**Pruebas de Software**

PEC1: Prueba de algoritmos que resuelven el Problema de la Mochila

Alumno: Iván Fernández Rodríguez

DNI:34629848J

Correo electrónico: ifernande1631@alumno.uned.es

Centro Asociado: Ourense

Índice

[1. Solución tarea 1: Probar búsqueda\_exhaustiva 3](#_Toc164959818)

[a. Clases de equivalencia y valores límite. 3](#_Toc164959819)

[b. Especificación ACTS de los valores de prueba. 5](#_Toc164959820)

[c. Pruebas de unidades con pytest 5](#_Toc164959821)

[d. Corrección del programa 5](#_Toc164959822)

[2. Solución tarea 2: Prueba búsqueda\_con\_poda y algoritmo\_voraz 5](#_Toc164959823)

[3. Solución tarea 3: Implementación y prueba de una solución basada en programación dinámica 5](#_Toc164959824)

1. **Solución tarea 1: Probar búsqueda\_exhaustiva**

* 1. **Clases de equivalencia y valores límite.**

Para determinar las clases de equivalencia y los valores límite he seguido los siguientes pasos:

* **Identificar que variables interesa probar y su tipo**. Solamente tenemos una variable de entrada (mochila), que es una instancia de la clase Mochila. La descomponemos en las siguientes variables:
  + **Capacidad mochila (C)**
  + **Número de artículos (N)**
  + **Peso artículo (P)**
  + **Valor artículo (V)**
* **Identificar el tipo de cada variable**.
  + **C** : entero
  + **N** : entero
  + **P** : entero
  + **V** : entero
* **Determinar las dimensiones de cada variable**. Todas las variables son de tipo entero, por lo que, el dominio de valores es “infinito”.
  + **Capacidad**. Es una dimensión principal, ya que determina la restricción más fundamental del problema, pues los artículos deben caber dentro de la mochila sin exceder esta capacidad.
  + **Número de artículos**. El número de artículos es otra dimensión principal, ya que afecta directamente la complejidad del problema. cuantos más artículos haya para elegir, más opciones habrá y más compleja será la solución.
  + **Peso y Valor**. Son dimensiones secundarias. Aunque son importantes para determinar la solución óptima, no son tan fundamentales como la capacidad y el número de artículos. Estas dos dimensiones están intrínsecamente relacionadas entre sí, ya que buscamos maximizar el valor total llevando artículos con un peso total que no exceda la capacidad de la mochila.
* **Particionar la dimensión principal**. Consideraremos las siguientes particiones:
  + Capacidad > 0 (random(1, 20)
  + Capacidad = 0 (0)
  + Capacidad < 0 (-2)
  + Número de artículos > 0 (random(1 ,5)
  + Número de artículos = 0 (0)
  + Número de artículos < 0 (-1)
* **Determinar si existe alguna relación entre los valores primarios de las variables**. El número de artículos en el array, como el valor o el peso de cada uno de ellos no influye en los valores de la variable capacidad.
* **Diseñar la tabla de clases de equivalencia y valores límite para la dimensión principal**.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Variable** | **Clase válida** | **Clase inválida** | **Valor** | **Id. Del valor** |
| **capacidad mochila (C)** | 0 < C |  | random(1, 20) | C1 |
|  |  | C < 0 | -2 | C2 |
|  | C = 0 |  | 0 | C3 |
| **numero de artículos (N)** | 0 < N |  | random(1, 5) | N1 |
|  |  | N < 0 | -1 | N2 |
|  | N = 0 |  | 0 | N3 |
| **peso artículo (P)** | 0 < P |  | random(1, 10) | P1 |
|  |  | P = 0 | 0 | P2 |
|  |  |  | -2 | P3 |
| **valor artículo (V)** | 0 < V |  | random(1, 10) | V1 |
|  |  | V < 0 | -3 | V2 |
|  | V = 0 |  | 0 | V3 |

