

UPS Inteligente Utilizando Grafos

Alexander Guacán^[L00412289]

Universidad de las Fuerzas Armadas
adguacan@espe.edu.ec

Abstract.

El presente documento ha diseñado un programa que simula el funcionamiento de un Uninterruptable Power Supply (UPS), que a su vez se adaptó a un determinado escenario empleando el uso de grafos. Se analizó y creo dos algoritmos de búsqueda que determina el camino más corto a recorrer por un grafo desde un punto a otro. El escenario planteado es un cyber, el UPS tuvo la tarea de conocer el camino más corto por el circuito hasta un determinado dispositivo, para preservar su duración de encendido cuando el UPS usa su batería interna para suministrar energía eléctrica. Los resultados fueron los esperados pero queda en duda que algoritmo creado es el más óptimo o recomendado a utilizar en el escenario planteado.

1 Introducción

Un UPS, por su traducción al español, es un Sistema de Alimentación Ininterumpida. Este dispositivo permite suministrar de energía eléctrica por un tiempo definido, cuando el flujo normal falla. Los fallos pueden darse cuando se va la luz o un cortocircuito, o cuando existe una sobrecarga en la energía eléctrica suministrada por el tomacorriente de la pared. Ahí es donde interviene el UPS, el cual se encarga de bloquear este mal flujo de corriente de energía eléctrica y suministrar su propia fuente de alimentación dado por una batería interna [7].

Por otra parte, un grafo es una estructura de datos que está representada a través del enlace de nodos, estos enlaces son conocidos como aristas. Cada nodo almacena cualquier tipo de información, y los datos de las aristas que posee [4]. Por su parte, la teoría de grafos se encarga de estudiar el comportamiento y relación de estos últimos, como parte de una rama de las matemáticas aplicada en ciencias de la computación [6]. Sus aplicaciones varían desde, buscar una ruta de un punto a otro, ya sea la más larga o corta. También se pueden utilizar en el manejo de las redes sociales, ya que al conocer los diferentes enlaces con amigos y patrones de sitios web visitados por el usuario, el algoritmo puede brindar de información más acorde al cliente sobre lo que desea consumir.

En este laboratorio mezclaremos ambos conceptos, el objetivo será generar un sistema inteligente que simule el funcionamiento de un UPS. Establecer un escenario de funcionamiento del UPS en el cual pueda gestionar, a través del uso de grafos, un uso adecuado de su principal característica que es brindar energía

eléctrica temporal. Comprender algoritmos de búsqueda aplicado a los grafos que permitan determinar el camino más corto de un punto a otro. El escenario que se plantea para este proyecto es un cyber. El principal problema será determinar el camino más corto desde el UPS hasta un cierto dispositivo para suministrar energía directa al mismo para que su duración de encendido se incremente.

2 Desarrollo

Para diseñar el proyecto, primero comenzaremos explicando cada parte que conforma el UPS. Puede observar en la Figura 1, de forma simplificada, cada uno de los componentes que conforman un UPS. Es necesario especificar que existen distintos tipos de UPS y cada uno con capacidades distintas, en esta ocasión las características con las que cuenta el UPS a modelar son que sus voltajes de entrada que vienen desde el tomacorriente deben fluctuar entre 100V y 140V. Además su voltaje de salida debe ser de 120V, donde su batería recibe y otorga un voltaje de 12V con una intensidad de 9Ah (Amperio/hora).

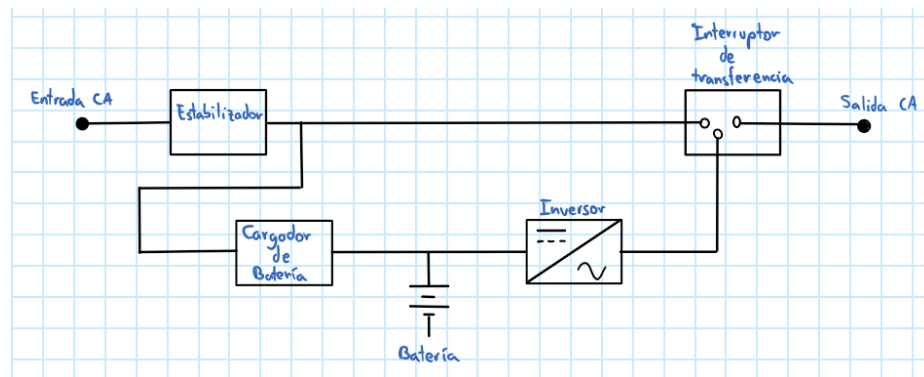


Fig. 1. Circuito eléctrico y componentes que conforman un UPS.

