

CatTheHobbie – The Sequel

S1-G1-G25

Marc González Carro Carlos Romero Rodríguez
(marc.gcarro) (c.romero)

Índice de la memoria

1) Lenguaje de programación elegido	3
1.1) Justificación	3
1.2) IntelliJIdea vs Terminal putty	3
1.2.1) Ventajas de IntelliJ IDEA	4
1.2.2) Ventajas de trabajar en la terminal Putty	4
2) Discusión de los algoritmos implementados	5
2.1) Navegación de alta velocidad:	5
2.1.1) Algoritmo Greedy	7
2.1.2) Algoritmo Backtracking	8
2.2) Flota al completo:	10
2.2.1) Algoritmo Greedy	11
2.2.2) Algoritmo BackTracking	13
2.2.3) Algoritmo Branch and Bound	13
3) Análisis de resultados	15
3.1) Navegación de alta velocidad:	15
3.1.1) Tiempo Greedy vs BackTracking	15
3.1.2) Tiempo BackTracking con y sin marcaje	18
3.1.3) Fuerza bruta sin PBMSC vs backtracking con PBMSC	19
3.1.4) Listas de navegantes y barcos ordenadas vs sin ordenar	22
3.2) Flota al completo:	23
3.2.1) Backtracking con PBMSC vs Branch & Bound	23
3.2.2) Backtracking con marcaje vs sin marcaje	25
4) Problemas Observados	26
5) Dedicación	27
6) Conclusiones	28
7) Bibliografía	29

1) Lenguaje de programación elegido

1.1) Justificación

Teniendo en cuenta, la limitación de conocimiento respecto a los lenguajes de programación existentes, ya que por ahora solo hemos dado en la universidad lenguaje C en terminal Putty (primer año de carrera) y Java (en IDE de programación IntelliJ Idea), nos hemos decantado por utilizar IntelliJ Idea y programarlo en Java por los siguientes motivos:

- Java es una excelente elección de lenguaje de programación debido a sus muchas ventajas en términos de **portabilidad, bibliotecas integradas y seguridad**. La capacidad de ejecutar código en múltiples plataformas sin realizar cambios adicionales es una ventaja significativa para los desarrolladores que desean llegar a un público más amplio. Además, el gran conjunto de bibliotecas integradas permite a los desarrolladores ahorrar tiempo y esfuerzo al no tener que escribir todo el código desde cero.
- La **seguridad integrada en Java** es otra ventaja importante, ya que los mecanismos de seguridad integrados, como la seguridad de la máquina virtual de Java, ayudan a proteger las aplicaciones contra ataques externos. El sistema de gestión de memoria automático también es una ventaja para los desarrolladores, ya que no tienen que preocuparse por administrar manualmente la memoria de la aplicación.
- Sin embargo, es importante tener en cuenta que Java también tiene algunas desventajas en comparación con otros lenguajes. La **sintaxis del lenguaje puede ser más complicada que otros lenguajes**, lo que puede dificultar la curva de aprendizaje para algunos desarrolladores. Además, la necesidad de tener la máquina virtual de Java instalada puede ser un problema para algunos usuarios en cuanto a espacio interno de la computadora, o falta de memoria RAM.
- También es cierto que **Java puede ser menos eficiente** en términos de rendimiento que otros lenguajes como C o C++. Sin embargo, en la mayoría de los casos, la diferencia de rendimiento es insignificante, y la portabilidad y seguridad que ofrece Java pueden superar estas desventajas.

1.2) IntelliJIdea vs Terminal putty

IntelliJ IDEA y la terminal Putty son herramientas muy diferentes que se utilizan para fines distintos. IntelliJ IDEA es un IDE (entorno integrado de desarrollo) diseñado específicamente para desarrollar aplicaciones en lenguaje Java, mientras que Putty es

una terminal que permite conectarse a servidores remotos y ejecutar comandos en ellos.

Dicho esto, si comparamos el uso de IntelliJ IDEA para desarrollar en Java y el uso de la terminal Putty para programar en lenguaje C, podemos hacer las siguientes observaciones:

1.2.1) Ventajas de IntelliJ IDEA

- **Interfaz de usuario intuitiva:** IntelliJ IDEA proporciona una interfaz de usuario gráfica que es fácil de entender y navegar, lo que lo hace mucho más fácil de usar que una terminal de línea de comandos como Putty.
- **Autocompletado y corrección de errores:** IntelliJ IDEA tiene una función de autocompletado que ayuda a los desarrolladores a escribir código más rápido y con menos errores. Además, también ofrece corrección automática de errores, lo que ahorra tiempo al solucionar problemas de código.
- **Integración con herramientas de construcción:** IntelliJ IDEA se integra perfectamente con herramientas de construcción como Maven y Gradle, lo que facilita la creación de proyectos y la administración de dependencias.
- **Depuración y pruebas integradas:** IntelliJ IDEA tiene características integradas de depuración y pruebas que hacen que el proceso de desarrollo sea más fácil y eficiente.

1.2.2) Ventajas de trabajar en la terminal Putty

- **Flexibilidad:** La terminal Putty es una herramienta muy flexible que se puede utilizar para conectarse a una amplia variedad de sistemas remotos, lo que permite trabajar en proyectos de lenguaje C que se ejecutan en servidores remotos.
- **Sin necesidad de una herramienta adicional:** A diferencia de IntelliJ IDEA, que requiere una instalación adicional para programar en lenguaje C, la terminal Putty se puede utilizar directamente desde cualquier sistema operativo que la admita.
- En la terminal Putty, los desarrolladores pueden **trabajar directamente con el compilador y los archivos fuente**, lo que les permite personalizar el proceso de compilación según sus necesidades. Además, algunos desarrolladores prefieren trabajar en la terminal porque les permite tener un control más preciso sobre los comandos que ejecutan y los resultados que obtienen.

2) Discusión de los algoritmos implementados

2.1) Navegación de alta velocidad:

- **Algoritmos implementados:**

- BackTracking
- Greedy

Para resolver este problema de maximización de velocidades, nos hubiese gustado poder implementar los 3 algoritmos obligatorios: BackTracking, Branch & Bound y Greedy.

Pese a ello, al tratarse de un problema de **maximización de velocidades**, esto supone dificultades para poder aplicar el algoritmo que queramos. Es por ello que pese a que la idea principal era resolver el problema con los 3 algoritmos, finalmente se ha tenido que optar de manera obligatoria por usar backtracking y, ya que se quiere encontrar la mejor posible solución y no es suficientemente complejo, creemos que un enfoque greedy también la puede llegar a garantizar, siempre y cuando se implemente de la manera correcta. Por otro lado, Branch and bound se ha tenido que descartar al no ser un problema de minimización, por lo que no puede aplicarse la PBMSC (Poda basada en la mejor solución en curso).

También, hemos tenido que pensar una heurística excelente para llevar a cabo el mejor tiempo de ejecución, y esto es lo que nos ha llevado algo más de tiempo para diseñar. El hecho es que al ser un problema de maximización, en un principio, puede ser difícil descartar opciones a medio recorrido, ya que no se sabe hasta el final si esa opción es válida o no, en otras palabras, supera la mejor velocidad guardada hasta el momento. Por ese motivo, se nos ocurrió una posible alternativa, que es pasarle como parámetros de entrada a la función, los array de barcos y navegantes ordenados de mejor a peor, en función de las especificaciones del enunciado.

Para ordenar los barcos, utilizamos la misma fórmula que en la primera práctica, y pese a ser una estimación propia, ya que esta fórmula ha sido diseñada por nosotros, realmente es totalmente válida, ya que se aplica para todos los barcos, con lo cual, parten de la misma ventaja, y por tanto, la ordenación final será con los barcos correctamente ordenados. Evidentemente, la fórmula está basada en la lógica, y es por ello, que cogemos el peso del barco y lo dividimos entre la eslora, la capacidad y la velocidad. Con lo cual, un resultado lo más cercano a cero significa que el divisor es mucho más grande que el dividendo, en otras

palabras, que el barco es mejor. Sabiendo esto, aprovecharemos esta fórmula para ordenar los barcos con el valor de la división de más pequeño a más grande utilizando el mismo quicksort que la primera práctica.