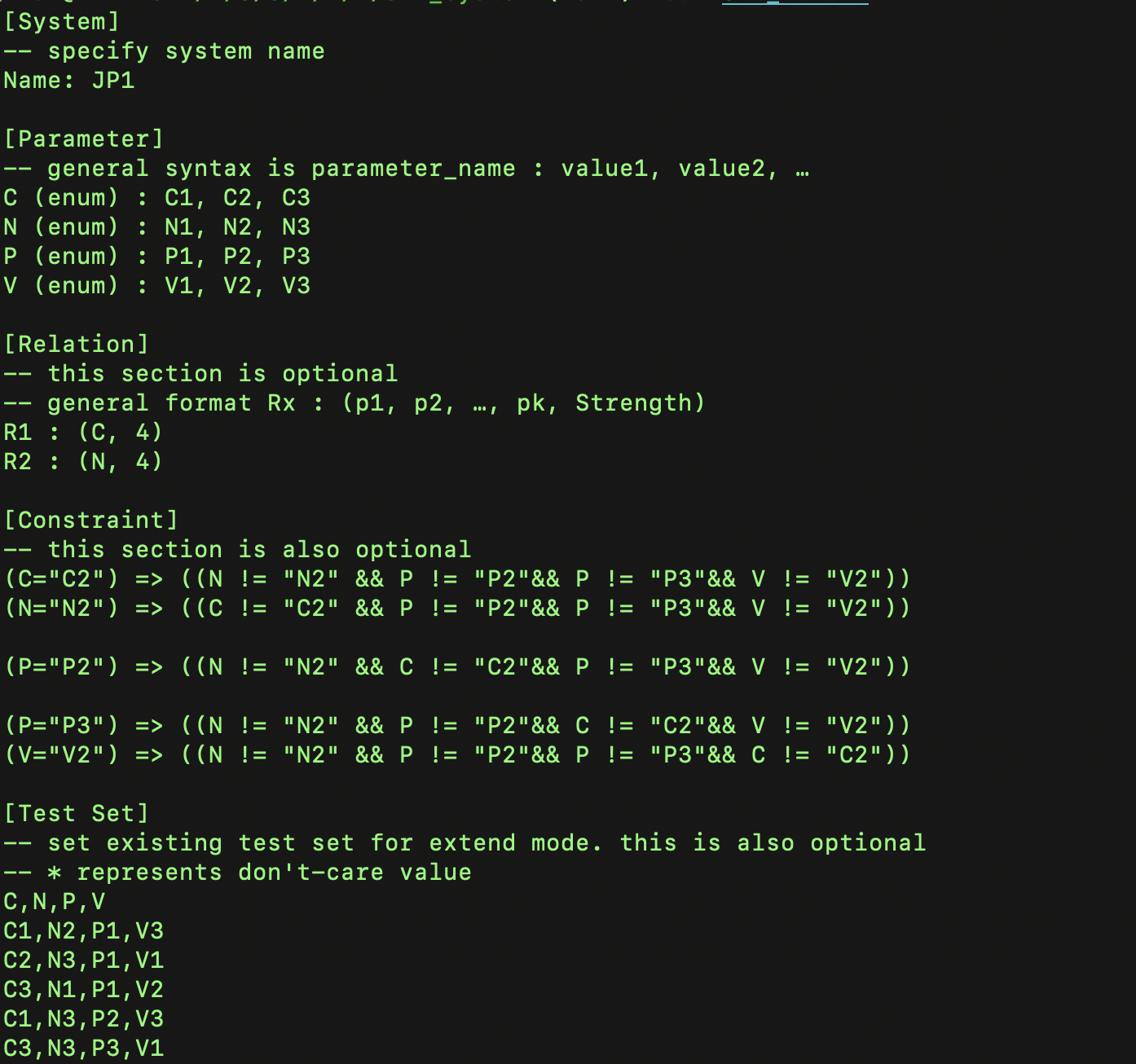
* **Identificar las restricciones.**

Las restricciones entre las dimensiones de interés podrían incluir:

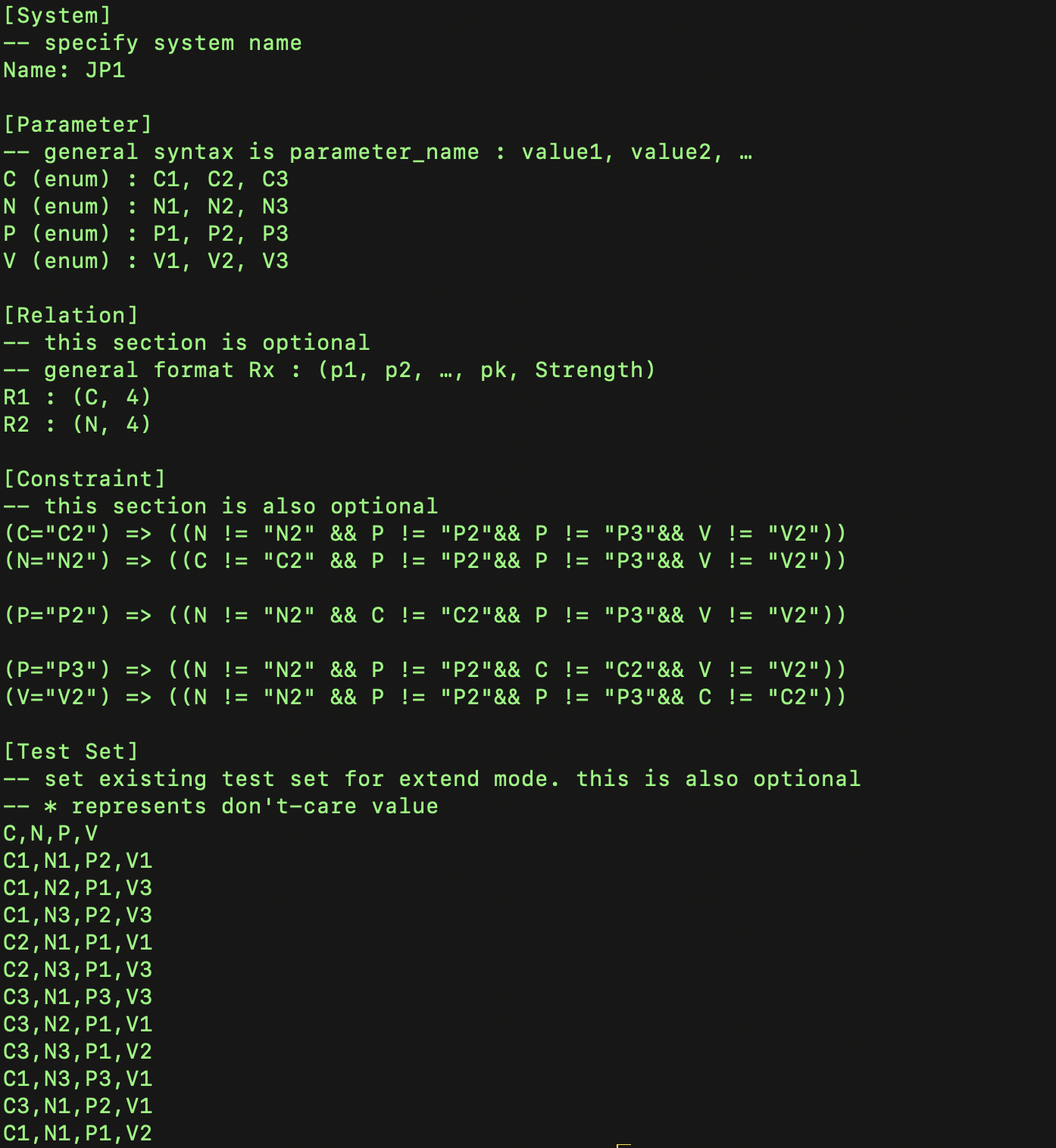
* + Capacidad de la mochila: debe ser un valor positivo. Debe ser lo suficientemente grande como para permitir la colocación de al menos un artículo, de lo contrario, el algoritmo de búsqueda exhaustiva no tendría sentido.
  + Valor de los artículos: los valores de los artículos deben ser valores no negativos, de lo contrario, invalidaría la necesidad de la búsqueda exhaustiva.
  + Peso de los artículos: deben ser valores positivos y menores o iguales a la capacidad de la mochila, de lo contrario será imposible seleccionarlos.
  + Número de artículos: debe ser un valor entero no negativo. Si es cero, entonces no hay necesidad de ejecutar el algoritmo.
* **Reescribir las restricciones**. El objetivo de las siguientes restricciones es que no se evalúe más de un valor no válido en cada test:
  1. **Especificación ACTS de los valores de prueba.**

