

Ampliación de Programación Memoria de Prácticas

Raúl Reguillo Carmona



Índice

1. Primera Práctica: Jon Nieve	4
1.1. Enunciado	4
1.2. Planteamiento	4
1.3. Codificación	9
1.4. Complejidad	10
1.5. Conclusiones	10
2. Segunda Práctica: Daenerys	12
2.1. Enunciado	12
2.2. Planteamiento	12
2.3. Estrategia Voraz	15
2.3.1. Primera heurística	16
2.3.2. Segunda heurística	16
2.3.3. Sobre tentativas basadas en teoría de grafos	16
2.3.4. Definición del algoritmo	17
2.4. Estrategia Dinámica	17
2.5. Estrategia Backtracking	18
2.6. Conclusiones	19
3. Tercera Práctica: Juegos	21
3.1. Enunciado	21
3.2. Planteamiento	21
3.2.1. Nota sobre la implementación	21
3.2.2. Codificación	21
3.3. Adaptación para juego vs COM	22
3.4. Algunas consideraciones	24
3.5. Conclusiones	25
4. Consideraciones	26
4.1. Ejecución de los programas	26

Listings

1.	Función principal	10
2.	Declaración de constantes	13
3.	Lectura del archivo de configuración	13
4.	Estructura del algoritmo dinámico	18
5.	Control mediante backtracking	18
6.	Pseudocódigo de la funcion	21
7.	Generación de objetos	24
8.	Compilación y ejecución	26

Índice de figuras

1.	Ilustración de los tres posibles casos	6
2.	Eliminar el mínimo de elementos de cada lado	7
3.	Casos básicos del punto 3. <i>c</i>	8
4.	Algunos casos básicos del punto 3. <i>e</i>	9
5.	Grafo del problema	15
6.	Salida del programa	20
7.	Muestra del programa	23
8.	Muestra del programa (II)	24
9.	Estado de conflicto	25

1. Primera Práctica: Jon Nieve

1.1. Enunciado

Jon Nieve, ascendido a oficial, está al mando de un grupo de N soldados ordenados por edad. Ha ideado una descabellada formación militar que consiste en defender en fila india siguiendo dicho orden. Su objetivo es mentir sobre su edad para colocarse en la posición central de la fila (la posición más segura en una posible batalla). Sin embargo la estrategia no es el único punto débil de Jon Nieve, la algoritmia también y Jon tiene un problema. Ha llegado un nuevo grupo de M soldados que también tenemos ordenados por edad. Jon Nieve necesita saber qué edad debe decir que tiene para colocarse en la posición central de la fila ordenada de soldados que resultaría si se mezclasen ambos grupos. La complejidad debe ser estrictamente menor de $O(N+M+1)$.

1.2. Planteamiento

La solución consiste en hallar el número que debería quedar entre medias de los vectores después de mezclarlos. Esto es hallar la **mediana** de los dos vectores luego de ordenarlos. Pero para ello no podemos mezclarlos previamente porque sobrepasamos la complejidad. Se supone que los vectores originales vienen previamente ordenados, es decir, son no decrecientes.

Se pide un algoritmo *Divide y Vencerás* que para el problema en particular es una estrategia que se ajusta a la perfección. Se procederá por partes:

1. Determinar dónde buscar: lo que es equivalente a descartar elementos de los vectores originales. El modo de decidir qué elementos se quedan fuera y cuáles no se calcula en base a las medianas de los vectores originales. Supongamos el vector de N elementos, cuya mediana es el elemento x así como el vector de M elementos cuya mediana es el elemento y . Podemos tener lo siguiente:
 - $x < y$: lo que implicaría que la solución debe ser un número comprendido en la mitad *superior* del vector N y la mitad *inferior* del vector M
 - $x > y$: lo que implicaría que la solución debe ser un número comprendido en la mitad *inferior* del vector N y la mitad *superior* del vector M
 - $x = y$: Hemos dado con una solución, pues si las medianas son iguales, seguirán siéndolo tras mezclar los dos vectores
2. Descartar elementos: Una vez sabemos dónde estará el número que buscamos, lo que hay que hacer es descartar donde no nos interese buscar. No obstante hay que tener

precaución, puesto que si se descarta alegremente es posible que **se mueva la mediana del vector resultante**, así pues hay que descartar siempre **el mismo número de elementos de los vectores**. El número siempre deberá ser el más restrictivo, esto es, el menor, pues nunca podemos descartar elementos si son candidatos a ser una posible solución.

3. Continuar recursivamente hasta tener uno de los siguientes casos base:

- a) $|N| = |M| = 1$: Los vectores de longitudes N y M (en adelante, vector N y vector M respectivamente) se han ido reduciendo de esta manera hasta que ambos cuentan con únicamente **un elemento**. En este caso, la solución es la **media de ambos elementos**.
- b) $|N| = 2, |M| = 1$: En este caso hay que evaluar en qué posición se ha de ubicar el símbolo solitario. El elemento central después de colocarlo, será un resultado válido.
- c) $|N| = |M| = 2$: Pueden darse una serie de combinaciones para este caso
 - N esté contenido en M : La solución será la media de los dos elementos de N
 - M esté contenido en N : homólogo al anterior
 - N sea anterior estricto a M : la solución será la media entre el mayor elemento de N y el menor de M
 - N sea mayor estricto a M : en este caso la solución será la media entre el menor de N y el mayor de M
 - N sea anterior y esté superpuesto con M : con lo cual la media entre los dos elementos centrales será solución
 - N sea posterior y solape a M : es el mismo caso
- d) $|N| = 1, |M| > 2$: El planteamiento para este caso sigue siendo igual. Sin embargo hay que tener la suficiente precisión como para ubicar el elemento independiente en su posición, puesto que así podremos hacer cálculos de manera más precisa. No es lo mismo que el elemento esté en uno de los extremos y con lo cual no necesariamente intervenga en el cálculo de una edad media que se ubique cerca de la mediana del vector M y sí que participe. También es importante saber si es mayor o menor que la mediana del vector M , pues así elijiremos qué dos números computarán la solución media.
- e) $|N| = 2, |M| > 2$: Aunque parezca un caso redundante, es de vital importancia para el desarrollo. Existe una serie de casos particulares que se centran en este caso base. Mucho depende de la paridad o no de los vectores originales así como

el de mayor longitud una vez llegamos a este punto, es por eso por lo que a veces podría fallar si no se tiene en cuenta este particular. El planteamiento es igual a los casos anteriores (ver apartado *c*). Se puede comprobar y de hecho intuir que los casos de solapamiento siempre serán los más numerosos, es por eso por lo que debemos tener especial cuidado al controlar estas situaciones.