Aquí adjuntamos una imagen para que se entienda mejor:

```
private static void ordenarXPrestaciones (ArrayList<Boat> barcos, ArrayList<Boat> boatSorted) {
    float[] prestaciones = new float[barcos.size()];

    for (int i = 0; i < barcos.size(); i++) { // llenar el array de strings "prestaciones"
        Boat aux1 = barcos.get(i); // con todos los valores de la division
        prestaciones[i] = (float) (aux1.getPeso()/(aux1.getStore()+aux1.getCapacity()+aux1.getVelocity()));
    }
    long start = System.nanoTime();
    quickSort(prestaciones, izq: 0, der: barcos.size() - 1); //función recursiva
    long end = System.nanoTime();
}
```

Por otro lado, para la ordenación de los navegantes de mejor a peor, nos hemos basado en hacer otra fórmula, esta vez, dividiendo el peso del propio navegante entre el sumatorio de su suma de habilidades y su ratio de victorias. De igual manera que la fórmula de los barcos, si analizamos la fórmula veremos que un valor más próximo a cero significa que el dividendo es mucho mayor al divisor, y esto indica que el navegante tiene una gran suma de habilidades y un gran número de victorias. De esta manera, utilizando el mismo quicksort que en la primera práctica y que en el ordenamiento de barcos, los ordenamos de mejor a peor. Adjuntamos imagen ilustrativa para que se entienda mejor:

```
private static void ordenarNavegantes(@NotNull ArrayList<Sailor> sailors, ArrayList<Sailor> sailorsSorted){
    float[] prestaciones = new float[sailors.size()];
    float sumHab;

    for (int i = 0; i < sailors.size(); i++) { // llenar el array de strings "prestaciones"
        Sailor aux1 = sailors.get(i);
        sumHab = 0.0f;
        for(int z = 0; z < aux1.getHabilities().length; z++){
            sumHab += aux1.getHabilities()[z];
        }
        sumHab = sumHab / 6.0f; // con todos los valores de la division
        prestaciones[i] = aux1.getPeso()/(sumHab + aux1.getRatio_victories());
    }
    long start = System.nanoTime();
    quickSort(prestaciones, izq: 0, der: sailors.size() - 1); //función recursiva
    long end = System.nanoTime();
}
```

Finalmente, una vez tenemos ambos objetos ordenados en un array de mejor a peor, utilizamos el algoritmo de backtracking o el greedy, pero esta previa ordenación nos asegura una significativa reducción de tiempo del algoritmo backtracking (que es el que suele tardar más al probar todas las posibles combinaciones, aunque sea con poda o no) de varios segundos sin ordenamiento previo, o incluso en uno de nuestros portátiles con poca potencia, ni siquiera vimos que finalizara, a tardar este tiempo (0,2s) con el fichero L (con 50 barcos y 200 navegantes) y 0,02 con el XS (6 barcos y 14 navegantes):

```
143 switch (showMenuProblema_3_1()) {
144     case 1 -> {
145         long start1 = System.nanoTime();
146         backTracking(config, 0, sailorsSorted, boatsSorted, best, 0);
147         long end1 = System.nanoTime();
148         System.out.println("\nTiempo backtracking ordenando previamente: " + (end1 - start1) + "ns -> " + ((end1 - start1) / Math.pow(10, 9)) + "s.");
149     }
}

Run

-----
Tiempo backtracking ordenando previamente: 204627000ns -> 0.204627s.

Elija una de las 4 opciones:
1- Problema 3_1
2- Problema 3_2
3- Salir
```

Analizando el tiempo, también podemos apreciar que es bastante proporcional al número de barcos (lo calculamos con los barcos, ya que són los que limitan la capacidad máxima, es decir, el número de navegantes). Esto se puede demostrar dividiendo 50 barcos del fichero L, entre 6 barcos del fichero XS (lo cual da como resultado 8,333333...) y multiplicando por el tiempo del fichero XS o dividiendo entre el tiempo del fichero L, viendo que el resultado en ambos casos se asimila mucho al tiempo del fichero contrario.

En resumen, consideramos que pese a gastar algo más de tiempo implementando una buena heurística de ordenamiento, ha valido la pena en cuanto a tiempo de ejecución y en otras palabras, ha ahorrado subprocesos para la RAM en portátiles menos potentes, que al fin y al cabo es de lo más importante de un algoritmo. De todas formas, en cuanto a costes y otras posibles mejoras, estas serán analizadas en el siguiente punto, “Análisis de resultados”.

2.1.1) Algoritmo Greedy

El algoritmo greedy, en este problema en particular, trata de buscar la solución más óptima en base a tomar decisiones óptimas en cada paso del código, pero sin probar todas las posibles combinaciones a diferencia del backtracking o branch and bound (aunque en este ejercicio no se usa este último).

En este ejercicio, el algoritmo greedy funciona de la siguiente manera:

- Previamente al algoritmo, como ya hemos mencionado, se ordenan los barcos por velocidad máxima de mayor a menor y los navegantes por prestaciones, de mejor a peor también.
- Para cada barco, se eligen los navegantes disponibles con mayor factor individual hasta llenar el barco con la capacidad máxima que este presenta.
- Finalmente se repite el proceso para todos los barcos hasta llenarlos, pudiendo sobrar navegantes (pero con el previo ordenamiento de

navegantes, nos aseguramos que en el supuesto caso de que sobren navegantes, estos serán los peores, con lo cual no afecta demasiado al resultado final).

```
private static void greedy(@NotNull ArrayList<Sailor> sailors, ArrayList<Boat> boats) {
    double best = 0.0;
    for (Sailor sailor : sailors) {
        Boat bestBoat = null;
        double bestSpeed = 0.0;
        for (Boat boat : boats) {
            if (boat.getCapacity() > 0 && sailor.getPeso() <= boat.getPeso()) {
                double impNav = calculateImpNav(sailor, boat);
                double velocidad = boat.getVelocity() * impNav;
                if (velocidad > bestSpeed) {
                    bestBoat = boat;
                    bestSpeed = velocidad;
                }
            }
        }
        if (bestBoat != null) {
            sailor.setId(bestBoat.getId());
            best += bestSpeed;
            bestBoat.setCapacity(bestBoat.getCapacity() - 1);
        }
    }
}
```

(La función subrayada, básicamente es una función para calcular en cada paso el navegante con mejor relación prestación-velocidad, ya que aunque previamente estén ordenados, en algunos casos un navegante puede ser mejor para un tipo de barco específico que otro. Básicamente, es como una doble “checkeada” para asegurar totalmente el correcto funcionamiento del algoritmo).

En este caso, el algoritmo greedy es muy eficiente, ya que utiliza decisiones locales óptimas para llenar los barcos con los navegantes disponibles. Además, al tener los navegantes y barcos ordenados, se evita la exploración de combinaciones inválidas, lo que reduce el tiempo de ejecución.

2.1.2) Algoritmo Backtracking

El algoritmo de backtracking explora exhaustivamente todas las posibles soluciones hasta encontrar la óptima. En este caso, el algoritmo de backtracking puede funcionar de la siguiente manera:

- Genera todas las posibles combinaciones de navegantes y barcos.
- Calcula la velocidad real de cada combinación.