El UPS cuenta con 3 modos o estados. El primero es el modo normal, observe Figura 2, aquí el voltaje está dentro de los límites especificados en el anterior párrafo y la carga de la batería está completa. Por lo que únicamente el estabilizador se encargará de regular el voltaje del tomacorriente a 120V [5]. Se debe aclarar que el *Interrupción de transferencia* únicamente sirve como puente entre los diferentes canales de corriente. En este modo se tomará la corriente desde el estabilizador, dado que no hay ningún problema con el voltaje leído desde el tomacorriente de pared [1].

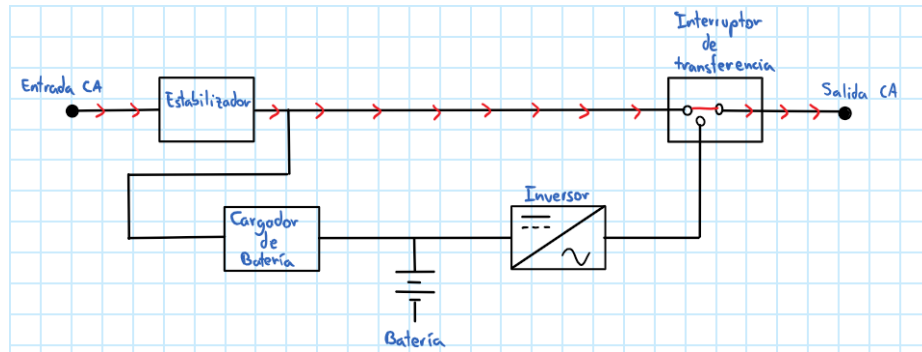


Fig. 2. Fluctuación de energía eléctrica en modo normal.

El siguiente modo con el que cuenta el UPS es el modo normal pero sin carga. Este modo se refiere cuando el voltaje leído desde el tomacorriente cumple con los estándares estables, pero el porcentaje de la batería es menor al cien por ciento. Para ello se comienza a cargar la batería a la vez que se suministra de forma normal energía eléctrica a los dispositivos. Observe que en la Figura 3 antes de que la corriente eléctrica del estabilizador llegue a la batería, debe pasar por su respectivo cargador. Este dispositivo se encarga de disminuir la corriente eléctrica de 120V a 12V que es lo necesario para cargar la batería [2].

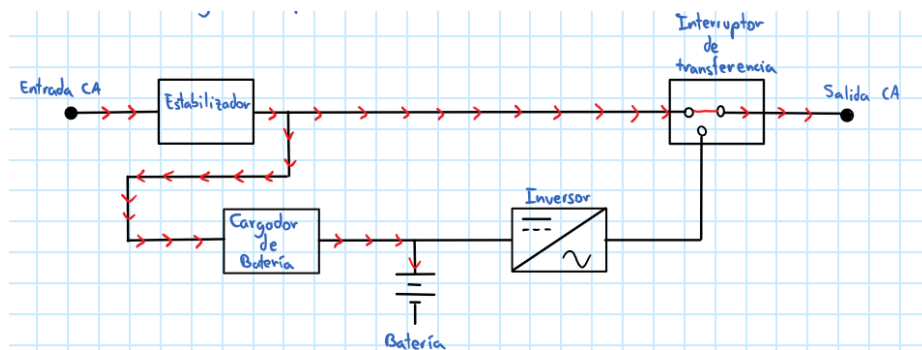


Fig. 3. Fluctuación de energía eléctrica en modo normal pero con carga de batería incompleta.

