una función (llamémosla `obtener_posicion`) que dado el tablero y el estado actual de los barcos, devuelva una estructura que describa unívocamente la posición de los barcos. En nuestro código esa función devuelve una cadena de texto que por cada barco tiene su vida y su posición actual. Esto nos va a permitir hacer uso de *memoización* en nuestro algoritmo dinámico, porque un barco puede estar en una posición particular con una vida particular más de una vez durante el juego, pero el resultado óptimo va a ser el mismo en cada caso.

2) **¿Cuál es el conjunto de disparos?**

El conjunto de disparos posibles está definido por la cantidad de barcos y la cantidad de lanzaderas. Es más correcto ver ese conjunto como *conjunto de impactos*, porque si un barco es impactado por un disparo, no tiene relevancia si le disparó la lanzadera 1 o 2. Por ende el disparo `(barco1, barco2)` es exactamente igual a `(barco2, barco1)` y los queremos tratar como uno solo. En el código, tenemos una función `posibles_disparos` que devuelve los disparos posibles sin ese tipo de repetidos.

3) **¿Cual es la función `f`?**

La función `f` del planteo teórico es la función, cuyo resultado se quiere minimizar. En nuestro algoritmo, es la función que debería decir dado el estado del juego y los disparos a hacer, cuantos puntos va a recibir el contrincante después de aplicar esos disparos. En el código de nuestro algoritmo es la función `resultado_de_disparos`, y devuelve la cantidad de barcos vivos después de disparar. Devuelve un segundo valor que se va a explicar en la parte de Optimizaciones.

El algoritmo práctico se puede ver en el archivo `estrategia_dinamico.py`, clase `EstrategiaDinamico`. Su funcionamiento se puede resumir en el siguiente pseudocódigo:

```
def siguiente_turno(tablero, cantidad_lanzaderas, barcos):

    posicion = obtener_posicion(tablero, barcos)

    if not resultado:
        resultado: = {}

    if posicion not in resultado:
        minimo: = infinito
        for disparos in disparos_posibles:
            k: = optimo_recursivo(posicion, disparos, tablero, barcos)
            if k < minimo:
                minimo := k
                resultado[posicion] = disparos
```

```
    return resultado[posicion]


def optimo_recursivo(posicion, disparos, tablero, barcos):

    if M[posicion, disparos]:
        return M[posicion, disparos]
    else:
        m := f(disparos, barcos, tablero)
        if m == 0:
            M[posicion, disparos] := 0
            resultado[posicion] := disparos
        else:
            barcos = copiar(barcos)
            M[posicion, disparos] := maximo
            resolver_turno(disparos, barcos, tablero)
            nueva_posicion := obtener_posicion(barcos, tablero)
            for disparos_nuevos in disparos_posibles:
                k := optimo_recursivo(posicion, disparos, tablero, barcos)
                if m + k < M[posicion, disparos]:
                    M[posicion, disparos] := m + k
                    resultado[nueva_posicion] := disparos_nuevos

    return M[posicion, disparos]
```

Se mantienen dos estructuras para memoizar los resultados que se van calculando recursivamente. En la matriz `M` en cada punto (p_i, d_j) se va guardando la mínima cantidad puntos que se pueden lograr si se elige realizar disparos d_j en la posición p_i. En el diccionario `resultado` en cada posición p_i se guardan los disparos d_k que minimizan la cantidad de puntos a lograr por el contrincante.

Como a medida que se hacen llamados recursivos, se altera el estado del juego (la estructura mutable `barcos`, que mantiene la vida y la posición de los barcos), y en Python las estructuras mutables se pasan por referencia, es necesario hacer una copia de dicha estructura antes de realizar cambios sobre ella (para no alterar estado en otros llamados recursivos).

La función `resolver_turno` aplica los disparos en cuestión, y luego mueve los barcos una posición hacía delante.

1.4.3. Análisis de complejidad

Supongamos que en nuestro juego participan n barcos y m lanzaderas. La cantidad de disparos a hacer es igual a la cantidad de lanzaderas (m). Todos los datos se guardan en estructuras con tiempo de acceso $O(1)$, asique vamos a despreciar los accesos en este análisis. Por cada llamado recursivo se hace lo siguiente:

- Llamado a `f` que implica recorrer una vez barcos, y una vez lanzaderas: $n + m$
- Copia de barcos: n
- Llamado a `resolver_turno`, que recorre disparos y luego recorre barcos: $n + m$
- `obtener_posicion` recorre una vez barcos: n

Orden de cada llamado recursivo de esta forma es $O(4n + 2m) = O(n + m)$.