- Elige la combinación que maximiza la velocidad real y agrega su velocidad, al resultado final.
- Finalmente, una vez se lleva al caso final en el que todos los barcos están llenos, solo en el caso de que su velocidad máxima supere la velocidad máxima actual, se printa la solución y se actualiza dicha velocidad máxima y se guarda dicha configuración, para que al final se mantenga la velocidad máxima con la mejor combinación de navegantes-barcos.

```

203 private static void backTracking(int @NotNull [] config, int k, ArrayList<Sailor> sailors, ArrayList<Boat> boats, double best){
204     double aux1;
205     config[k] = 0;
206
207     while (config[k] <= 1) {
208         if (k < config.length - 1) {
209             for(int h = 0; h < boats.size(); h++){
210                 sailors.get(k).setId(boats.get(h).getId());
211                 if(checkSolution(sailors,boats)){
212                     backTracking(config, k+1, sailors, boats, best);
213                 }
214             }
215         } else {
216             // Solution case
217             if(checkFinalSolution(sailors,boats)){
218                 aux1 = printSolution(sailors, boats);
219                 if(aux1 > best){...}
220             }
221             // Next successor
222             config[k]++;
223         }
224     }
225 }

```

Función para “checkear” la solución actual/parcial, comprobando que los navegantes del barco, no superen su capacidad:

```

private static boolean checkSolution(ArrayList<Sailor> sailors, @NotNull ArrayList<Boat> boats) {
    for (Boat boat : boats) {
        int quantity = 0;
        for (Sailor sailor : sailors) {
            if (sailor.getId() == boat.getId()) {
                quantity++;
            }
        }
        if (quantity > boat.getCapacity()) {
            return false;
        }
    }
    return true;
}

```

Función para “checkear” la solución final de la misma forma que la función anterior, pero esta vez una vez ha terminado de generar una combinación de barcos-navegantes:

```

private static boolean checkFinalSolution(ArrayList<Sailor> sailors, @NotNull ArrayList<Boat> boats) {
    for (Boat boat : boats) {
        int quantity = 0;
        for (Sailor sailor : sailors) {
            if (sailor.getId() == boat.getId()) {
                quantity++;
            }
        }
        if (quantity > boat.getCapacity() || quantity == 0) {
            return false;
        }
    }
    return true;
}

```

El algoritmo de backtracking garantiza la solución óptima, pero puede ser muy lento en casos de gran tamaño si no se aplica un previo proceso de ordenación, ya que exploran todas las posibles soluciones. Además, en este caso, el algoritmo de backtracking puede generar muchas combinaciones no válidas, y pese a comprobarlas, esto aumenta el tiempo de ejecución, y pese a ya ser bastante rápido.

2.2) Flota al completo:

- **Algoritmos implementados:**
 - BackTracking
 - Branch and Bound
 - Greedy

Para resolver este segundo problema, este caso de minimización de centros de barcos, sí que hemos podido implementar los 3 algoritmos obligatorios: BackTracking, Branch & Bound y Greedy.

Al tratarse de un problema de **minimización de centros**, esto no supone dificultad alguna para poder aplicar cualquiera de los 3 algoritmos. Ya que a diferencia del primer problema, Branch and bound se puede implementar perfectamente, pudiendo aplicar la PBMSC (Poda basada en la mejor solución en curso). Y también cabe destacar que a diferenciar que los problemas de maximización, en este tipo de problemas si que se puede podar cómodamente, ya que en todo momento se puede controlar si la configuración que se está generando, supera el mejor resultado guardado (en este caso, el más bajo), en cuyo caso anularemos dicha poda y pasaremos a explorar otra configuración comprobando si pasa lo mismo o no, y así sucesivamente hasta comprobar todas las posibles combinaciones.

En este problema, no hemos ordenado con previa heurística los centros en las funciones BackTracking y Branch and Bound, pero en cambio en el Greedy si que lo hemos aplicado, buscando en cada paso, el centro que pueda aportar más cantidad de tipos de barco, para completar el conjunto de centros, y llegar a un candidato de solución. Bien es cierto, que lo podríamos haber aplicado al BackTracking y al Branch and Bound, pero hemos querido probar sin ello, para comprobar si con ficheros muy grandes tardan relativamente mucho dando errores, o no tienen problema alguno (mera curiosidad).

Por otro lado, como el enunciado no garantiza una posible solución:

Així doncs, voldrem trobar el conjunt més petit de centres que, entre tots, ens permeti cobrir aquesta necessitat, tenint en total com a mínim un vaixell de cada tipus. Tingueu present que, segons les dades d'entrada, pot ser que no existeixi una solució.

Previamente a llamar a los algoritmos del problema 2, tenemos una función que se encarga de comprobar que en el fichero de barcos, al menos haya un barco de cada tipo con un bucle básicamente, y un TreeSet en el que se van añadiendo los tipos encontrados y un TreeSet que se compone de los tipos de barco, hardcodedos como entrada. Para que así queden ordenados los tipos de manera alfabética, y a la hora de comparar, se compare exactamente los tipos encontrados con los guardados. Esto lo hemos hecho para asegurar, pero como el enunciado dice que los tipos solo pueden ser de uno de los 6 tipos, se podría haber hecho con un “Set<String>” y sabríamos que hay uno de cada tipo si “Set<String>.size == 6”. Adjuntamos imagen de la función con un TreeSet y un bucle:

```
private static boolean checkCenters(@NotNull List<Center> centers) {  
    TreeSet<String> dictionary = new TreeSet<>(Arrays.asList("Windsurf", "Optimist", "Laser", "Pati Català", "HobieDragoon", "HobieCat"));  
    TreeSet<String> aux = new TreeSet<>();  
    for (Center c : centers) { for (Boat boat : c.getBoats()) { aux.add(boat.getType()); } }  
    return aux.equals(dictionary);  
}
```

2.2.1) Algoritmo Greedy

Tal y como hemos mencionado anteriormente, para este algoritmo sí que hemos usado cierta heurística, ya que considerando que el algoritmo Greedy se caracteriza por no probar todas las combinaciones para ahorrar tiempo y en algunos casos, pudiendo aportar soluciones incorrectas, hemos implementado un sistema que nos permita asegurarnos que en cada paso que da el algoritmo, al menos elija la mejor opción, es decir, el centro que aporta más tipo de barco restantes para completar el subconjunto. Para ello hemos usado la función “.sort” de java:

```
// Ordena los centros en función de la cantidad de tipos de barcos que ofrecen para los tipos de barcos restantes
Global.centers.sort((c1, c2) -> centerTypeCounts.get(c2).size() - centerTypeCounts.get(c1).size());
```

Por otro lado, para hacer más eficiente el algoritmo, hemos implementado un sistema de Clave valor, que nos permita mediante el nombre del centro que se está evaluando en el nivel del algoritmo, saber el conjunto de tipos de barco que tiene, para así poder ordenarlos en la función anterior:

```
public void greedy() {
    List<Center> selectedCenters = new ArrayList<>();
    Set<String> remainingBoatTypes = new HashSet<>(Global.boatTypes);

    while (!remainingBoatTypes.isEmpty()) {
        Map<Center, Set<String>> centerTypeCounts = new HashMap<>();
```

Es por ello que aparte de utilizar un Set de String, para garantizar que los tipos de barco no se repitan, también hemos usado un HashMap, el cual realiza la finalidad previamente mencionada, a través del nombre del centro, aporta los tipos de barco que tiene, teniendo un coste $O(1)$.

Finalmente con un bucle anidado, conseguimos asegurarnos que el centro leído, puede aportar un tipo de barco que no esté ya añadido en la configuración parcial, ya que no sirve de nada añadir centros con barcos repetidos, ya que va en contra de la finalidad del ejercicio. Para ello hacemos “marcaje”/ comprobación, para saber si dicho centro se ha añadido o no a la configuración, tal y como se puede apreciar en la imagen inferior:

```
boolean found = false;
for (Center center : Global.centers) {
    if (selectedCenters.contains(center)) { continue; }
    for (Boat boat : center.getBoats()) {
        String type = boat.getType();
        if (remainingBoatTypes.contains(type)) {
            selectedCenters.add(center);
            remainingBoatTypes.remove(type);
            found = true;
            break;
        }
    }
    if (found) { break; }
}
if (!found) { break; }
```

2.2.2) Algoritmo BackTracking

Para este algoritmo, como ya hemos mencionado anteriormente, no hemos usado una heurística como tal, aunque podríamos haberla aplicado ordenando previamente los centros como en el greedy, por cantidad de barcos de tipos distintos de mayor a menor.