Por último tenemos el modo inversor. En este modo el voltaje leído del tomacorriente se sale de los límites, existiendo un cortocircuito (menor a 100V) o una sobrecarga (mayor a 140V). Cuando se presenta uno de estos dos escenarios se comienza a utilizar la batería interna del UPS. Observe en la Figura 4 que ahora

la energía eléctrica parte de la batería y pasa por el inversor, de ahí el nombre de dicho modo. El inversor se encarga de transformar la corriente directa que proporciona la batería en corriente alterna incrementada a 120V, necesarios para el correcto funcionamiento de los dispositivos [3]. Es aquí cuando el estado de *Interruptor de transferencia* cambia, haciendo un puente desde el estabilizador hacia la salida de corriente del UPS.

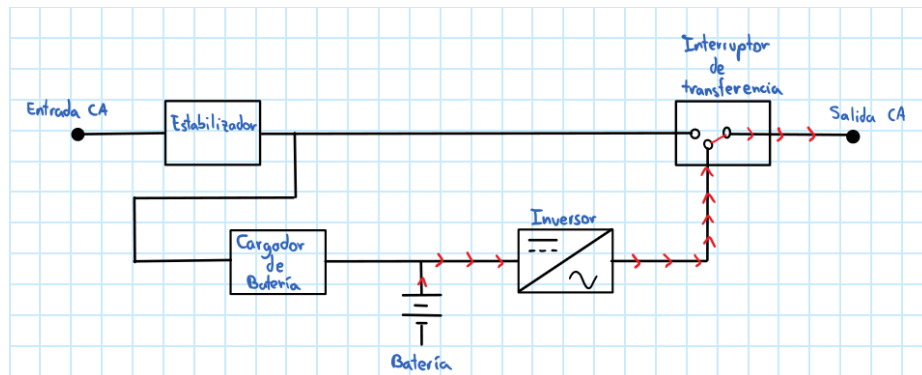


Fig. 4. Fluctuación de energía eléctrica en modo inversor.

El grafo que simulará la distribución de los dispositivos en el cyber lo puede observar en la Figura 5. Aquí podemos que el nodo cero representa al UPS y todos los demás nodos serán los dispositivos conectados a él. Observe también que un dispositivo puede ser otro "conector" pero en realidad lo que se refiere es que donde está conectado el dispositivo n , también existe otro conector junto como el de un cortapicos por el cual se puede ampliar la conexión de los dispositivos. Para hacerlo más flexible, permitiremos al usuario configurar el dispositivo que considere como principal.

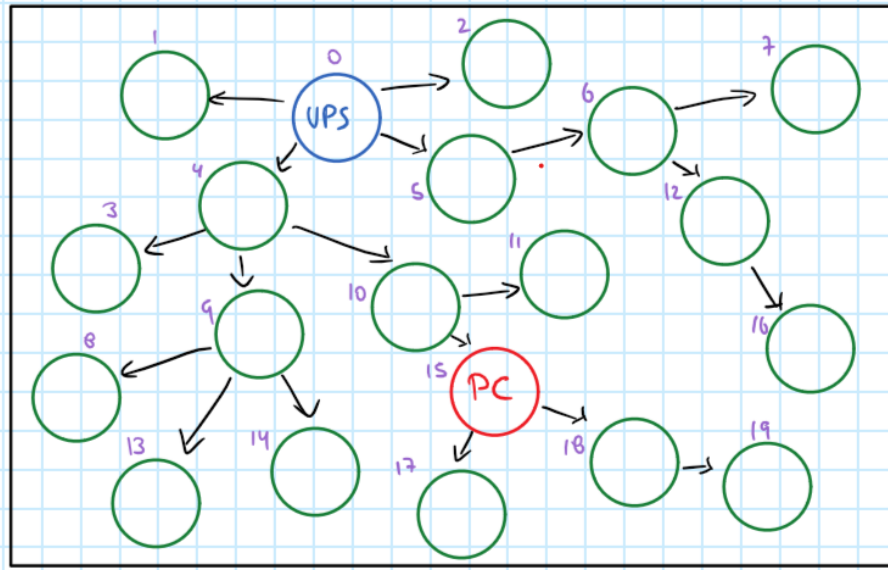


Fig. 5. Distribución de los dispositivos de un cyber en forma de grafo.