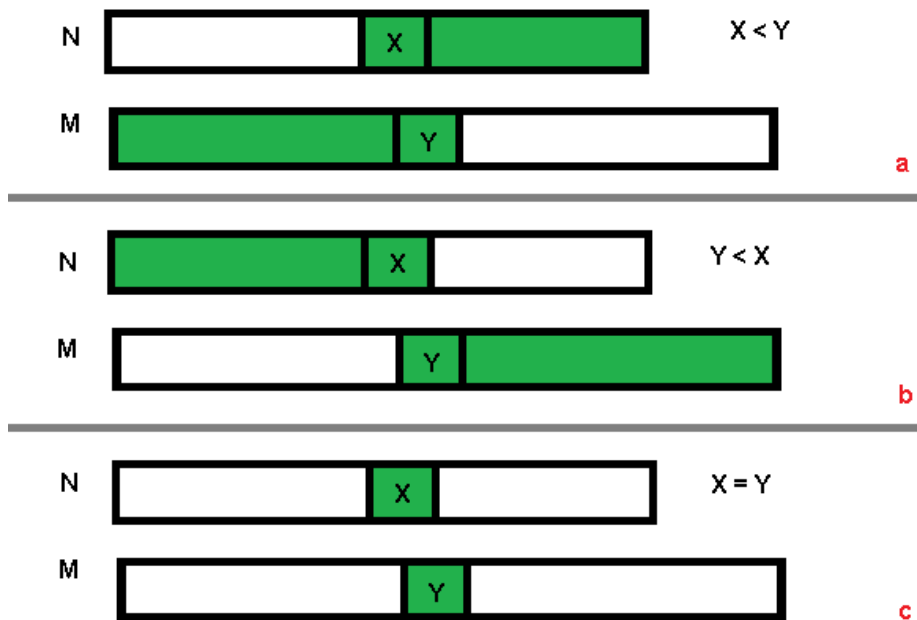


Figura 1: Ilustración de los tres posibles casos

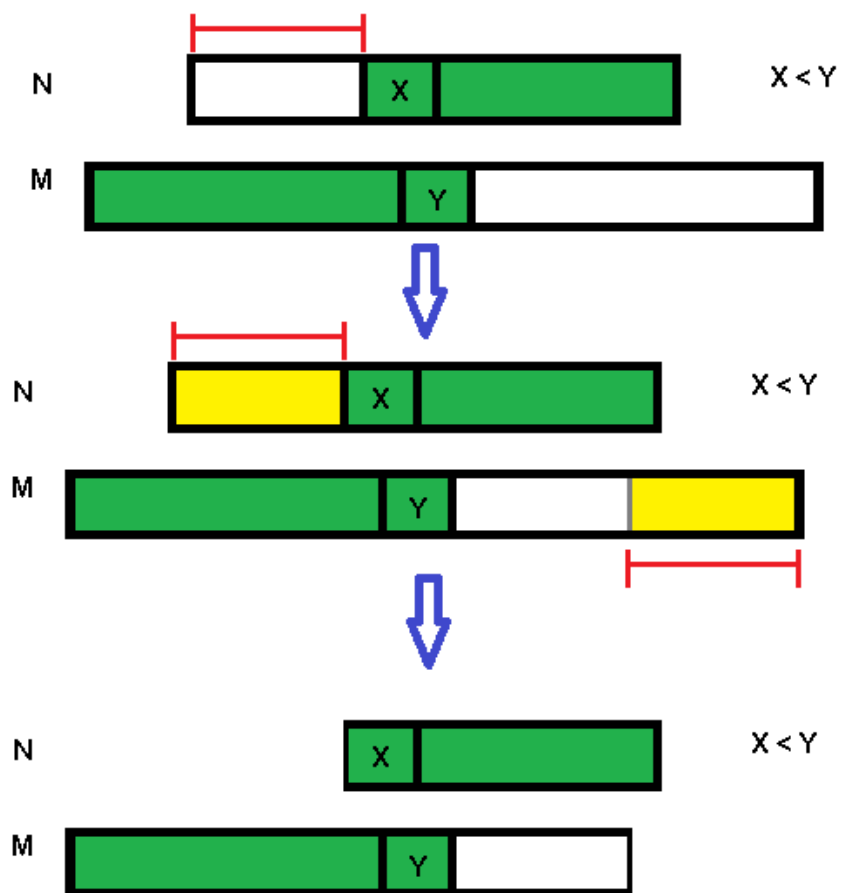


Figura 2: Eliminar el mínimo de elementos de cada lado

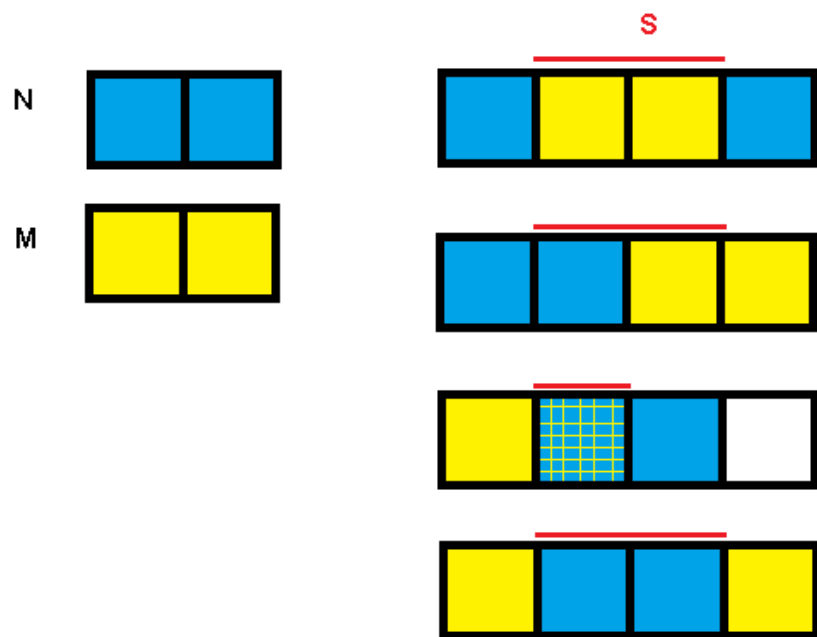


Figura 3: Casos básicos del punto 3.c

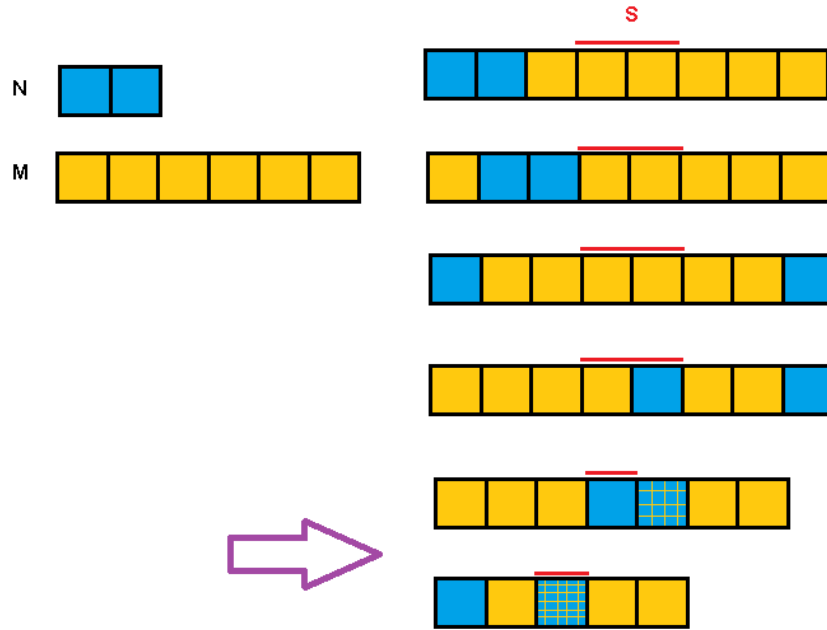


Figura 4: Algunos casos básicos del punto 3.e

1.3. Codificación

A la hora de transcribir a código, se ha dividido la tarea en varias secciones:

- Generar los vectores de manera aleatoria
- Ordenarlos mediante *Quicksort*
- Una vez ordenados, operar con ellos según el método visto anteriormente.

Además, se han incluido toda suerte de funciones necesarias para mostrar por pantalla el estado actual, la solución final calculada de manera iterativa (para comparar con el resultado obtenido), calcular medias, máximos, mínimos, etc.

La principal dificultad en este caso radica en controlar bien los índices de los vectores para no provocar desbordamientos o captura de datos erróneos. A continuación se expone un pequeño pseudocódigo que relata el funcionamiento del método principal:

Listing 1: Función principal

```
int years(int X[], int Y[], int lix, int lsx, int liy, int lsy){
    SI vectores concuerdan con caso base THEN
        CALCULAR RESULTADO SEGUN CASO
        return RESULTADO

    ELSE
        //CASO GENERAL
        Hallar medianas de N y M
        Comparar
        Escoger minimo numero de elementos a eliminar
        Actualizar limite superior e inferior de vectores
        Llamar recursivamente a years()
}
```

1.4. Complejidad

Es fácil ver que la complejidad es siempre menor que la indicada, puesto que buscará siempre primero en los casos base (lo que se reduce a orden 1) y de no encontrar, llamará recursivamente tras operar. El número de llamadas será siempre menor estricto que la suma de N y M. De esta manera se confirma que no nos hemos excedido.

1.5. Conclusiones

A la hora de resolver este problema, existen dos factores primordiales para encarrilar la solución:

1. Percatarse ante todo que con lo que se debe trabajar son medianas de vectores. Es importante el concepto de dejar a ambos lados del elemento clave la misma cantidad de elementos (en caso de ser impar) o identificar los dos elementos centrales para hallar una media entre ellos (si el vector original es de longitud par). Subyace a esto la idea intuitiva de combinar de alguna manera ambos vectores valiéndonos de sus respectivas medianas como referencia. Lo que nos lleva al siguiente punto.