Por otro lado sí que se usa una poda, y se aplican mejoras de rendimiento para hacer más eficiente y rápido el algoritmo. En este código se realiza una poda para evitar explorar ramas que no conducirán a una solución óptima. Si la cantidad de centros seleccionados es mayor que el tamaño de la mejor solución encontrada hasta el momento, entonces la rama actual se poda.

```
// PBMSC
if (centerContainsBoatType && newSelectedCenters.size() < bestSize) {
    newSelectedCenters.add(center);
    backTracking(centers, newBoatTypes, newSelectedCenters, bestSolutions, minBoats);
    newSelectedCenters.remove(center);
}
```

Y en cuanto a mejoras de rendimiento, la utilización de la estructura de datos HashSet es una mejora importante de rendimiento en este algoritmo. En concreto, la operación de eliminación de tipos de barcos es una operación clave para la eficiencia del algoritmo, ya que se realiza de manera repetitiva en cada iteración del bucle. Utilizar un HashSet para almacenar los tipos de barcos permite una búsqueda más eficiente de los tipos de barcos y elimina la necesidad de iterar sobre toda la lista de tipos de barcos:

```
2 usages
public void backTracking(List<Center> centers, @NotNull Set<String> boatTypes,
```

2.2.3) Algoritmo Branch and Bound

Para esta función también se utiliza una poda, comprobando si el coste de la configuración parcial (el número de centros) supera a la mejor solución, en cuyo caso, se parará la poda y se explorará una nueva rama. También, usamos, aunque debería ser básico y obligatorio, una optimización una vez la configuración está completa de centros, para ver si puede ser una futura solución, comparando de igual manera su coste (número de centros) con el resultado de la mejor configuración encontrada hasta el momento.

```

public void branchAndBound() {
    priorityQueue.add(new Config());

    while (!priorityQueue.isEmpty()) {
        for (Config successor : priorityQueue.poll().expand()) {
            if (successor.isFull()) {
                //Optimización
                if (successor.getCost() < minNumberOfCenters) {
                    minNumberOfCenters = successor.getCost();
                }
            } else {
                //PBMSC
                if (successor.getCost() < minNumberOfCenters) {
                    priorityQueue.add(successor);
                }
            }
        }
    }
}
}

```

Por otro lado, basándonos en la heurística de la función expand, únicamente en ella llenamos al futuro successor con los tipos de barco del centro que se está explorando en ese nivel, y una vez llenado, se comprueba si el número de centros de este successor (una vez recorridos todos los niveles/ probadas todas las posibles combinaciones) es menor a la mejor solución, esta se actualiza con el nuevo valor y si almacena dicha configuración también, ambas en una variable global:

```

public List<Config> expand() {
    List<Config> successors = new ArrayList<>();
    for (int i = 0; i < Global.centers.size(); i++) {
        if (!visitedCenters[i]) {
            Config successor = new Config( that: this);

            //Añadir los tipos de barco que tiene ese centro
            for (int j = 0; j < Global.centers.get(i).getBoats().size(); j++) {
                successor.availableBoatTypes.add(Global.centers.get(i).getBoats().get(j).getType());
            }

            if (successor.availableBoatTypes.size() == 6) {
                successor.minNumCenters++;
                if (successor.getCost() < best) {
                    best = successor.getCost();
                }
            }

            successor.config[successor.level] = i;
            successor.level++;
            successor.visitedCenters[i] = true;

            successors.add(successor);
        }
    }
    return successors;
}

```


También, otro punto a destacar, es que en la clase Globals (únicamente se utiliza para hacer el Branch and Bound), como en el enunciado nos aseguran que solo existen 6 tipos de barco, nos hemos creado un Set (aunque con una List<String>) también serviría, pero el Set nos permitía hacer pruebas por nuestra cuenta más cómodamente al no aceptar elementos repetidos, y en este hemos puesto los 6 tipos de barco que especifica el enunciado:

```
public class Global {
    10 usages
    public static List<Center> centers;
    3 usages
    public static Set<String> boatTypes = new HashSet<>(Arrays.asList("Windsurf", "Optimist", "Laser", "Pati Català", "HobieDragoon", "HobieCat"));

    no usages
    public Global() {
        Global.centers = new ArrayList<>();
    }

    1 usage
    public static void setCenters(List<Center> centers) { Global.centers = centers; }
}
```

Además hemos añadido en esta misma clase, otra variable global para almacenar la lista de centros en la configuración actual.

3) Análisis de resultados

3.1) Navegación de alta velocidad:

Sin tampoco pensar mucho, podemos asegurar que un algoritmo Greedy, **siempre** será más rápido que un algoritmo de BackTracking o un Branch and Bound, ya que no explora todo el espacio de soluciones posibles, y por tanto, no las puede contemplar en la solución final. Es por ello, que podemos medir sus tiempos y corroborar esta afirmación, pero por otro lado, creemos que no tiene sentido compararlos, ya que al no usar el Greedy fuerza bruta, no se pueden considerar algoritmos de la misma categoría. Por eso, en vez de compararlos detalladamente entre ellos, profundizaremos más en la comparación del BackTracking con y sin marcaje para ver su efecto en cuanto al tiempo y coste del algoritmo.

3.1.1) Tiempo Greedy vs BackTracking

Como vemos en la siguiente imagen, el tiempo del greedy es incomparable con el del BackTracking (0,01s del greedy vs 0,2s del backtracking ambos con el fichero L).

```
151         long start2 = System.nanoTime();
152         greedy(sailorsSorted, boatsSorted);
153         long end2 = System.nanoTime();
154         System.out.println("\nTiempo greedy: "+(end2-start2)+"ns -> "+((end2-start2)/Math.pow(10, 9))+".s.");
155     }
}

Run
navegante: Angel Himalayan,
Barco ID: 70616
Navegante: Wilma Leggrowbach
Navegante: Chris Ko
Barco ID: 60
Navegante: Pepe Roni
Navegante: Eli Ondefloor

Velocidad Total: 0,330492

-----

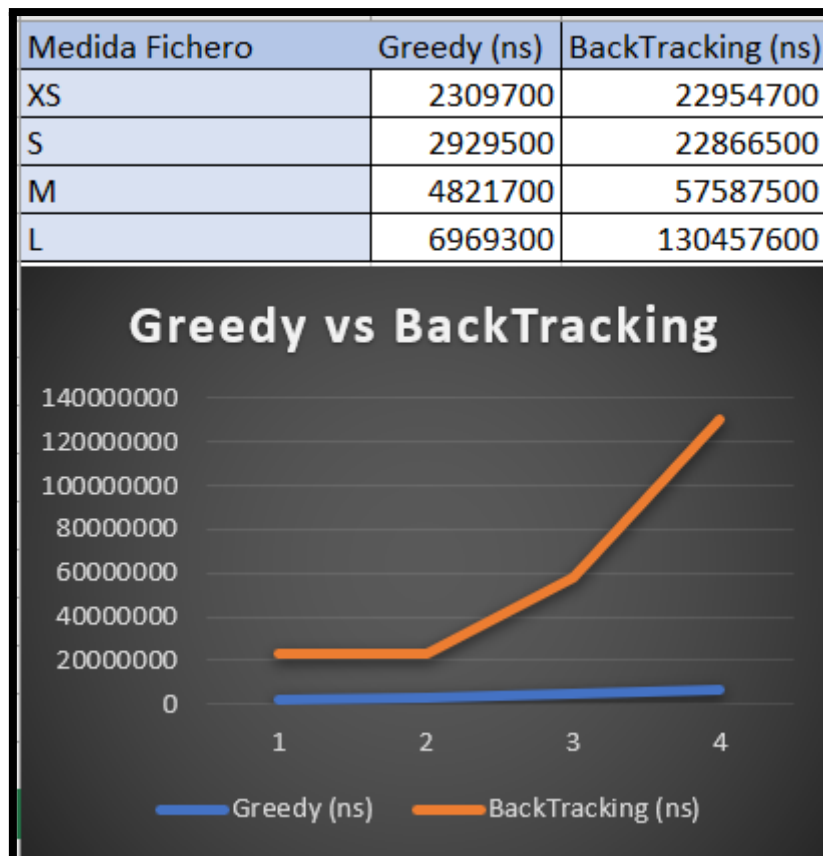
Tiempo greedy: 10979200ns -> 0.0109792s.

Elija una de las 4 opciones:
1- Problema 3_1
2- Problema 3_2
3- Salir
```

Por otro lado también podemos darnos cuenta que no tardan lo mismo porque los costes entre el Greedy y el BackTracking no son los mismos. Por un lado el algoritmo de **Greedy** (al depender principalmente del número de navegantes y barcos que se tengan; el bucle externo recorre todos los navegantes, y para cada uno de ellos se recorren todos los barcos, por lo tanto, si se tienen “n” navegantes y “m” barcos, **el coste es del orden de $O(n*m)$**). Además, dentro del bucle interno se realizan una serie de operaciones de comparación y cálculo de velocidad, pero estas operaciones son constantes y no dependen del tamaño de los datos de entrada, por lo que no influyen en el coste asintótico del algoritmo).