Bien, el modelado de este problema respecto al UPS se ha optado por el patrón observador. Este patrón trata de simular una suscripción a un canal, donde el canal será el componente observable y los suscriptores serán los observadores. Así pues en nuestro caso el UPS será el objeto observable y la pantalla donde se muestran las estadísticas será el observador. Puede observar en la Figura 6 que utilizamos interfaces para hacer una conexión entre ambos conceptos ya que de primeras no conocemos que tipo de elementos pueden observar o que componentes pueden ser observados. La interfaz *Observable* cuenta con metodos para agregar, eliminar y notificar de cambios a los observadores, y por su parte la interfaz *Observer* cuenta con un solo método el que se encargará de extraer la información de interés del objeto al cual observa.

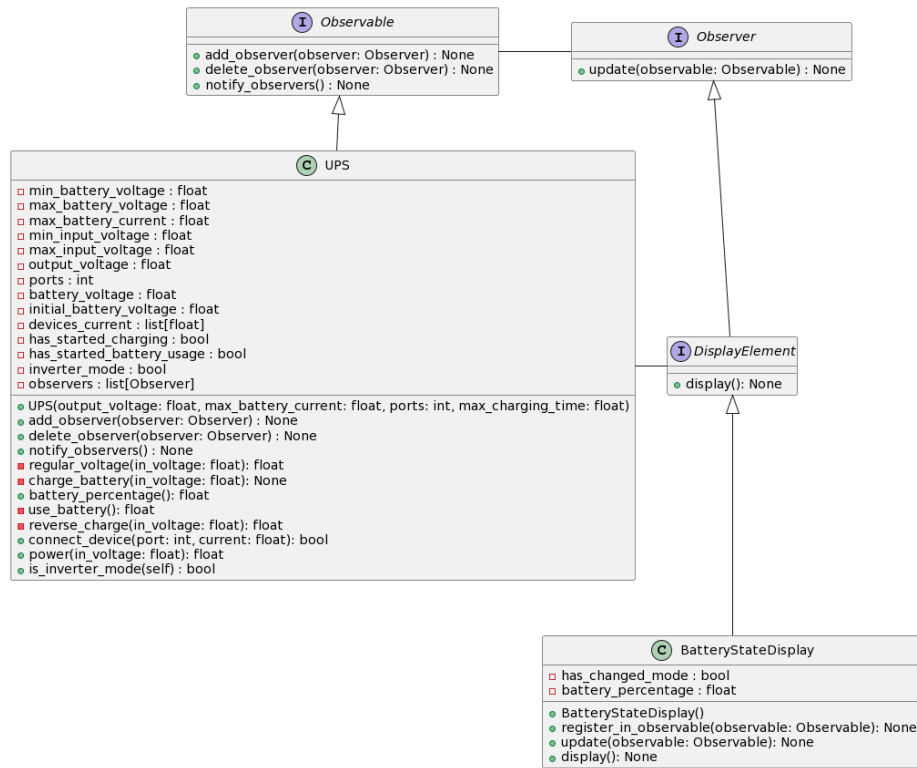


Fig. 6. Diagrama de clases del UPS utilizando el patrón observador.

Antes de explicar el código recuerde que puede encontrar el código completo en **GitHub**. Asimismo puede observar una explicación más detallada de todo el proyecto en el siguiente **Video**. Para crear el grafo se ha empleado una lista de adyacencias, la cual consiste en que cada lista en una serie de listas contiene los nodos a los cuales se conecta cada uno de ellos. El primer algoritmo de búsqueda empleado es la búsqueda en profundidad, observe Listing 1.1, el cual se mueve de manera recursiva desplazando por el grafo hasta encontrar el nodo objetivo o un nodo hoja. En este algoritmo se encuentran todos los posibles caminos desde el nodo inicial hasta el nodo objetivo por lo que será necesario de un filtro para encontrar el más corto requerido por el programa.

```

1  def __paths_recursively(self, current_node: int,
2      objective_node: int, paths: list[list[int]], path = 0) ->
3      tuple[list[list[int]], int]:
    """
    Retorna todos los caminos posibles desde un punto o
    nodo inicial hasta el nodo objetivo o hasta que llegue a
    un nodo hoja
  