2. Un segundo y erróneo paso puede ser el de descartar las mitades de los vectores que más allá de la mediana no nos sirven. Lo crucial en este problema es mantener siempre la mediana centrada y eso se consigue eliminando, en cada vector, el mismo

número de elementos del lado correspondiente. Llegada a esta situación, sólo queda estudiar concienzudamente los casos base que hemos tratado, controlar la paridad de los vectores y ser precisos con los índices que marcan límites superiores e inferiores de cada uno de los vectores.

El resultado es un programa eficiente que en pocos pasos (generalmente en menos de cuatro llamadas recursivas) resuelve el problema.

Sobre el tiempo de resolución es muy dependiente de las **longitudes** de los vectores y sus **proporciones** relativas. Se pueden identificar así los siguientes casos:

- Cuando los vectores son **similares en tamaño** el algoritmo los va reduciendo de manera continua y similar, pudiendo converger a la solución en unas cuantas llamadas recursivas al no alcanzar rápido los casos base (longitudes 1 y 2 respecto a un tamaño mayor).
- Cuando uno de los vectores es **mayor respecto del otro**, la convergencia es mayor. Si bien, el número de elementos que se eliminan de cada vector es menor, pues está condicionado por el vector más pequeño siempre, siempre se alcanza un caso base muy rápido que es cuando el vector más corto llega a longitud 2 ó 1.
- Cuando el **tamaño de los vectores es muy grande**, la solución se suele encontrar en un único paso. Esto es debido a que la generación de valores de cada vector está restringida a un rango acorde al problema (en estos mundos de fantasía, uno ya es un hombre con 15 ó 16 años y un anciano en torno a los 30 si es que acaso sigue con vida, así que se escogió este rango de edad para representar el problema). Si se generan muchos números dentro de este rango, tienden a repetirse, una vez ordenados, en franjas. Cuanto más grande sean esas franja de edad, es más probable que se solapen ambos vectores en la franja que coincida con la edad central y por lo tanto más posibilidades hay de que se alcance el primer caso base: si las medianas coinciden, hay solución.