Mientras que por otro lado el coste del algoritmo de **BackTracking** es significativamente mayor (se trata de un **coste exponencial $O(n^m)$** , donde “n” es el número de navegantes y “m” el número de barcos, en el caso de no tener ordenados ni los barcos ni los navegantes, pero al aplicar una heurística en la que le pasamos las dos listas ordenadas, la complejidad teórica del algoritmo de backtracking sigue siendo exponencial en el peor de los casos, ya que explora todas las posibles soluciones. Sin embargo, si los navegantes y los barcos están ordenados como es el caso, entonces el número de soluciones que se exploran podría ser menor en la práctica, ya que el algoritmo no tendría que explorar todas las combinaciones posibles de navegantes y barcos. Por lo tanto, aunque la complejidad teórica del algoritmo siga siendo exponencial en el peor de los casos, su tiempo de ejecución es más rápido en la práctica).

A continuación se puede observar de manera gráfica el coste de un algoritmo Greedy ($n*m$) y el de un BackTracking (n^m):



Esta gráfica sirve para corroborar el coste teórico calculado anteriormente para el algoritmo Greedy y el algoritmo BackTracking, y en efecto tal y como hemos mencionado anteriormente, y tal y como se puede observar en la gráfica, el algoritmo Greedy a mayor tamaño de fichero, el tiempo se mantiene lineal (en una línea recta), la cual es un vector multiplicada por el factor de la $m \cdot n$, mientras que el algoritmo de BackTracking se puede observar cómo a medida que incrementa el fichero, el tiempo crece exponencialmente, demostrando así el coste " n^m ". [Los números 1-4, hacen referencia a los ficheros del XS-L respectivamente].

También podemos observar que el resultado final/ velocidad total, varía significativamente entre el Greedy y el BackTracking.

Velocidad total greedy (0,33 km/h):

```

Navegante: Eli OnDefloor

Velocidad Total: 0,330492

-----

Tiempo greedy: 10979200ns -> 0.0109792s.

```

Velocidad BackTracking (0,2 km/h):

```
Barco ID: 60
  Navegante: Jona Talaxian
  Navegante: Pepe Roni

Velocidad Total: 0,205839

-----

Tiempo backtracking ordenando previamente: 140255800ns -> 0.1402558s.

Elige uno de los 4 cruceros:
```

Esto se debe a que tal y como hemos mencionado anteriormente, un greedy reduce en tiempo y costes, pero no garantiza la mejor solución final, tal y como se puede apreciar. Eso no descarta, que en algunos casos, sí que la encuentre y más rápido que un BackTracking. Pero como un algoritmo Greedy siempre tiene contraejemplos, los cuales apreciamos en el AC5 de ordenación de tesoros, creemos que sí se tiene tiempo y el rendimiento no es un gran problema, es infinitamente mejor probar un BackTracking ya que garantizará una solución correcta en el 100% de los casos siempre y cuando esté bien implementado.

3.1.2) Tiempo BackTracking con y sin marcaje

En esta comparativa se analizarán los efectos de aplicar marcaje durante el proceso de fuerza bruta de BackTracking. La finalidad de aplicar un marcaje, es simple: evitar explorar soluciones parciales que ya han sido exploradas anteriormente y que no llevan a una solución válida. Esto se hace mediante la asignación de una marca o etiqueta (booleano) a cada solución parcial que se ha explorado, de manera que cuando se explora otra solución parcial, se comprueba si esta solución ya ha sido explorada (booleano = true) previamente mediante la verificación de su etiqueta. Si la solución ya ha sido explorada, se puede evitar su exploración y continuar con la búsqueda de otras soluciones no exploradas. De esta manera, el marcaje puede ayudar a reducir el tiempo de ejecución y mejorar la eficiencia del algoritmo de backtracking.

```
Velocidad Total: 0,205839
```

```
-----
```

```
Tiempo backtracking ordenando previamente sin marcaje: 133872500ns -> 0.1338725s.
```

```
Tiempo backtracking ordenando previamente con marcaje: 34600ns -> 3.46E-5s.
```

En efecto así es, podemos observar como el tiempo es infinitamente menor para el backtracking con marcaje versus sin marcaje, y ambos leen el fichero L. Esto se debe a tal y como previamente mencionamos, a la no consideración de opciones repetidas, que en un algoritmo con coste exponencial, pueden ser bastantes y estas perjudican bastante al tiempo de ejecución.

Aun así, creemos que ambos algoritmos son lo suficientemente rápidos como para apenas notar la diferencia, al menos para el ojo humano, ya que los ficheros no son muy grandes. Pero en ficheros, por ejemplo un fichero con todos los usuarios de Instagram (millones), la diferencia se puede reducir en segundos, minutos o horas de diferencia entre un algoritmo con y sin marcaje, ya que habrá cientos de miles o millones de combinaciones repetidas, cosa que se puede evitar. Con lo cual, creemos y recomendamos, siempre que sea posible (que suele ser en el 100% de los casos), aplicar marcaje para así hacer un algoritmo totalmente eficiente y escalable en el tiempo, en este caso, con datasets de mayor tamaño.

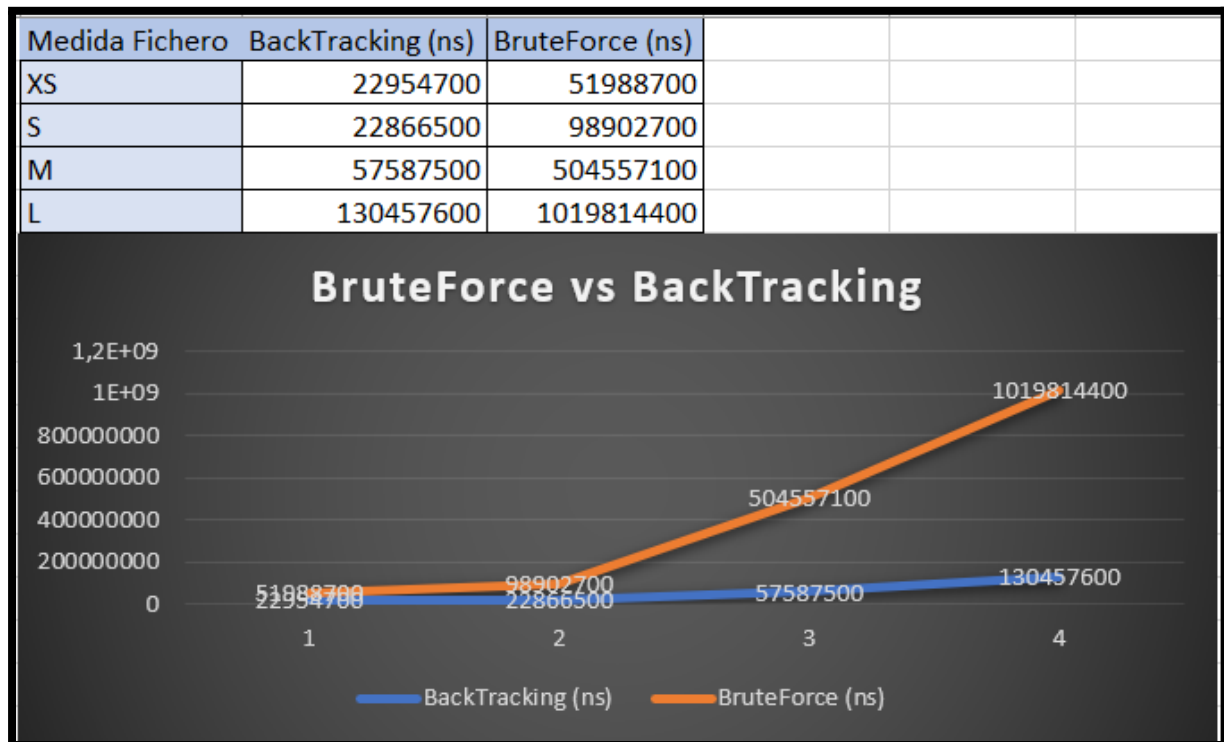
3.1.3) Fuerza bruta sin PBMSC vs backtracking con PBMSC

En primer lugar, podemos decir que el algoritmo de Fuerza Bruta es un enfoque exhaustivo y sistemático para encontrar todas las posibles soluciones a un problema (Lo implementamos, pero como tardaba tanto con los ficheros M y L, pese a esperar más de 7 minutos y escuchar la sobre ventilación del portátil, decidimos eliminarlo y en el informe, hacer la comparativa de manera teórica). Por otro lado, el Backtracking es un algoritmo de búsqueda que explora todas las posibles soluciones de manera incremental, deteniéndose en el momento en que detecta que una solución parcial no puede llevar a una solución completa.