```

```

4
5     Args:
6         current_node (int): _description_ Nodo en el que
se encuentra en el grafo
7         objective_node (int): _description_ Nodo final al
que se desea llegar
8         paths (list[list[int]]): _description_ Lista de
caminos posibles
9         path (int, optional): _description_. Defaults to
0. Numero de camino posible, sirve como indice para la
lista de caminos
10
11     Returns:
12         tuple[list[list[int]], int]: _description_
Retorna los caminos y el numero de caminos
13     """
14     # Si un no ya ha sido recorrido
15     if current_node in paths[path]:
16         return paths, path + 1
17
18     # Agregamos el nodo que estamos accediendo al
recorrido
19     paths[path].append(current_node)
20
21     # Si el nodo a recorrer es el nodo objetivo
22     if current_node == objective_node:
23         return paths, path + 1
24
25     # Si el nodo tiene uno o mas de un hijo
26     for son_node in self.__nodos[current_node]:
27         path = self.__paths_recursively(son_node,
objective_node, paths, path)[1]
28
29     # Es el ultimo hijo del nodo
30     if son_node == self.__nodos[current_node][-1]:
31         return paths, path
32
33     # Creando un nuevo camino
34     paths.append([])
35
36     # Itera la rama padre
37     i = 0
38     # Punto de quiebre con el nodo padre del nodo
actual
39     node = -1
40
41     # Copiando la rama padre
42     while node != current_node:
43         # Agregando al nuevo camino los nodos
recorridos anteriores a esta nueva ramificacion

```

```

44         paths[path].append(paths[path - 1][i])
45         # Avanzamos al siguiente nodo en la lista
recorrida
46         node = paths[path - 1][i]
47         # Incrementamos la posicion de la lista de
nodos recorridos
48         i += 1
49
50     return paths, path + 1

```

Listing 1.1. Búsqueda en profundidad. Archivo Grafo.py.

El segundo algoritmo de búsqueda es por anchura. Este algoritmo en vez de moverse de forma recursiva lo que hace es que visita todos los nodos vecinos de un determinado nodo y si no encuentra el nodo objetivo se sigue desplazando por los demás nodos vecinos que contengan más vecinos. En este caso, el camino que devuelve es el primero que encuentra por lo que puede que no sea el más corto de todas las posibilidades. Observe Listing 1.2.

```

1     def breadth_first_search(self, start: int, end: int) ->
list[int]:
2         """
3         Retorna el primer camino desde el nodo start hasta el
nodo end
4
5         Args:
6             start (int): _description_ Nodo desde donde
empieza el recorrido por el grafo
7             end (int): _description_ Nodo donde termina el
recorrido del grafo
8
9         Returns:
10            list[int]: _description_ Recorrido en orden por
cada nodo hasta llegar al nodo end. Retorna [] si no hay
camino disponible
11            """
12            # Camino desde start a end
paths = [[int(start)]]
13            # Lista de nodos visitados
visited_nodes = list[int]()
14            # Nodos vecinos por visitar de cada nodo
queue = list[int]()
15
16            # Se recorrera los nodos vecinos de start
queue.append(start)
17            # Nodo start ya se ha visitado
visited_nodes.append(start)
18
19            last_path = 0
20
21            # Mientras haya nodos vecinos por recorrer
22
23
24
25
26