2. Segunda Práctica: Daenerys

2.1. Enunciado

Daenerys, noble heredera de la casa Targaryen, se ha propuesto recuperar el poder en los siete reinos y para ello debe conseguir el apoyo de N pueblos guerreros de la región del Sur. Todos y cada uno de los pueblos están unidos por áridos caminos y peligrosas rutas marítimas, de los que Daenerys conoce el tiempo en días necesario para recorrerlos (por cada par de pueblos i, j se conoce el tiempo T_{ij} necesario para llegar de i hasta j). Daenerys tiene un innato poder de convicción (véase, tiene unos temibles dragones) que le permite obtener una cantidad determinada de oro (R_i) y unas unidades de ejército (U_i) de cada uno de los pueblos. Sin embargo, cada día debe gastar una cantidad concreta de oro S en víveres por cada unidad de ejército a sus órdenes. Daenerys comienza su aventura desde su campamento inicial, con una cantidad de X monedas de oro y sin ejército a sus órdenes. En todos los casos debe también acabar la aventura en su campamento.

- **Variante 1:** Daenerys necesita saber en qué orden debe visitar todos y cada uno de los pueblos tardando el menor tiempo posible y sin quedarse sin oro para pagar a su ejército.
- **Variante 2:** Daenerys necesita saber en qué orden debe visitar todos y cada uno de los pueblos para maximizar el oro tras conseguir todas las unidades de ejército disponibles.
- **Variante 3:** La codicia ha podido con Daenerys y su objetivo es obtener la mayor cantidad de oro sin importarle el tiempo ni las unidades de ejércitos conseguidas, ya que cuando reúna lo máximo posible, se fugará al paraíso fiscal de las islas de hierro. En este caso no tiene por qué visitar todos los pueblos; eso sí, cada vez que recauda oro de un pueblo y decida visitar otro, debe llevarse su ejército al siguiente pueblo y pagar sus víveres) para no levantar sospechas.

2.2. Planteamiento

Dependiendo de la estrategia, el planteamiento puede variar. Es por ello que se particularizará cada planteamiento según la estrategia escogida, pues el método es distinto siempre. Es por eso por lo que no se puede dar una regla general, sino que los planteamientos se deben idear según los casos.

No obstante sí se pueden llevar a cabo una serie de buenas prácticas que facilitan, sobre todo al ahora de programar, el diseño de la aplicación.

En primer lugar, es importante seguir una numeración estándar que cubra todos los casos. Para ello se ha declarado de la siguiente manera:

Listing 2: Declaración de constantes

```
#define CAMP 0
#define BHORASH 1
#define MEEREEN 2
#define YUNKAI 3
#define ASTAPOR 4

#define GOLD 1
#define UNITS 2

#define TRUE 1
#define FALSE 0

#define RESERVED -2
#define INFINITE 9999

#define INPUTFILE "src/P2_DaenerysInput.txt"
#define PLACES 4
```

De este modo, cuando accedamos a la matriz de distancias en la posición [1][2] o bien la [2][1], lo que estaremos haciendo es recuperar el dato de la distancia entre Bhorash y Meeren. En el array de recursos y unidades, la posición [1][GOLD] y [1][UNITS] devolverán los datos económicos y militares de dicha ciudad.

Por otro lado, `INPUTFILE` nos guardará la ruta del archivo de entrada. Este archivo se lee de manera sencilla y se guardan sus datos en estructuras que cumplen con esta norma:

Listing 3: Lectura del archivo de configuración

```
fichero = fopen(INPUTFILE, "r");

fscanf(fichero, "N\t%d\n", &N);
fscanf(fichero, "S\t%d\n", &S);
fscanf(fichero, "X\t%d\n", &X);

for(i=0; i<=PLACES; i++){
```

```

    for (j=0; j<PLACES; j++)
        fscanf(fichero , "%d\t" , &distancia[i][j]);
        fscanf(fichero , "%d\n" , &distancia[i][j]);
    }

    RnU[0][0]=0;
    RnU[0][1]=0;
    RnU[0][2]=0;

    for (i=1; i<=PLACES; i++)
        fscanf(fichero , "RU%d\t%d\t%d\n" , &RnU[i][0] , &RnU[i][1] , &RnU[i][2]);

    fclose(fichero);

```

Como se puede apreciar, se debe cuadrar una fila en el arreglo `RnU` (resources and units) para que la fila cero no tenga datos (realmente el campamento no tiene datos).

Siguiendo estos convenios e implementando funciones que nos faciliten la existencia (tales como para imprimir tipos predefinidos o mostrar el estado en un momento), podemos centrarnos en el problema.

Debemos tener en cuenta que nos enfrentamos a un problema variante del famoso *viajante de comercio*. Hay que plantearse que en este caso al viajante le cuesta viajar más allá del tiempo, es decir, existen dos variables de control, lo que cambia bastante el problema. Sin embargo seguimos contando con un grafo, fuertemente conexo, lo que no condiciona la elección de ruta al no tener que reservar *a priori* ningún nodo o secuencia de nodos para el final.

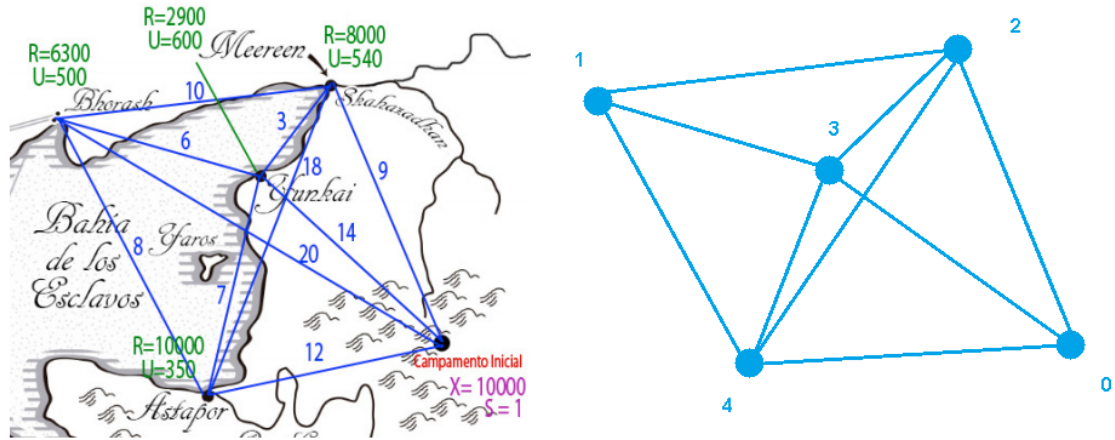


Figura 5: Grafo del problema

2.3. Estrategia Voraz

Una estrategia voraz no requiere de recorrer todas las posibilidades. Únicamente nos hace falta tener una heurística que optimice en la medida de lo posible el resultado final obtenido, eligiendo paso tras paso.