He obtenido los dos juegos de pruebas alternativos como indica el enunciado, y los he exportado en formato csv (JP1-output.csv y JP2-output.csv) con el fin de usarlos en los test con la librería Pandas de Python.

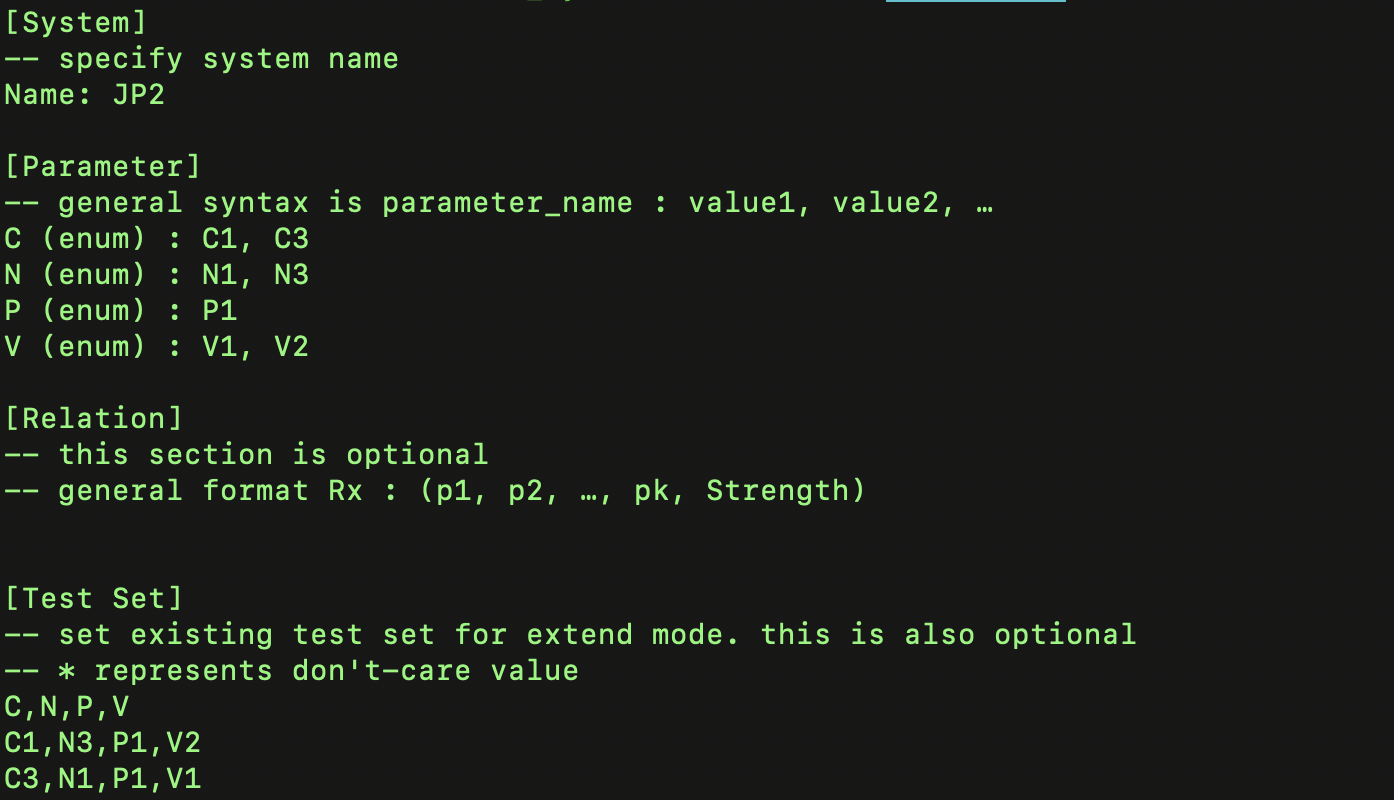
* **Para JP1 y t=1**: Utilizando ACTS con interfaz gráfica obtenemos el siguiente test set. Tenemos en cuenta las restricciones especificadas en el apartado anterior.