Ahora, ¿cuántos llamados recursivos se hacen? Por cada posición se prueban todos los disparos posibles, sin repeticiones, llamemos a este número D. El número de estas combinaciones D se puede calcular como:

$$D = \binom{n+m-1}{m}$$

Lo que queda ver es cuántas posiciones va a recorrer el algoritmo. Lamentablemente, no encontramos una cota exacta. Una cota aproximada superior se puede hallar calculando la cantidad de todas las posiciones posibles, llamemos ese número P. Conocemos los posibles daños que se pueden inferir a un barco. Podemos estar seguros de que la cantidad de vida que puede llegar a tener un barco en cualquier posición es un múltiplo del *máximo común divisor (mcd)*, entre los posibles daños que se pueden impactar a ese barco. Ese número siempre va a ser menor que la vida inicial del barco V_M. Supongamos que los daños posibles son el conjunto Q, podemos calcular la número de diferentes valores de vida del barco como $V_M/mcd(Q)$. Ese valor lo tenemos que multiplicar por la cantidad de posiciones posibles del barco, que es igual a $|Q|$. Finalmente tenemos que calcular ese número por cada barco i:

$$P = \sum_{i=1}^{n} V_{M\,i} * |Q_i|/mcd(Q_i)$$

Juntando esas cuentas, podemos estar seguros de que el orden de nuestro algoritmo siempre va a ser menor que:

$$\sum_{i=1}^{n} V_{M\,i} * |Q_i|/mcd(Q_i) * \binom{n+m-1}{m} * (n+m)$$

Este es el orden de cálculo de los disparos óptimos a partir de una posición inicial. Una vez calculado, cada turno se resuelve en $O(1)$.

1.4.4. Optimizaciones

El orden calculado en la sección anterior crece extremadamente rápido y si el algoritmo realmente tiene que hacer esa cantidad de ejecuciones, se va a volver inaplicable salvo para algunos inputs muy pequeños. Para mejorarlo se aplicaron algunas optimizaciones:

- En el juego, un barco puede tener una vida menor a 0 (puede ocurrir si el daño impactado es mayor a la vida actual del barco). Para no considerar posiciones con vida negativa, en el cálculo de posición todas las vidas negativas se convierten a 0.
- Usando la misma lógica, si un conjunto de disparos no causa daño, el llamado recursivo detecta este caso y devuelve que los puntos logrados en ese caso son infinitos (para asegurarse que este conjunto de disparos no se elige como el óptimo). Para eso la función `resultado_de_disparos` devuelve un segundo parámetro con la cantidad de daño impactada.
- Si uno de los disparos de un conjunto de disparos va a un barco que no tiene vidas en este momento, seguro que ese conjunto de disparos no es el óptimo (porque a lo sumo puede ser igual que el óptimo, en el caso de que los otros disparos causen el mismo daño que en un set de disparos óptimo). Para ignorar

esos casos, antes de cada llamado recursivo se hace un checkeo con la función `disparos_optimos`, que si detecta que es el caso, no se hace un llamado recursivo. Eso adiciona un recorrido de barcos (n), pero mejora la performance global.

- Todas las estructuras elegidas tiene costo de acceso O(1).

Si bien el resultado de este algoritmo es el óptimo, y en todos los casos performa igual o mejor que su contraparte greedy, el orden del algoritmo es muy alto, y solo es aplicable para tableros no muy grandes. Por ejemplo, para la definición del tablero del archivo `tablero2`, que tiene 4 barcos, con 5 posiciones posibles cada uno, y un *mcd* $\sim$ *10*, usando 3 lanzaderas el algoritmo tarda unos 5 segundos en la máquina que se usó para desarrollo.

1.5. Algoritmo de posicionamiento

El algoritmo para la estrategia dinámica también se puede usar para seleccionar la mejor posición. Se pueden recorrer las posiciones iniciales posibles, y consultar usando el mismo algoritmo.

El pseudocódigo sería el siguiente, suponiendo que la función `optimo_recursivo` es la explicada en la estrategia dinámica:

```
def mejor_posicion(tablero, cantidad_lanzaderas, barcos):
    minimo := 0
    mejor_posicion := null
    for posicion in posiciones_posibles:
        for disparo in disparos_posibles:
            k: = optimo_recursivo(posicion, disparos, tablero, barcos)
            if k > minimo:
                minimo := k
                mejor_posicion = posicion
    return mejor_posicion
```

Como `optimo_recursivo` hace uso de memoización internamente, el orden no debería crecer aún más, asique el algoritmo de mejor_posición debería tener el mismo orden.

2. Parte 2: Sabotaje!

2.1. Consigna

Son agentes secretos de la organización C.O.N.T.R.O.L y reciben una información anónima que los pone en alerta: La organización K.A.O.S planea sabotear la red secreta de transporte de información. Hace unos días se reportó el robo de una copia del mapa maestro de la red. Dan por supuesto que el robo fue de sus rivales y que ellos conocen las debilidades de su red. Siendo que únicamente tienen personal suficiente para vigilar 2 ejes de su red, que únicamente se pueden sabotear ejes (y no vertices) y que los saboteadores quieren hacer el máximo daño posible:

- Diseñe un algoritmo genérico que funcione con cualquier tipo de red que determine qué ejes vigilar. Preséntelo con un pseudocódigo, explique su funcionamiento y si el mismo además es óptimo.
- Analice la complejidad del mismo. De igual manera analice si utiliza alguna reducción.
- Programe el algoritmo.
- Si es necesario programe el algoritmo para determinar el flujo máximo. Utilícelo para calcular el de la red antes y después de los posibles sabotajes.
- En una situación real existirán varias fuentes y varios sumideros. Explique y analice una forma de resolver el mismo problema en este caso.