La principal diferencia entre ambos algoritmos radica en el hecho de que el algoritmo de Fuerza Bruta probará todas las combinaciones posibles sin tener en cuenta si algunas de ellas son claramente subóptimas, mientras que el Backtracking es capaz de podar ramas del árbol de búsqueda que ya no pueden

conducir a soluciones óptimas mediante la PBMSC (poda basada en la mejor solución en curso).

El coste computacional del algoritmo de Fuerza Bruta puede ser muy elevado, ya que tiene que explorar todas las posibles soluciones. En cambio, el Backtracking es capaz de reducir significativamente el número de soluciones a explorar mediante la aplicación de podas y restricciones que ayudan a descartar soluciones subóptimas rápidamente.



Tal y como se puede apreciar en este gráfico, la principal razón por la que el tiempo de un algoritmo de BruteForce crece exponencialmente más rápidamente que el de un algoritmo BackTracking es que el primero explora todas las soluciones posibles, mientras que el segundo descarta ciertas soluciones de forma temprana, lo que reduce significativamente el espacio de búsqueda.

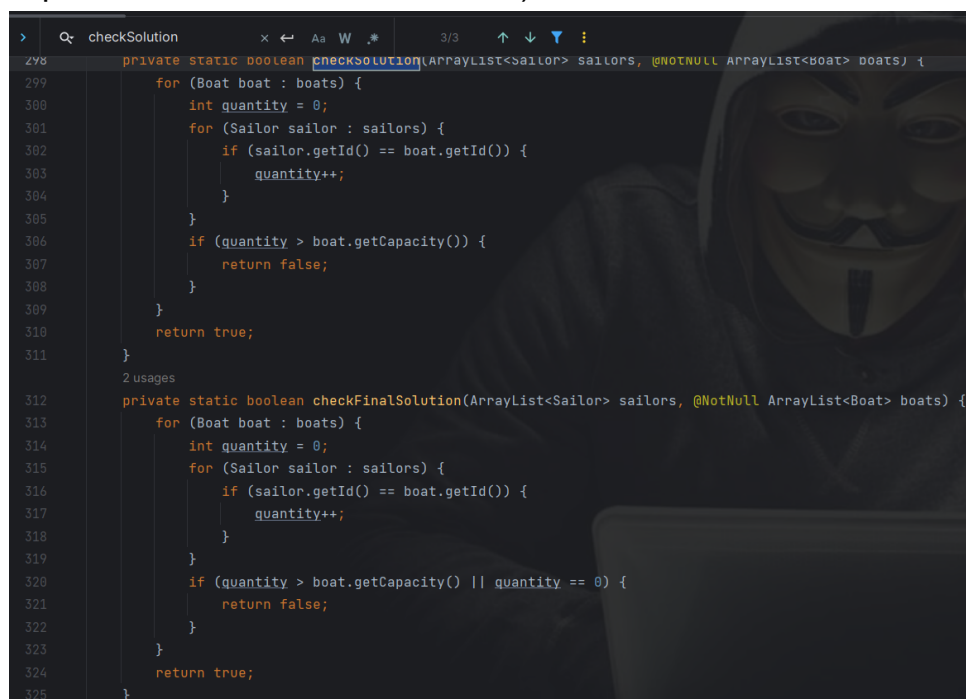
En el caso de un algoritmo de fuerza bruta, el tiempo de ejecución aumenta exponencialmente con el tamaño del problema. Esto se debe a que el número de posibles soluciones aumenta de forma exponencial con el tamaño del problema, lo que hace que el tiempo necesario para explorar todas las soluciones también aumente de forma exponencial.

Por otro lado, el algoritmo BackTracking, mediante la poda de ramas que no pueden contener la solución óptima, reduce significativamente el espacio de búsqueda. Esto hace que el tiempo necesario para explorar todas las soluciones sea mucho menor que en un algoritmo de fuerza bruta, especialmente en problemas grandes.

En cuanto al rendimiento, el algoritmo de Fuerza Bruta puede ser muy lento en la mayoría de los casos, mientras que el Backtracking puede ser mucho más rápido debido a las podas que realiza. Además, el Backtracking puede ser aún más rápido si se aplican mejoras como la poda basada en la mejor solución en curso, lo que permite descartar rápidamente soluciones que no pueden superar la mejor solución encontrada hasta ese momento. Y si a más a más, se le aplica un marcaje al BackTracking, la diferencia puede ser abismal, pese al tiempo extra de programar y de pensar que eso conlleve

En resumen, podemos afirmar que el Backtracking es un enfoque más sofisticado que el Fuerza Bruta para resolver problemas de búsqueda en los que nos queremos asegurar de la mejor solución, probando todas las posibles combinaciones, ya que este último es un Fuerza Bruta, solo que mejorado para agilizar el proceso. Aunque puede requerir más esfuerzo de programación, el Backtracking puede ser mucho más eficiente y rápido en la mayoría de los casos gracias a las podas y restricciones que aplica, y recomendamos usar esta segunda opción, siempre que la cantidad de ramas que se desea explorar sea un número considerable ya que su gasto es exponencial, y a partir de 7 o 8 barcos, ya empieza a ser muy costoso en tiempo.

Adjuntamos imagen de la PBMS del BackTracking, la cual se aplica tanto en la configuración parcial, como en la configuración final, para asegurarnos que el número de navegantes por barco de la configuración generada, no supera el máximo número de navegantes por barco (ambas funciones són idénticas, solo que como queríamos que tuviesen nombres más autodescriptivos, las hemos duplicado cambiándoles el nombre):



```
> Q checkSolution x ← Aa W .* 3/3 ↑ ↓ 🔍
298 private static boolean checkSolution(ArrayList<Sailor> sailors, @NotNull ArrayList<Boat> boats) {
299     for (Boat boat : boats) {
300         int quantity = 0;
301         for (Sailor sailor : sailors) {
302             if (sailor.getId() == boat.getId()) {
303                 quantity++;
304             }
305         }
306         if (quantity > boat.getCapacity()) {
307             return false;
308         }
309     }
310     return true;
311 }
2 usages
312 private static boolean checkFinalSolution(ArrayList<Sailor> sailors, @NotNull ArrayList<Boat> boats) {
313     for (Boat boat : boats) {
314         int quantity = 0;
315         for (Sailor sailor : sailors) {
316             if (sailor.getId() == boat.getId()) {
317                 quantity++;
318             }
319         }
320         if (quantity > boat.getCapacity() || quantity == 0) {
321             return false;
322         }
323     }
324     return true;
325 }
```

Estas dos funciones se llaman estratégicamente en el BackTracking, ya que no vale comprobar configuraciones generadas en cualquier sitio. Aquí se puede observar:

```
private static void backTracking(int @NotNull [] config, int k, ArrayList<Sailor> sailors, ArrayList<Boat> boats, double best){
    double aux1;
    config[k] = 0;

    while (config[k] <= 1) {
        if (k < config.length - 1) {
            for(int h = 0; h < boats.size(); h++){
                sailors.get(k).setId(boats.get(h).getId());
                if(checkSolution(sailors,boats)){
                    backTracking(config, k + 1, sailors, boats, best);
                }
            }
        } else {
            // Solution case
            if(checkFinalSolution(sailors,boats)){
                aux1 = printSolution(sailors, boats);
                if(aux1 > best){
                    System.out.print("\nMejor Solución Encontrada!\n");
                    best = aux1;
                    System.out.print("\n-----\n");
                    for(Boat b : boats){
                        System.out.printf("\nBarco ID: %d\n", b.getId());
                        for(Sailor s : sailors){
                            if(s.getId() == b.getId()){
                                System.out.printf("\tNavegante: %s\n", s.getName());
                            }
                        }
                    }
                }
            }
        }
    }
}
```

3.1.4) Listas de navegantes y barcos ordenadas vs sin ordenar

- **Costes:**

El algoritmo de backtracking con los barcos y navegantes previamente ordenados presenta una mejora significativa en cuanto a los costes se refiere. La razón es que, al tener los barcos y navegantes ordenados, se pueden explorar primero las combinaciones más prometedoras, lo que reduce el número de ramas que deben ser exploradas. De esta forma, se reducen los costes de exploración y se mejora el rendimiento del algoritmo, agilizando significativamente el tiempo.