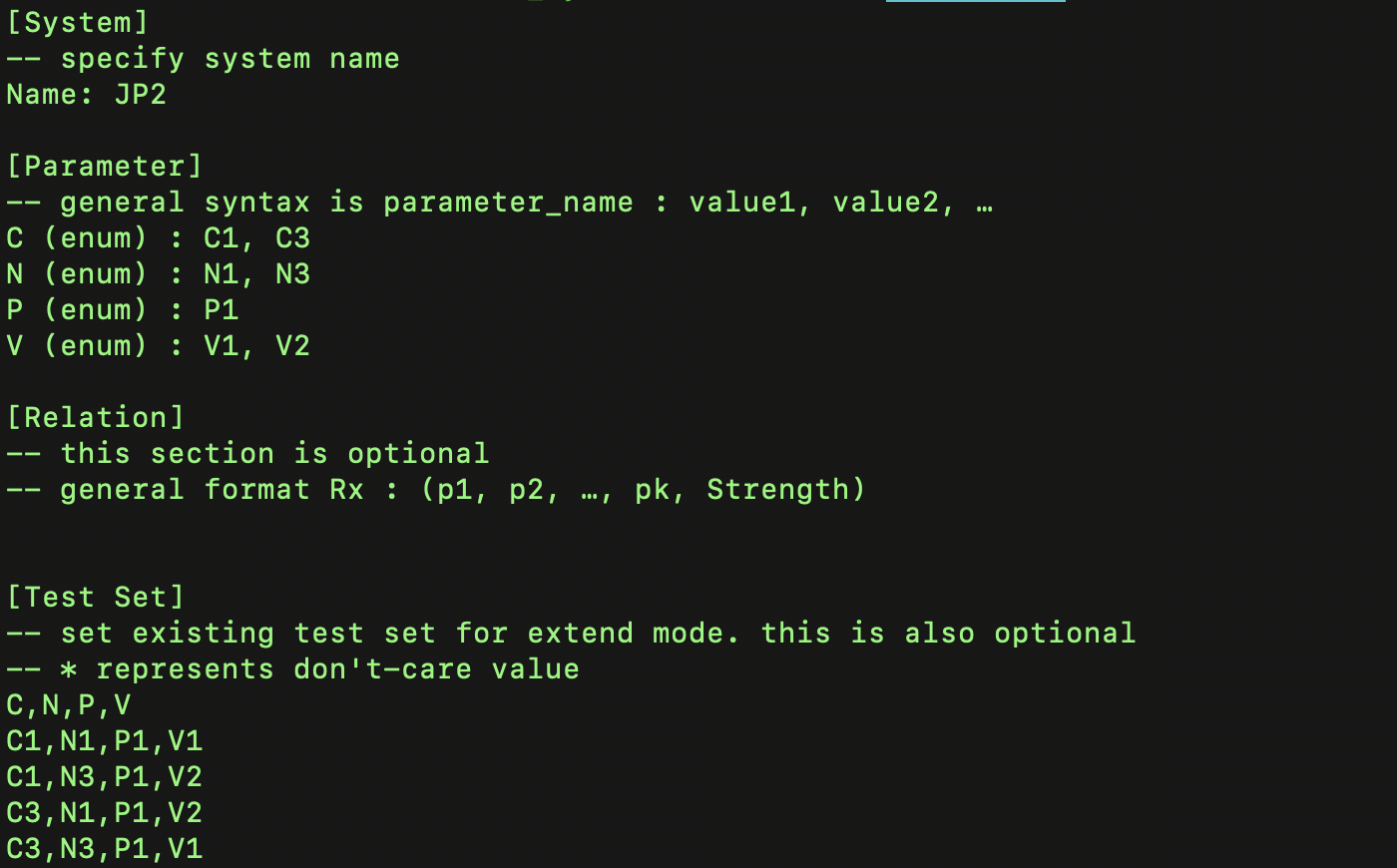
* **Para JP1 y t=2**:



* **Para JP2 y t=1**: Para JP2 no tenemos en cuenta los valores no válidos, por tanto eliminamos C2, N2, P2, P3 y V2.



* **Para JP2 y t=2**:



La diferencia principal entra JP1 y JP2 radica en cómo manejan los casos de valores inválidos. JP1 los incluye en los casos de prueba generados, lo que permite probar cómo responde el programa a estas situaciones. Por otro lado, JP2 los excluye y se enfoca en probar el comportamiento normal del programa, con la expectativa de que cualquier valor invalido genere una excepción.

Las implicaciones prácticas de estas diferencias son:

* JP1 ayuda a identificar posibles fallos o vulnerabilidades en el programa cuando se enfrenta a entradas inesperadas o inválidas.
* JP2 se centra en garantizar que el programa se comporte correctamente bajo condiciones normales, asumiendo que cualquier entrada inválida será manejada adecuadamente mediante excepciones u otro mecanismo de detección de errores.
* Combinados, JP1 y JP2 proporcionan una cobertura completa de las diferentes situaciones de entrada y ayudan a garantizar la robustez y fiabilidad del algoritmo de búsqueda exhaustiva ante una amplia gama de escenarios.

Generamos entonces un archivo csv de cada juego de pruebas alternativo para utilizarlo en Python con la librería pandas.

* 1. **Pruebas de unidades con pytest**

Para la implementación de los test, es necesario interpretar el fichero csv para cada juego de pruebas y asignar los valores correctos para cada caso de test. Para ello, he implementado una función que asigna el valor correcto para cada identificador de valor. La función es la siguiente:

**def genera\_valores(*id*):**

**if *id* == "C1":**

**return random.randint(1,20)**

**if *id* == "C2" or *id* == "P3":**

**return -2**

**if *id* == "C3" or *id* == "N3" or *id* == "P2" or *id* == "V3":**

**return 0**

**if *id* == "N1":**

**return random.randint(1,5)**

**if *id* == "N2":**

**return -1**

**if *id* == "P1" or *id* == "V1":**

**return random.randint(1,10)**

**if *id* == "V2":**

**return -3**

**else:**

**return "Error ID valor"**

**test\_ACTS\_JP1**: primero lee el fichero ‘JP1-output.csv’ generado por ACTS, que contiene los casos de prueba, donde cada fila representa un conjunto de valores de entrada. A continuación, se inicializan 3 variables booleanas a False para rastrear si se encuentran errores. Seguidamente, se itera sobre cada fila del DataFrame “df”, que contiene los casos de prueba, donde, para cada fila, se extraen los identificadores de las variables de entrada que se utilizarán para generar los valores reales de las variables. A continuación, para cada fila, se generan los valores de entrada reales utilizando la función anterior. Se intenta ejecutar el algoritmo de búsqueda exhaustiva con los valores generados para la mochila. Si se produce una excepción, se establecen las variables de control de fallos según corresponda. Por último, se realizan las aserciones finales para verificar si se detectaron fallos en todas las variables. Si alguna está a True (no hubo fallos en dicha variable), la aserción fallará y se señalará que ha habido un error en las pruebas.