Intentar antever un par de pasos más allá de la propia elección sería romper en cierta medida el esquema que requiere una estrategia voraz.

Consideremos lo siguiente:

- La acumulación de tropas: medida que vamos avanzando, nodo tras nodo, el número de tropas crece. Esto significa que cada paso que demos será más caro que el anterior.
- Longitud del camino: Un camino largo puede ser peligroso para nuestra economía se se carga con demasiadas tropas. Subyace la idea de que quizá sea una buena idea realizar los caminos más largos antes, pero esto comienza a atentar contra los principios de un voraz.

Partiendo de esta base, lo que tenemos ya es la primera de las heurísticas:

2.3.1. Primera heurística

Puesto que al final contaremos con el total de tropas, el desgaste será mayor si el último salto que se de sea de mayor longitud, es por eso por lo que **se debe dejar el camino más corto de vuelta al campamento para el final.**

Teniendo esto, el problema se reduce a hallar el ciclo Hamiltoniano que comience en el campamento y acabe en la ciudad más próxima a éste.

Pero ¿cómo recorremos el resto de vértices?. Hay tres factores que juegan un papel determinante en este punto:

- El oro botenido al llegar a cierta ciudad: G
- Las tropas que obtendremos al llegar a dicha ciudad: T
- La distancia que nos separe de esta ciudad: L

Es aquí donde entra en juego la segunda de las heurísticas.

2.3.2. Segunda heurística

El factor más atractivo a la hora de ir a cierta ciudad es la cantidad de oro que de ésta podamos extraer, en lo que se refiere a todas las variantes. Sin embargo, una ciudad muy alejada o una ciudad que nos cargue con demasiadas tropas puede lastrar nuestra economía. Así pues parece razonable decidir a qué ciudad viajar mediante el cálculo del siguiente cociente:

$$\frac{G}{T \cdot L}$$

De esta manera obtendremos el cociente más interesante en cada caso para seleccionar un destino en un paso.

Esta estrategia ha dado muy buenos resultados en las dos variantes implementadas (la tercera, debido a su naturaleza, se ha dejado por cuestiones de tiempo) obteniendo siempre el resultado óptimo para el caso particular.

2.3.3. Sobre tentativas basadas en teoría de grafos

Se quiso avanzar en teoría de grafos con una serie de heurísticas que pudiesen facilitar la búsqueda de candidatos a la hora de expandir el camino. En particular, con lo siguiente:

Se dice que un grafo tiene *propiedad métrica* si dado un par de vértices i, k , cuya distancia de separación es IK , existe un nodo j tal que $IK + KJ \leq IK$, se escogerá este camino: $i \rightarrow j \rightarrow k$. No obstante para una estrategia voraz, esta regla rompe con la filosofía de la misma y por idénticas razones no es aplicable a backtracking ni dinámica al ser éstas estrategias de recorrido completo, por lo que no se debe aplicar al descartar soluciones.

2.3.4. Definición del algoritmo

En definitiva, la solución voraz se compone de:

1. **Elementos candidatos:** Los compondrán los nodos no visitados (excluyendo el que guarde la longitud más corta al campamento, por la primera heurística).
2. **Elementos seleccionados:** Se guardarán los nodos por los que se vaya pasando a medida que se construye la solución.
3. **Función solución:** Comprobar si hemos visitado todo el grafo.
4. **Criterio de factibilidad:** Un nodo será factible si no ha sido visitado y si visitarlo no supone llegar a una cantidad negativa de oro.
5. **Función de selección:** Según la segunda heurística, el índice calculado será el oro recibido en esa ciudad, dividido por el producto del número de tropas y la distancia que nos separa.
6. **Función objetivo:** Según variantes. Menor tiempo posible, o maximizar oro.

2.4. Estrategia Dinámica

Una estrategia dinámica presenta la ventaja de la completitud, pero obliga al recorrido de todo el árbol. Además la complejidad para su implementación invita a programar durante largas horas un algoritmo que no es intuitivo y es complejo de comprender.

En la estrategia dinámica se construye la solución de una manera inversa. Explorando todos los nodos, paso a paso, vamos conservando una solución (la mejor en cada nivel) para pasarla al final como óptima.

Haciendo una adaptación del backtracking, ha sido una implementación particularmente difícil debido a la precisión con la que había que calcular los datos y controlar las estructuras. Al realizar cálculos intermedios, se pueden generar *segmentation faults* con demasiada facilidad.

La estructura básica para esta estrategia es la siguiente:

Listing 4: Estructura del algoritmo dinámico

DINAMICA (SolOptima):

```
SI paso recursivo == ULTIMO THEN:
    Calcular datos hasta el campamento

SI NO:
    Actualizar informacion
    Llamada recursiva a DINAMICA (con SolParcial)

COMPARAR SolParcial con SolMejor
    Actualizar SolMejor

DESHACER CAMBIOS

Actualizar SolOptima
```

Donde SolParcial y SolMejor son soluciones intermedias y locales a cada paso recursivo. Finalmente se actualiza SolOptima con la mejor de las opciones barajadas.

2.5. Estrategia Backtracking

Una estrategia *backtracking* otorga muchas ventajas:

- Es versátil
- Sigue un esquema que requiere de pocas variaciones
- Explora fielmente el espacio de estados en busca de la mejor solución
- Es completa

Y esas son las virtudes en este caso. Tan sólo hubo de hacerse distinción entre la variante 3 y las demás, al cambiar sensiblemente el contenido.

Dentro de la condición de control del algoritmo de *backtracking* sólo nos hace falta distinguir en qué variante estamos para aplicar un criterio u otro:

Listing 5: Control mediante backtracking

```

if ( variante==1){
    if (( Sol.time < SolOpt -> time) && Sol.balance >0){
        copySol(&Sol , SolOpt );
    }
else if ( variante==2){
    if (( Sol.time < SolOpt -> time) && ( Sol.balance > SolOpt->balance )){
        copySol(&Sol , SolOpt );
    }
}

```

En este caso, el problema se define de la siguiente manera:

1. **Test de solución:** El nodo en el que estamos no reduce nuestra economía por debajo de cero. La solución vendrá cuando todos los nodos estén visitados y el último sea el regreso al campamento.
2. **Test de fracaso:** El nodo reduce nuestra economía por debajo de cero.
3. **Generación de descendientes:** Todo nodo adyacente no visitado.