- **Heurísticas:**

En términos de heurísticas, el algoritmo de backtracking con los barcos y navegantes previamente ordenados utiliza dos heurísticas: una basada en el orden de los barcos y otra basada en el orden de los navegantes. La finalidad de ambas heurísticas es permitir explorar primero las combinaciones más prometedoras. En el supuesto caso de que sobraran navegantes, estos serían los peores. Por su parte, el backtracking sin previa ordenación no utiliza ninguna heurística, por lo que explora todas las combinaciones posibles, lo que puede hacer que su rendimiento sea mucho peor que el del algoritmo de backtracking con los barcos y navegantes previamente ordenados.

- **Mejoras de rendimiento o podas:**

En cuanto a mejoras de rendimiento o podas, el algoritmo de backtracking con los barcos y navegantes previamente ordenados utiliza varias técnicas de poda para evitar explorar ciertas ramas del árbol de búsqueda que no conducen a una solución óptima. Estas técnicas de poda se basan en la capacidad de los barcos y en la exploración de las combinaciones más prometedoras primero. Por su parte, el backtracking sin esta previa ordenación, no utiliza ninguna técnica de poda, por lo que puede explorar todas las ramas del árbol de búsqueda, lo que puede hacer que su rendimiento sea mucho peor que el del algoritmo de backtracking con los barcos y navegantes previamente ordenados.

En base a estas tres comparaciones, podemos concluir lo siguiente: el algoritmo de backtracking con los barcos y navegantes previamente ordenados presenta una mejora significativa en términos de costes, heurísticas y mejoras de rendimiento o podas respecto al backtracking sin esta previa ordenación. Además, las heurísticas y técnicas de poda utilizadas en el algoritmo de backtracking con los barcos y navegantes previamente ordenados permiten evitar explorar ciertas ramas del árbol de búsqueda que no conducen a una solución óptima y, es por ello, que tarda bastante menos tiempo.

3.2) Flota al completo:

3.2.1) Backtracking con PBMS vs Branch & Bound

En esta comparativa, analizaremos el rendimiento y tiempo de ambos algoritmos de fuerza bruta, pero esta vez, ambos algoritmos, no tendrán listas ordenadas como parámetros de entrada, lo que hará que la comparativa sea totalmente igualada.

En primer lugar el BackTracking de este segundo problema de minimización, utiliza PBMS ya que se puede saber en todo momento si el coste de la configuración actual (parcial o final) supera a la mejor configuración, en cuyo caso se descarta ahorrando tiempo de exploración de una rama que no puede aportar la mejor configuración. De esta manera, el tiempo se reduce bastante. A continuación se puede ver el tiempo del algoritmo de BackTracking para el fichero L (50 barcos):


```
NUEVA SOLUCIÓN ÓPTIMA num.centers (2) num.Boats (8)
Optimal Centers-> Kaitain: Time Travel Mustard Seed (HobieDragon)
Optimal Centers-> Cronenberg World: Nova Force Corn Oil (Laser)
Mullet (Patí Català) / Soy Sauce Seven of Nine (HobieCat)

Tiempo BackTracking con Marcaje y PBMSC: 31399400 ns (31.0ms)
```

Por otro lado, tenemos el algoritmo de Branch & Bound, el cual también utiliza PBMSC de la misma manera que el BackTracking, comprobando si el coste de la configuración actual supera la mejor configuración para descartar dicha configuración y generar una nueva. Aquí se puede apreciar el tiempo con el fichero L también:

```
Con que método desea hacer 'Flota al complet'?

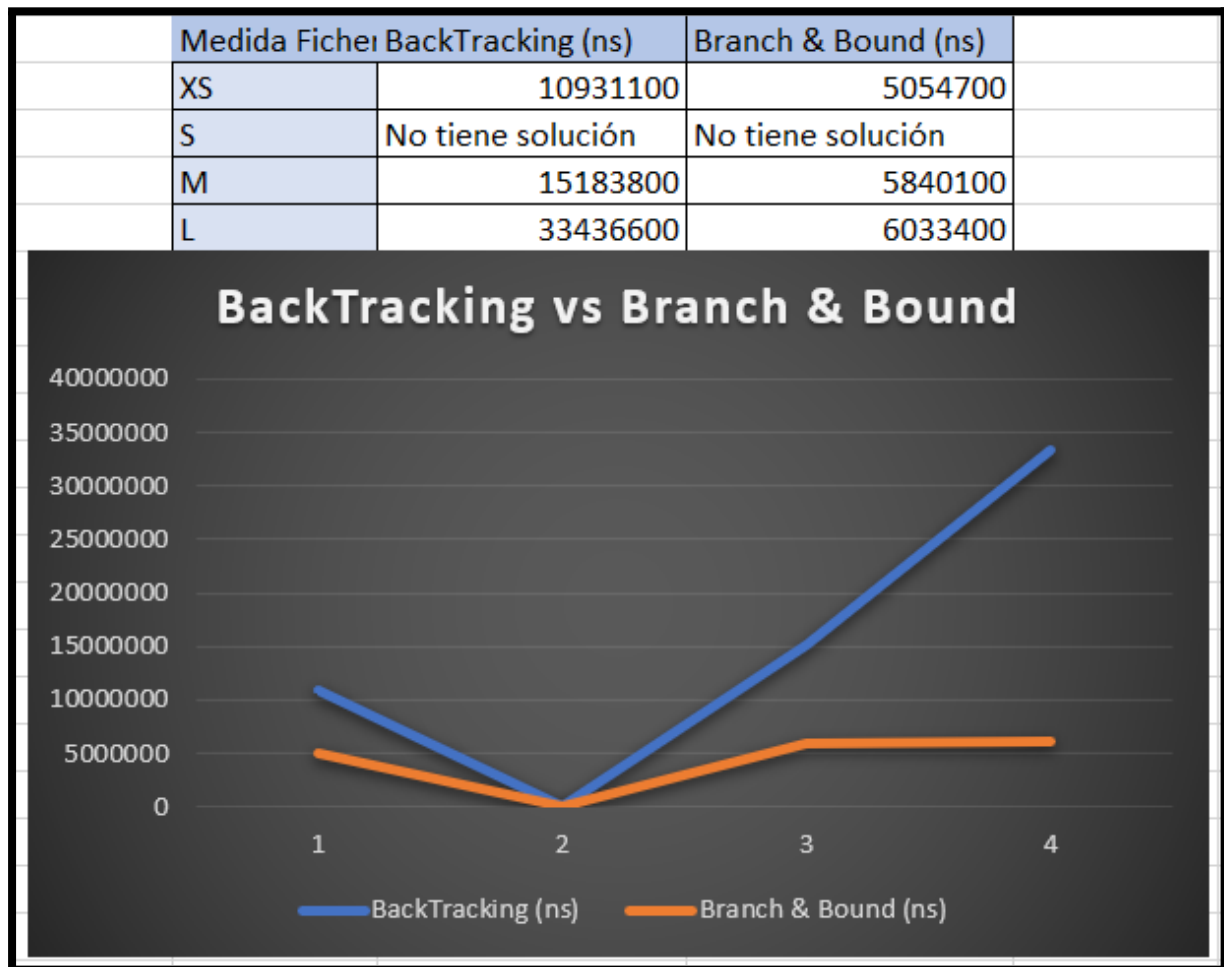
1- BackTracking
2- Branch & Bound
3- Greedy
4- Volver al menú principal

Opcion? 2
MinNumberCenters: 2

Tiempo Branch and Bound: 5026700 ns (5.0ms)
```

Esta diferencia de tiempo tiene explicación, y es que este margen de tiempo se debe a la forma en la que ambos algoritmos exploran el espacio de configuraciones. Mientras que el BackTracking es un algoritmo de “*Deep Searching First*”, cuya finalidad es ir explorando las ramas hasta el final, y si esta no tiene una solución menor a la mejor del momento, no se almacena y se continúa con la siguiente rama y así hasta el final. Por otro lado, el algoritmo Branch & Bound explora de manera horizontal “*Breadth Searching First*”, de manera paralela a los finales de rama, y es por ello que el tiempo es menor, ya que para cada nivel explora los costes de las ramas del mismo nivel y se queda con la mejor. Este proceso aunque parezca lo mismo que el BackTracking no lo

es, y eso se refleja en los tiempos entre ambos algoritmos, y cabe destacar que ambos utilizan PBMSC.