**test\_ACTS\_JP2**: De la misma manera, la función comienza leyendo el fichero csv correspondiente ‘JP2-output.csv’ generado por ACTS. Se itera sobre cada fila de ‘df’. Para cada fila, se extraen los identificadores de las variables de entrada (id\_capacidad, id\_num\_articulos, id\_peso, id\_valor) que se utilizarán para generar los valores reales de las variables. Seguidamente, para cada fila se generan los valores de entrada reales con la función anterior. Luego, se intenta ejecutar el algoritmo de búsqueda exhaustiva. Si se detecta un error durante la ejecución del algoritmo de búsqueda exhaustiva (es decir, si se produce una excepción ValueError), se aserta (assert) que la prueba ha fallado. Esto se hace mediante la instrucción assert False. Además, se agrega otra aserción al final de la iteración para garantizar que la prueba se complete correctamente. Esto se hace para asegurarse de que todas las filas del archivo CSV se procesen y no se interrumpa la ejecución.

1. **Solución tarea 2: Prueba búsqueda\_con\_poda y algoritmo\_voraz**
   1. **Testing diferencial**

La función ‘***genera\_aleatorio’*** genera una instancia aleatoria de un problema de la mochila. Toma como entrada el número de artículos ‘*numArticulos*’ y crea una mochila con un conjunto aleatorio de artículos, donde cada uno tiene un valor y un peso aleatorio entre ciertos límites. Después de generar los artículos, ajusta la capacidad de la mochila para que sea la mitad del total de los pesos de los artículos y devuelve la mochila creada.

La prueba ‘***test\_busqueda\_con\_poda***’ ejecuta el algoritmo de búsqueda exhaustiva y el algoritmo de búsqueda con poda 100 veces cada uno. Para cada ejecución, genera una mochila aleatoria con un número aleatorio de artículos y luego ejecuta ambos algoritmos para obtener soluciones (s1 y s2) y valores de solución (v1 y v2). A continuación, comprueba si los valores de solución son iguales y si los artículos seleccionados en ambas soluciones son los mismos. Si hay alguna discrepancia, imprime los resultados (utilizando la opción *pytest -s*).

La prueba ‘***test\_algoritmo\_voraz***’ compara el algoritmo de búsqueda exhaustiva con el algoritmo voraz. El proceso es el mismo que en la función anterior: genera una mochila, ejecuta ambos algoritmos y compara los resultados.

La función ‘***imprimir\_resultados***’ imprime los resultados de las pruebas cuando se encuentran discrepancias entre las soluciones de los algoritmos. El objetivo es saber si ha fallado algún algoritmo, o la solución alcanzada por ambos difiere (lo que es normal en varios casos).

**b. Escalabilidad de los algoritmos**

El funcionamiento de ‘**test\_escalabilidad\_exhaustiva\_vs\_poda()**’ es el siguiente: primero, se crea un archivo CSV llamado ‘escalabilidad\_exhaustiva\_vs\_poda.csv’ para almacenar los resultados, y se escribe una fila de encabezado en el fichero csv que describe las columnas. A continuación, se itera sobre una serie de valores para el número de artículos en la mochila, desde 5 hasta 17, donde, para cada valor de ‘numArticulos’ se genera una mochila aleatoria. Seguidamente, se mide el tiempo de ejecución de la búsqueda exhaustiva y se registra el valor óptimo y el tiempo de ejecución en el archivo CSV. A continuación, se mide el tiempo de ejecución de la búsqueda con poda y se registra el valor óptimo y el tiempo de ejecución en el archivo CSV.

El funcionamiento de ‘***test\_escalabilidad\_poda\_vs\_voraz***’ consiste en, primero, crear un fichero csv, como indica el enunciado, para almacenar los resultados. A continuación, se escribe en el fichero una fila de encabezado que describe las columnas. Seguidamente, se itera sobre una serie de valores para el número de artículos en la mochila, desde 5 hasta 17, donde, para cada valor, se genera una mochila aleatoria. Por último, se mide el tiempo de los algoritmos y se registra el valor óptimo y el tiempo de ejecución de ambos en sendos ficheros.