Cambia sensiblemente para la variante 3.

1. **Test de solución:** El nodo en el que estamos no reduce nuestra economía por debajo de cero. La solución vendrá cuando demos con un nodo (un estado en el recorrido) cuyo valor de oro sea el máximo.
2. **Test de fracaso:** El nodo reduce nuestra economía por debajo de cero.
3. **Generación de descendientes:** Todo nodo adyacente no visitado.

En este caso no hace falta regresar al inicio (y sería una locura volver durante el recorrido como mero tránsito, al recorrer una distancia extra para perder dinero por las tropas).

2.6. Conclusiones

Dado un problema, podemos contemplar que hay distintas formas de resolverlo en la mayoría de los casos. Unas más eficientes y otras más complejas, pero que aseguran completitud.

```
einath@Debbie: ~/Dropbox/Virtual Machine/AProg/
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
=====
1. Algoritmo voraz
2. Backtracking
3. Prog. Dinamica
=>2
Salida backtraking:
Tiempo: 38 - Balance: 7040
=====
Day      Place  Units  Gold  Spent  Balance
=====
0        CAMP   0      0     0     10000
12       ASTAPOR 350    10000 0     20000
20       BHORASH 850    6300  2800  23500
26       YUNKAI  1450   2900  5100  21300
29       MEEREEN 1990   8000  4350  24950
38       CAMP   1990   0     17910 7040
=====
```

Figura 6: Salida del programa

Las tres estrategias se adaptan bien a esquemas de optimización y en ocasiones puede ser interesante sacrificar la completitud por la eficiencia, como en el caso de los voraces.

Sobre la complejidad de implementación, el orden de dificultad para esta práctica diría que se establece en primer lugar (y más sencillo) el *backtracking*. Su esquema bien conocido y su potencia hacen que únicamente haya que preocuparse de no dejar ningún dato suelto. Seguidamente pondría al *voraz* pues no es inmediato hallar las heurísticas que definan el comportamiento del programa. Finalmente el más tedioso resulta el algoritmo *dinámico*. Quizá por falta de costumbre o porque realmente lo es, pero ha sido el que más vueltas ha requerido para este caso.

De cualquier forma, son tres estrategias con gran versatilidad y potencia, y jugar con sus características es generalmente lo que nos hace decantarnos por una u otra.

3. Tercera Práctica: Juegos

3.1. Enunciado

Los Lannister siempre pagan sus deudas y Tyrion Lannister ha prometido a su hermano Jaime repartir el tesoro encontrado formado por \mathbf{N} objetos de determinado valor V_i y peso P_i . Sobre una mesa coloca la colección de \mathbf{N} objetos. Deberán cogen alternativamente un objeto de la colección. El peso total de los objetos que coge un jugador no puede sobrepasar una cantidad dada \mathbf{K} (no pueden transportar más). El objetivo para ambos es lograr reunir la colección de objetos de mayor valor. Construya un algoritmo que dada una configuración del juego la evalúe, y devuelva la decisión que produce el resultado más favorable. Opcionalmente, implemente el juego completo en el que el usuario puede seleccionar el objeto elegido y la máquina juegue utilizando el algoritmo anteriormente implementado.

3.2. Planteamiento

La función de evaluación se fundamenta en obtener siempre la mejor jugada, dejando la peor en la medida de lo posible al oponente.

3.2.1. Nota sobre la implementación

Se ha utilizado como base un ejercicio resuelto de años anteriores. El ejercicio propone de base una serie de funciones que alivian la tarea de programar más allá de lo necesario.

Del mismo modo proponía un algoritmo similar que ha sido modificado para el particular.

3.2.2. Codificación

Listing 6: Pseudocódigo de la funcion

```
void Evaluar (JugadorA , JugadorB , K, Objeto [] , N , *Pos , *NumObjeto):
```

```
    SI JugadorA no puede jugar THEN:
```

```
        SI Jugador B no puede jugar THEN:
```

```
            SI JugadorA mejor puntuacion JugadorB THEN:
```

```
                Pos = GANADORA
```

```

    SI NO, SI JugadorA igual puntuacion JugadorB THEN:
        Pos = TABLAS
    SI NO:
        Pos = PERDEDORA

    SI NO:
        Evaluar(JugadorB, JugadorA, ... )
        SI PosicionOponente = GANADORA THEN:
            Pos = PERDEDORA
        SI NO, SI PosicionOponente = PERDEDORA THEN:
            Pos = GANADORA
        SI NO:
            Pos = TABLAS
    SI NO :
        Pos = PERDEDORA
    MIENTRAS Hay Objetos YY Pos != GANADORA LOOP:
        SI Objeto[i] No cogido YY JugadorA pueda coger Objeto[i] THEN:
            Objeto[i] COGIDO
            Actualizar datos JugadorA
            Evaluar (JugadorB, JugadorA, ...)
            SI PosicionOponente != GANADORA THEN:
                NumObjeto = i
                SI PosicionOponente = PERDEDORA THEN:
                    Pos = GANADORA
                SI NO:
                    Pos = Tablas

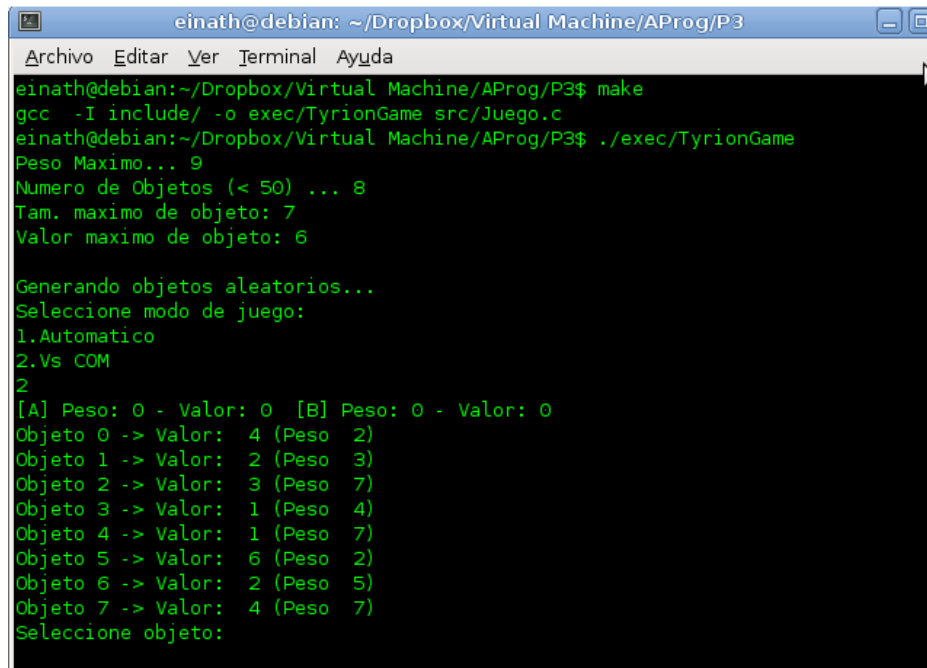
        DESHACER coger objeto
        i++