```



```

27     while len(queue) > 0:
28         # Seleccionamos cada nodo no visitado en el orden
           que se encuentran
29         current_node = queue.pop(0)
30
31         # Recorremos todas las rutas para ver con cual
           conecta
32         for path in paths:
33             # Verificamos el nodo de conexion directa al
           nodo actual
34             for node in path:
35                 # Sigue la conexion del ultimo nodo
36                 if current_node in self.__nodos[node] and
           node == path[-1]:
37                 # Incrementamos el camino recorrido
38                 path.append(current_node)
39                 last_path = paths.index(path)
40                 # Bifurcacion de un nodo
41                 elif current_node in self.__nodos[node]
           and node != path[-1]:
42                 # Copiamos la ruta anterior a la
           bifurcacion
43                 paths.append(path[:path.index(node) +
           1])
44                 # Actualizamos el ultimo camino
           recorrido
45                 last_path = len(paths) - 1
46                 # Se encuentra el nodo objetivo
47                 if current_node == end:
48                     # Ultimo camino recorrido
49                     return paths[last_path]
50
51                 # Recorremos cada nodo vecino al nodo actual
52                 for neighbor in self.__nodos[current_node]:
53                     # Nodo vecino no ha sido visitado
54                     if neighbor not in visited_nodes:
55                         # Nuevo nodo vecino a ser recorrido
56                         queue.append(neighbor)
57                         # Nodo visitado
58                         visited_nodes.append(neighbor)
59
60                 # No se encontro un camino desde start hasta end
61                 return []

```

Listing 1.2. Búsqueda en anchura. Archivo Grafo.py.

Pasando al código del UPS se ha encapsulado el funcionamiento de cada dispositivo en metodos ya que los datos que procesan son solo de lectura que van dependiendo de cada uno de los componentes en el orden en que es procesado el voltaje del tomacorriente. Observe en el Listing 1.3 que la función tiene como

dato de entrada el voltaje dado por el tomacorriente, posteriormente este es procesado de acuerdo a si está dentro del rango de voltaje estable y si la batería falta de cargar. Caso contrario comienza el modo inversor y la batería empieza a utilizarse hasta que se desgaste por completo.

```
1  def power(self, in_voltage: float) -> float:
2      """
3          Enciende el UPS
4
5          Args:
6              in_voltage (float): _description_ Voltaje
              suministrado por el tomacorriente
7
8          Returns:
9              float: _description_ Voltaje de salida necesario
              para alimentar los dispositivos especificado por el
              fabricante
10         """
11         # Voltaje de salida que sera tomado del tomacorriente
            o por la bateria del UPS
12         out_voltage = float()
13
14         # Voltaje dentro de los limites que no generan un
            cortocircuito o sobrecarga establecidos por el fabricante
15         if in_voltage >= self.__min_input_voltage and
            in_voltage <= self.__max_input_voltage:
16             self.__has_started_battery_usage = False
17             self.__inverter_mode = False
18             # Voltaje de salida tomado del tomacorriente y
            regulado por el estabilizador
19             out_voltage = self.__regular_voltage(in_voltage)
20
21             # Bateria con carga incompleta
22             if self.battery_percentage() < 100:
23                 # Cargar bateria con voltaje regulado por
                estabilizador
24                 self.__charge_battery(self.__regular_voltage(
                in_voltage))
25
26             # Voltaje del tomacorriente no valido y la bateria
                tiene carga
27             elif self.battery_percentage() > 0:
28                 self.__has_started_charging = False
29                 self.__inverter_mode = True
30                 # Voltaje de salida tomado por la bateria e
                incrementado por el inversor
31                 out_voltage = self.__reverse_charge(self.
                __use_battery())
32                 # Voltaje de tomacorriente no optimo y no hay bateria
                disponible
```

```
33         else:
34             out_voltage = 0
35
36             self.notify_observers()
37
38             return out_voltage
39
40     def is_inverter_mode(self) -> bool:
41         return self.__inverter_mode
```

Listing 1.3. Gestión de UPS. Archivo UPS.py.

Para fusionar ambos conceptos debemos tener en cuenta que es necesario construir el grafo y determinar la cantidad de corriente que consume todos los dispositivos. Para ello se optó por guardar dicha información en archivos planos de texto. Para el grafo observe la Figura 7 aquí cada línea representa un nodo, y cada lista de números representa los nodos a los cuales se conecta.