Para **representar gráficamente los test de escalabilidad** (utilizando *pandas* y *matplotlib*), he implementado el fichero ‘***graficas.py***’, cuya ejecución se debe realizar una vez creados los ficheros csv con el resultado de los tests.

Gráfico, Gráfico de líneas

Descripción generada automáticamente

En el primer subgrafo, se compara el tiempo de ejecución de la búsqueda exhaustiva y la búsqueda con poda, y en el segundo, el tiempo de ejecución de la búsqueda con poda y el algoritmo voraz.

Se puede apreciar que el algoritmo más rápido es el algoritmo voraz. El menos eficiente es búsqueda exhaustiva, que tiene una complejidad temporal exponencial en función del número de artículos. El algoritmo búsqueda con poda tiene una complejidad de , que es más eficiente que búsqueda exhaustiva, pero no tanto como el algoritmo voraz, cuya complejidad temporal es solo durante la parte de ordenación, después, recorre los artículos con un bucle simple, que tiene una complejidad de .

1. **Solución tarea 3: Implementación y prueba de una solución basada en programación dinámica**

El funcionamiento de la solución basada en programación dinámica que he implementado es el siguiente: primero, se calcula la longitud *n* de la lista de artículos y se guarda la capacidad total de la mochila.

A continuación, se crea una tabla bidimensional ‘*tabla*’ de dimensiones:

Esta tabla se utiliza para almacenar los valores óptimos para subproblemas más pequeños del problema normal.

Seguidamente, se itera sobre cada artículo y sobre cada capacidad posible de la mochila. Para cada subproblema representado por una celda *tabla[i][j]*, se calcula el valor óptimo considerando si se debe incluir o no el artículo *i* en la mochila. Se comienza desde la celda en la esquina inferior derecha de la tabla y se sigue un camino hacia arriba y hacia la izquierda, seleccionando aquellos artículos que contribuyeron al valor óptimo.

A continuación, se crea una nueva instancia de la clase Mochila, que contiene solo los artículos seleccionados durante el proceso de reconstrucción.

Finalmente, se devuelve la nueva mochila con los artículos seleccionados y el valor total máximo que se puede obtener.

El funcionamiento del test ‘***test\_programacion\_dinamica***’, primero, crea una instancia de la clase Mochila con una capacidad predeterminada de 10. Se insertan varios artículos en la mochila, cada uno con un valor y un peso específicos. A continuación, se aplica el algoritmo de programación dinámica a la mochila de prueba. Esto devuelve una nueva mochila ‘*mochila\_resuelta*’ que contiene los artículos seleccionados según la solución óptima, junto con el valor total óptimo obtenido.

Seguidamente, se verifica que la capacidad de la mochila resuelta sea igual a la capacidad original especificada en la prueba. A continuación, se verifica que la suma de los pesos de los artículos seleccionados en la mochila resuelta no exceda la capacidad de la mochila. Por último, se compara el valor óptimo obtenido con un valor esperado predefinido. En este caso, el valor esperado es 90, que es el valor óptimo para la instancia de la mochila de prueba.

Los **test de escalabilidad** contra el algoritmo voraz y búsqueda con poda siguen el mismo esquema que los de búsqueda exhaustiva.

Se puede obtener el grafico resultante de este test ejecutando ‘*graficas.py*’, cuyo funcionamiento se ha explicado anteriormente.

Gráfico, Gráfico de líneas

Descripción generada automáticamente

Vemos que el algoritmo voraz sigue siendo más rápido, ya que la complejidad temporal del algoritmo de programación dinámica es , donde n es el número de artículos y W es la capacidad total de la mochila.

1. **Apendice A**

Es importante que la ubicación de ficheros y directorios siga el siguiente esquema para el correcto funcionamiento de la práctica:­­