Red Secreta:

El mapa de la red se debe obtener de un archivo llamado “redsecreta.map” con el siguiente formato:

- Cada línea del archivo corresponde a un eje de la red que une 2 vértices y su capacidad
- Los ejes están etiquetadas por números enteros
- La capacidad son números enteros.
- La fuente estará etiquetada como el número 0 (cero)
- El sumidero estará etiquetado como el número 1 (uno)

2.2. Análisis del problema

Tenemos nodos, aristas y cada una de ellas tiene una capacidad máxima. La idea de la resolución radica en calcular el flujo máximo que pasará por la red dada como entrada según las especificaciones del ejercicio. Una vez que está calculado el flujo máximo, se toma la arista por la que pasa mayor cantidad de información. Se quita esa arista y se vuelve a calular el flujo máximo. Nuevamente se toma la arista cuyo flujo es máximo. En esas dos aristas es en donde se pondría el personal para vigilar la red. El pseudocódigo del algoritmo es el siguiente: Para el pseudocódigo definimos: $C(u, v)$ como capacidad de la arista que une los vértices u y v en el grafo original. $G_f(u, v)$ como capacidad de la arista que une los vértices u y v en el grafo residual. $flujo_maximo_arista(u, v)$ tiene cuanto flujo pasa por esa arista

```
para cada arista (u,v) de G:
    G_f(u,v) = 0
    G_f(v,u) = 0

mientras exista un camino p desde la fuente al sumidero en el grafo residual:
    c_f := se toma minimo {G_f(u,v) : (u, v) en camino p}
    para cada arista (u,v) de p:
```

```
        G_f(u, v) -= c_f
        G_f(v, u) += c_f
        flujo_maximo_arista(v,u) = G_f(v,u)

ordenar de menor a mayor los valores de flujo_maximo_arista para todo (u,v)
tomar el ultimo
```

2.3. Análisis de complejidad

Para este problema se utilizó una reducción utilizando un algoritmo ya conocido que es el de Ford-Fulkerson. La complejidad de este algoritmo es $O(Ef)$, donde E es la cantidad de aristas y f es el flujo máximo en el grafo. Esta justificación se da porque cada camino de aumento puede ser encontrada en un tiempo $O(E)$ e incrementa el flujo por una cantidad entera de a lo sumo 1 con una cota máxima de f. Los caminos de aumento busca caminos de fuente a sumidero que sean acíclicos. La complejidad del algoritmo de caminos de aumento puede ser muy mala si se eligen mal los caminos, puede ser del orden de la capacidad máxima, la cual puede ser exponencial en el tamaño de la entrada. Existe una variación de Ford-Fulkerson que garantiza terminación y un tiempo de ejecución independientemente del valor del máximo flujo. Este algoritmo se llama Algoritmo de Edmonds-Karp. La complejidad de dicho algoritmo es $O(VE^2)$ donde V es la cantidad de vértices y E la cantidad de aristas. En este trabajo se utilizó esta variante.

2.4. Múltiples fuentes y múltiples sumideros

Podemos transformar el problema de múltiples fuentes y sumideros a un problema conocido de una fuente y un sumidero. Para cada una de las fuentes originales podemos conectarla con una arista de capacidad infinita (o en terminos programáticos un número muy grande) a una nueva fuente. De manera análoga hacer lo mismo con los sumideros. De esta manera podemos implementar lo que ya estudiado y analizado para un caso mucho más real.

3. Instalación y ejecución

El juego se desarrolló y se probó con Python 3.5

3.1. Parte 1

Desde la carpeta `parte1` se ejecuta de la siguiente manera:

```
python TP3-1.py <tablero> <lanzaderas> <estrategia>
```

Donde:

- `<tablero>` es ruta al archivo del tablero
- `<lanzaderas>` es un el número de lanzaderas. Tiene que ser un número entero.
- `<estrategia>` es el nombre de la estrategia a usar. Puede ser `naive`, `greedy` o `dinamico`.

Ejemplo de ejecución:

```
python TP3-1.py tablero3 3 dinamico
```

3.2. Parte 2

Desde la carpeta `parte2` se ejecuta de la siguiente manera, pasando una ruta a un archivo de red secreta:

```
python TP3-2.py <archivo de red secreta>
```

Por ejemplo:

```
python TP3-2.py redsecreta2.map
```