```

3.3. Adaptación para juego vs COM

La parte optativa tan sólo requería crear una función de control que midiese el flujo de la partida. La entrada incorrecta de datos se ha controlado mediante los oportunos bucles for, que impiden que el jugador humano pueda seleccionar un objeto que no esté disponible o que exceda su peso.

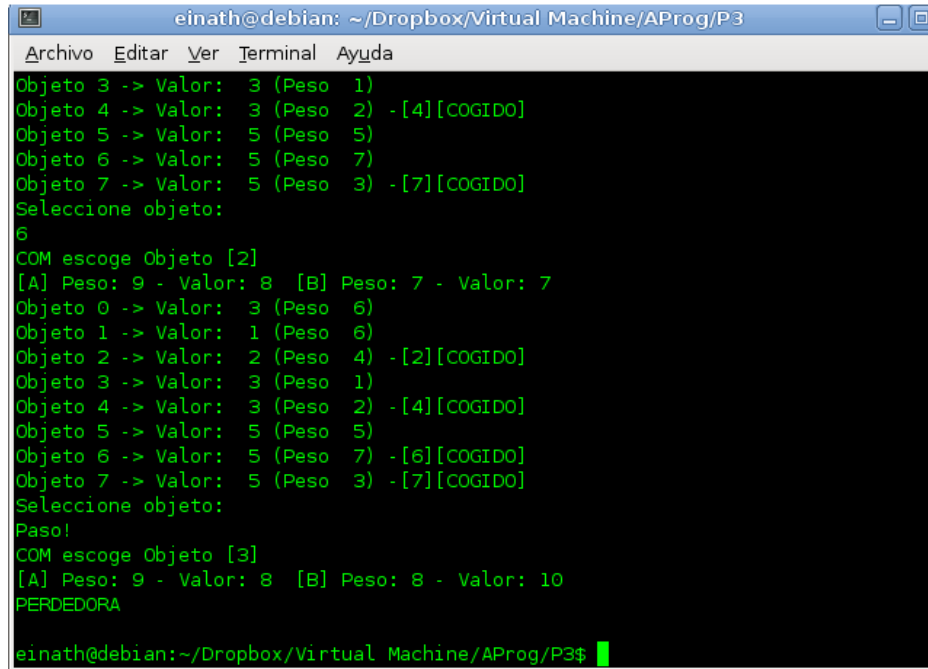
Cuando toca jugar a COM, sencillamente se realiza una llamada al método descrito pasándose él como `JugadorA` con el fin de optimizar sus jugadas.



```
einath@debian: ~/Dropbox/Virtual Machine/AProg/P3
Archivo Editar Ver Terminal Ayuda
einath@debian:~/Dropbox/Virtual Machine/AProg/P3$ make
gcc -I include/ -o exec/TyrionGame src/Juego.c
einath@debian:~/Dropbox/Virtual Machine/AProg/P3$ ./exec/TyrionGame
Peso Maximo... 9
Numero de Objetos (< 50) ... 8
Tam. maximo de objeto: 7
Valor maximo de objeto: 6

Generando objetos aleatorios...
Seleccione modo de juego:
1. Automatico
2. Vs COM
2
[A] Peso: 0 - Valor: 0 [B] Peso: 0 - Valor: 0
Objeto 0 -> Valor: 4 (Peso 2)
Objeto 1 -> Valor: 2 (Peso 3)
Objeto 2 -> Valor: 3 (Peso 7)
Objeto 3 -> Valor: 1 (Peso 4)
Objeto 4 -> Valor: 1 (Peso 7)
Objeto 5 -> Valor: 6 (Peso 2)
Objeto 6 -> Valor: 2 (Peso 5)
Objeto 7 -> Valor: 4 (Peso 7)
Seleccione objeto:
```

Figura 7: Muestra del programa



```
einath@debian: ~/Dropbox/Virtual Machine/AProg/P3
Archivo  Editar  Ver  Terminal  Ayuda
Objeto 3 -> Valor: 3 (Peso 1)
Objeto 4 -> Valor: 3 (Peso 2) -[4][COGIDO]
Objeto 5 -> Valor: 5 (Peso 5)
Objeto 6 -> Valor: 5 (Peso 7)
Objeto 7 -> Valor: 5 (Peso 3) -[7][COGIDO]
Seleccione objeto:
6
COM escoge Objeto [2]
[A] Peso: 9 - Valor: 8 [B] Peso: 7 - Valor: 7
Objeto 0 -> Valor: 3 (Peso 6)
Objeto 1 -> Valor: 1 (Peso 6)
Objeto 2 -> Valor: 2 (Peso 4) -[2][COGIDO]
Objeto 3 -> Valor: 3 (Peso 1)
Objeto 4 -> Valor: 3 (Peso 2) -[4][COGIDO]
Objeto 5 -> Valor: 5 (Peso 5)
Objeto 6 -> Valor: 5 (Peso 7) -[6][COGIDO]
Objeto 7 -> Valor: 5 (Peso 3) -[7][COGIDO]
Seleccione objeto:
Paso!
COM escoge Objeto [3]
[A] Peso: 9 - Valor: 8 [B] Peso: 8 - Valor: 10
PERDEDORA
einath@debian:~/Dropbox/Virtual Machine/AProg/P3$
```