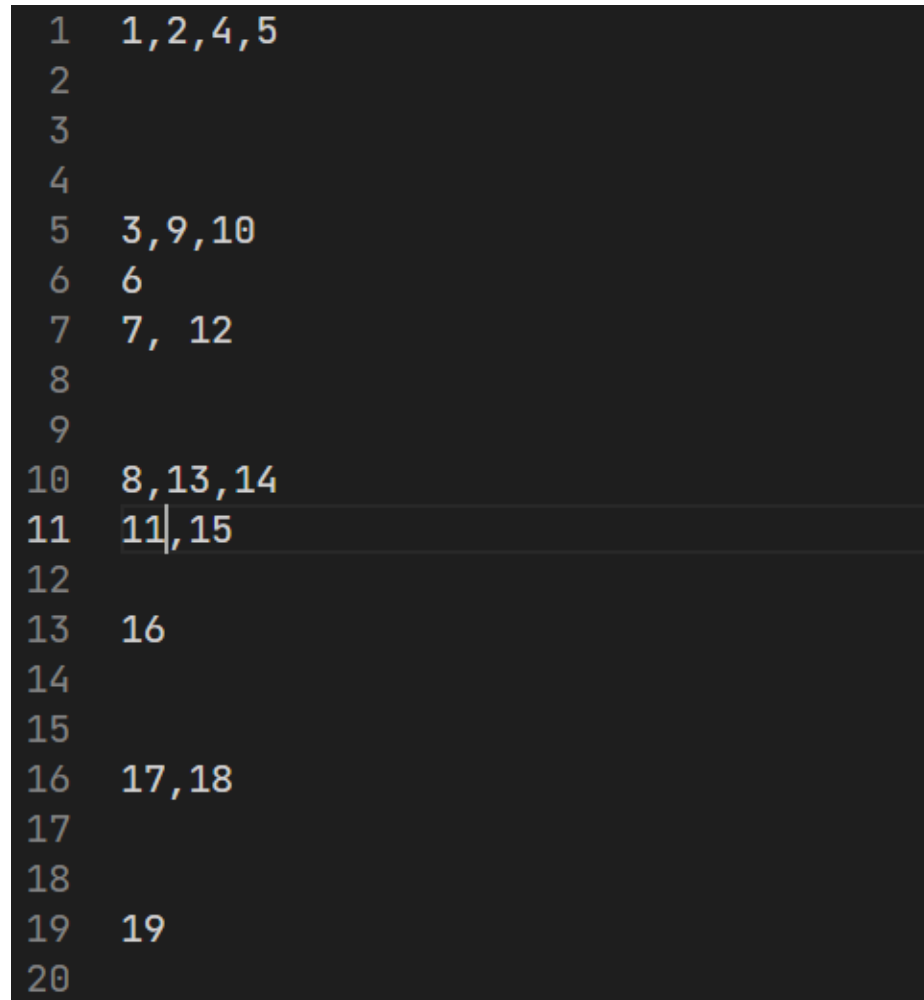


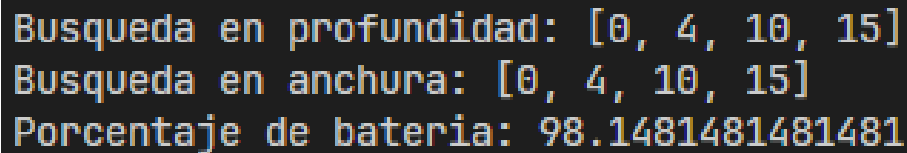
Fig. 7. Grafo adaptado en un archivo plano de texto.