Es importante destacar que el algoritmo Branch & Bound es más complejo en su implementación y requiere una mayor cantidad de recursos, pero a cambio ofrece un mejor rendimiento en términos de tiempo. Por lo tanto, la elección de uno u otro algoritmo dependerá de las necesidades específicas del problema y de los recursos disponibles para su implementación.

3.2.2) *Backtracking con marcaje vs sin marcaje*

Esta comparativa viene a ser la misma que el punto 3.1.2. Hemos querido implementar ambos BackTracking para ambos problemas, para así ver si realmente los backtracking con marcaje son más eficientes en dos problemas totalmente distintos (el primero de maximización y el segundo de minimización), y después de analizarlos y exportar los resultados, podemos concluir que sí que son más eficientes y rápidos.

4) Problemas Observados

Para la realización de esta práctica no hemos tenido problemas que resaltemos como extremadamente complicados. Sí que hemos tenido algún error al implementar el Branch & Bound, pero cuando se ejecuta el código y se ve que la solución no es la correcta (comparándola con el BackTracking del mismo problema), IntelliJIdea facilita un debugger excel·lente y enseguida hemos podido encontrar el error.

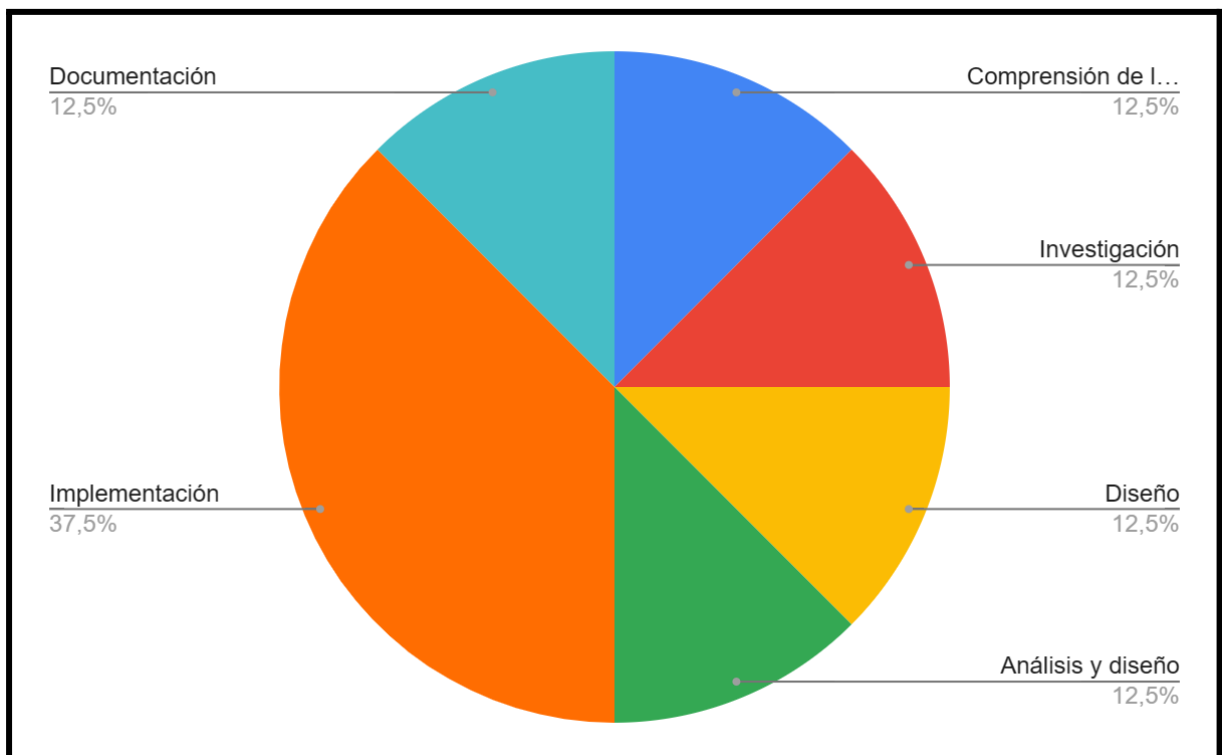
También otro problema que hemos tenido, pero no es que sea un problema como tal, si no que para alguna comparativa hemos querido hacer modificaciones en los algoritmos y a la hora de hacer un fuerza bruta sin PBMSC con el fichero L, nos hemos encontrado que no paraba de generar combinaciones de manera exponencial, y en unos de nuestros portátiles con poca RAM daba un error al cabo de varios minutos. No adjuntamos imagen del error, ya que el portátil hiperventilaba bastante y no queremos volverlo a forzar al máximo. El otro portátil, sí que no ha tenido ningún problema, y ha sido con este con el que se han realizado las pruebas más exigentes de algún algoritmo de fuerza bruta puro y sin PBMSC.

Finalmente, otra cosa que nos llevó algo más de tiempo y queremos mencionar, aunque no es un problema como tal, es el hecho de pensar cómo mejorar los algoritmos diseñando buenas heurísticas y luego implementarlas, para reducir significativamente los tiempos y procesos de carga de los algoritmos.

5) Dedicación

Para la realización de esta práctica estimamos una dedicación aproximada de 8h que podríamos desglosar en:

- Comprensión: 1h
- Investigación: 1h
- Diseño: 1h
- Implementación: 3h
- Anàlisis de resultados: 1h
- Documentación: 1h



6) Conclusiones

Gracias a la realización de esta práctica hemos podido mejorar nuestros conocimientos sobre la algoritmia. Hemos podido cumplir con el objetivo descrito en el enunciado, la realización de dos apartados a resolver usando un mínimo de dos algoritmos diferentes.

Ya que cada algoritmo de un mismo apartado cumplía la misma función hemos podido apreciar la diferencia tanto en el tiempo como en la logística de los algoritmos. Nos preguntamos cuál algoritmo sería más rápido para resolver un problema o cuál sería el más eficiente en relación a la memoria. Pero solo eran hipótesis que pudimos comprobar una vez teníamos la implementación hecha y operativa.

También hemos mejorado en el uso del lenguaje de programación, en concreto Java, explorando nuevas funciones proporcionadas o buscando librerías que nos pudiesen ayudar a entender según qué cosas.

Otro ámbito remarcable sería el uso del IDE para resolver el problema. Al utilizar IntelliJ para programar descubrimos el apartado de plugins y gracias a eso hemos mejorado cuantitativamente la calidad de nuestro entorno de programación. Por ejemplo, usamos el plugin Code with Me, para poder escribir código al mismo tiempo o algún plugin visual para poder diferenciar mejor las funciones entre otros muchos.

Como se ha demostrado en la parte de análisis de datos y comparaciones de algoritmos no es de extrañar que el más efectivo sea el Branch&Bounds. Con una muestra reducida puede que todos los algoritmos tarden aproximadamente lo mismo en ejecutarse plenamente, pero a medida que vamos ampliando la muestra el coste de cada uno sale a relucir.

Cabe recalcar también que un algoritmo no tiene solo una forma de ser programado, aplicando heurísticas y decisiones de diseño se pueden llegar a optimizar aún más

Más personalmente, esta práctica nos ha parecido entretenida ya que no tenía una complejidad demasiado elevada pero tampoco te daba el resultado en el enunciado. A causa de esto pudimos pasar una cantidad idónea de tiempo en la fase de pensar y diseñar los algoritmos.

7) Bibliografía

(Documentos informativos de la práctica)

Estudy 2021/22 - campus virtual La Salle BCN. eStudy 2021/22: Categorías. (n.d.). Retrieved March 17, 2023, from <https://estudy2122.salle.url.edu/course/index.php>