Figura 8: Muestra del programa (II)

La generación de los objetos puede hacerse de manera aleatoria o bien de manera manual. Por defecto está en aleatorio y no se han dado opciones a nivel de tiempo de ejecución para esta parte. Sin embargo, para cambiarlo, basta con ir al código y cambiar un comentario:

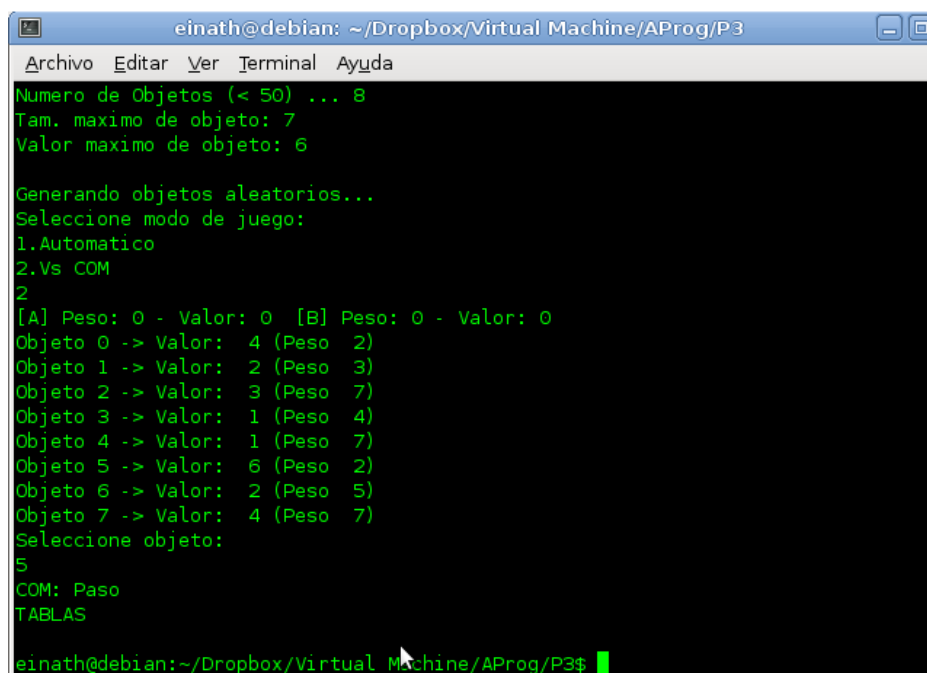
Listing 7: Generación de objetos

```
//Para generar los objetos de manera manual
// Entrada (&K,&N, Objeto);
//Para generar los objetos de manera automatica
Genera (&K,&N, Objeto);
```

3.4. Algunas consideraciones

Pueden generarse algunos fallos de segmento en el momento de escoger en el primer paso una jugada claramente beneficiosa para el jugador humano. Esa posición generalmente

debe devolver TABLAS, pero ocurre en algunos sistemas operativos (Ubuntu entre ellos, no así en Debian), que genera un fallo de segmento en lugar de mostrar una posición de TABLAS.



```
einath@debian: ~/Dropbox/Virtual Machine/AProg/P3
Archivo Editar Ver Terminal Ayuda
Numero de Objetos (< 50) ... 8
Tam. maximo de objeto: 7
Valor maximo de objeto: 6

Generando objetos aleatorios...
Seleccione modo de juego:
1. Automatico
2. Vs COM
2
[A] Peso: 0 - Valor: 0 [B] Peso: 0 - Valor: 0
Objeto 0 -> Valor: 4 (Peso 2)
Objeto 1 -> Valor: 2 (Peso 3)
Objeto 2 -> Valor: 3 (Peso 7)
Objeto 3 -> Valor: 1 (Peso 4)
Objeto 4 -> Valor: 1 (Peso 7)
Objeto 5 -> Valor: 6 (Peso 2)
Objeto 6 -> Valor: 2 (Peso 5)
Objeto 7 -> Valor: 4 (Peso 7)
Seleccione objeto:
5
COM: Paso
TABLAS
einath@debian:~/Dropbox/Virtual Machine/AProg/P3$
```

Figura 9: Estado de conflicto

3.5. Conclusiones

La función sobre la cual se ha trabajado refleja perfectamente el modo en que debe operarse para obtener un algoritmo eficiente. Contempla los casos posibles (poder o no poder escoger un objeto y seguidamente explorar en busca de la mejor solución).

La idea de recorrer el espacio de estados en busca de la mejor solución, es decir, la más beneficiosa para nosotros y perjudicial para el oponente, otorga al algoritmo de un alto grado de *inteligencia*. Se trata pues de un primitivo *MiniMax* sin ningún tipo de poda $\alpha\beta$ que aún así resulta suficientemente competitivo para una búsqueda simple.

4. Consideraciones

4.1. Ejecución de los programas

Todas las prácticas se encuentran programadas y probadas bajo GNU/Linux. Cada práctica se encuentra en su carpeta y cada una de éstas contiene lo siguiente:

- Archivo **makefile**: archivo de configuración de la utilidad **make**.
- Carpeta **src**: carpeta donde se albergan los archivos fuente.
- Carpeta **include**: en esta carpeta se guardan los archivos de cabeceras necesarios para la ejecución de las aplicaciones.
- Carpeta **exec**: los ejecutables tras utilizar **make** se guardan en esta carpeta.

Para compilar los programas, basta con colocarse con un terminal en la carpeta de práctica correspondiente y ejecutar **make**.

Listing 8: Compilación y ejecución

```
P1\$ make
gcc -I include/ -o exec/JonSnow src/JS.c
P1\$ ./exec/JonSnow
```

```
P2\$ make
gcc -I include/ -o exec/DaenerysTryp src/Daenerys.c
P2\$ ./exec/DaenerysTryp
```

```
P3\$ make
gcc -I include/ -o exec/TyrionGame src/Juego.c
P3\$ ./exec/TyrionGame
```

Los nombres de los fuentes y ejecutables son:

- Práctica 1: Archivo fuente: **JS.c**, ejecutable: **JonSnow**
- Práctica 2: Archivo fuente: **Daenerys.c**, ejecutable: **DaenerysTryp**
- Práctica 3: Archivo fuente: **Juego.c**, ejecutable: **TyrionGame**