Observe el Listing 1.4, aquí simulamos la lectura del tomacorriente obteniendo números random. Por lo tanto en el momento en el que el voltaje se salga de los límites establecidos el UPS pasará al modo inversor y operará hasta que termine su ejecución. Puede ver su ejecución en la Figura 8 que se adapta al grafo de la Figura 5.

```
1 def power_ups(self) -> None:
2     """
3     Comenzar las operaciones del UPS
4     """
```

```
5         # No se configuro el ups
6         if not self.__is_ups_configured:
7             return print("UPS no configurado")
8
9         # Lectura del tomacorriente
10        power_supply_voltage = 0
11        # Camino por algoritmo de busqueda por profundidad
12        deep_path = self.__graph.shortest_path(0, self.
__main_device)
13        # Camino por algoritmo de busqueda por anchura
14        breadth_path = self.__graph.breadth_first_search(0,
self.__main_device)
15
16        # Seguir operando mientras exista bateria
17        while self.__ups.battery_percentage() > 0:
18            # Limpiar pantalla
19            os.system("cls")
20            # Voltaje del tomacorriente dentro de los limites
permitidos
21            if not self.__ups.is_inverter_mode():
22                # Leer voltaje del tomacorriente
23                power_supply_voltage = self.
__read_power_supply_voltage()
24                print(f"Voltaje de tomacorriente: {
power_supply_voltage}")
25            # Sobrecarga o cortocircuito del tomacorriente de
pared. Se deja de leer el voltaje del tomacorriente
26            else:
27                print("Busqueda en profundidad:", deep_path)
28                print("Busqueda en anchura:", breadth_path)
29
30            # Enviar dato del voltaje del tomacorriente
31            self.__ups.power(power_supply_voltage)
32            # Pausar ejecucion del programa por un segundo
33            time.sleep(1)
```

Listing 1.4. Lectura de tomacorriente y gestion de UPS. Archivo System.py.



```
Busqueda en profundidad: [0, 4, 10, 15]
Busqueda en anchura: [0, 4, 10, 15]
Porcentaje de bateria: 98.1481481481481
```

Fig. 8. Grafo adaptado en un archivo plano de texto.

3 Discusión

Se exploraron solo dos algoritmos de búsqueda por lo que el programa podría mejorar en ese aspecto. Además para conocer cual de los dos algoritmos empleados es mejor, sería idóneo hacer más pruebas con grafos mucho más grandes. Otra forma de determinar el mejor algoritmo es calcular su complejidad tanto espacial como temporal a través de librerías como *Big O* de Python. Por otra parte resulta un poco confuso el uso de archivos para configurar el grafo, si bien es una alternativa rápida y simple para no tener que depender del todo del programa, si el programador o usuario que intente configurar desconozca por completa del tema, terminaría rompiendo el sistema.

El UPS si bien cumple su función, se debe recordar que solo es una simulación del verdadero trabajo que realiza, así que los cálculos empleados pueden variar en un escenario real. El sistema resulta algo incompleto en cuestión de mostrar todas las estadísticas del UPS, pero haría falta utilizar conceptos más avanzados como lo son hilos para usar el programa mientras simultáneamente se muestran dichas estadísticas. También cabe aclarar que el emplear un archivo donde se guarden los datos de las intensidades de cada uno de los dispositivos es por practicidad, dado que toda esa información sería directamente proporcionada por los sensores.

Utilizar patrones de diseño resulta bastante útil ya que nos permite generalizar un programa y hacerlo flexible al cambio con el tiempo. Pero no siempre debe emplearse en todo momento ya que puede resultar contraproducente y terminar haciendo más complejo y abstracto el programa. No hay que forzar el uso de patrones, con la experiencia se puede identificar a simple vista cada uno de ellos, y es ahí donde es correcto emplearlos.

4 Conclusiones

Dado la naturaleza de un circuito es mucho más complicado crear un grafo que sea bidireccional, por lo que la complejidad del problema no podría escalar demasiado en el escenario propuesto. Los grafos, si bien son una herramienta muy potente, no siempre puede aplicarse en todos los escenarios, pueden existir otras estructuras que den solución más rápidas o más livianas, en terminos de complejidad temporal. El objetivo siempre a la hora de encontrar una solución es buscar el equilibrio entre eficiencia y eficacia.

References

1. Faraday Electrónica. Cómo funciona el ups off-line. diagrama a bloques y funcionamiento., Jun 2020.
2. Luis Carlos Galán. CÓmo funciona un cargador de baterías. desarmando para comprender., Aug 2015.
3. Mentalidad De Ingeniería. Inversor de corriente explicado, Nov 2020.
4. Orlando. Qué son los grafos, Jul 2019.

5. VOGAR Reguladores. ¿cómo funciona un regulador de voltaje?, Feb 2020.
6. Paula Rochina. Teoría de grafos: Análisis relacional de las redes sociales, Mar 2017.
7. Transelec. Qué es un ups y como funciona, 2020.