PRACTICA DE BUSQUEDA : PROYECTO DE INTELIGENCIA ARTIFICIAL 2021

Grupo 3

Componentes:

-Jorge Sáenz De Miera

-Nico Vega Muñoz

-Javier Gil Domínguez

-David Lázaro Martín

-Pablo Martín Escobar

INDICE DE LA MEMORIA

1. PRESENTACIÓN

El proyecto consiste en diseñar una aplicación para hallar el trayecto óptimo entre dos estaciones metro de Kiev, para ello debemos tener en cuenta varios parámetros, como puede ser el número de estaciones y líneas, el tiempo que tarda el metro de media en recorrer cada tramo, la distancia de una estación a otra, los transbordos, qué día o a qué hora se realiza el trayecto, etc. Para el desarrollo de la practica vamos a utilizar el algoritmo A*, un algoritmo muy conocido y usado en Inteligencia Artificial para optimizar el cálculo de caminos mínimos.

En este proyecto además del cálculo teórico de estos caminos, se busca implementar un algoritmo que permita calcular la ruta más eficiente para cada origen y destino del metro de Kiev, así como una interfaz gráfica desde la cual poder hacer todos eso cálculos como cualquier otra aplicación. Por lo que debemos programar el uso reiterativo de dicho algoritmo, según las indicaciones podremos utilizar el lenguaje de programación y herramientas que deseemos.

2. RESOLUCIÓN Y DESCRIPCIÓN DEL PROYECTO

2.1- Enfoque

Una vez decidido que debemos utilizar el algoritmo A* para nuestra aplicación, debemos preguntarnos por qué es el idóneo para conseguir el camino mínimo optimo.

El problema de algunos algoritmos de búsqueda en grafos informados, como puede ser el algoritmo voraz, es que se guían en exclusiva por la función heurística, la cual puede no indicar el camino de coste más bajo, o por el coste real de desplazarse de un nodo a otro (como los algoritmos de escalada), pudiéndose dar el caso de que sea necesario realizar un movimiento de coste mayor para alcanzar la solución.

Es por ello bastante intuitivo el hecho de que un buen algoritmo de búsqueda informada debería tener en cuenta ambos factores, el valor heurístico de los nodos y el coste real del recorrido.

Así, el algoritmo A* utiliza una función de evaluación f(n) = g(n) +h(n), donde h(n) representa el valor heurístico del nodo a evaluar desde el actual, n, hasta el final, y g(n), el coste real del camino recorrido para llegar a dicho nodo, n, desde el nodo inicial. A* mantiene dos estructuras de datos auxiliares, que podemos denominar *abiertos*, implementado como una cola de prioridad (ordenada por el valor f(n) de cada nodo), y cerrados, donde se guarda la información de los nodos que ya han sido visitados. En cada paso del algoritmo, se expande el nodo que esté primero en abiertos, y en caso de que no sea un nodo objetivo, calcula la f(n) de todos sus hijos, los inserta en abiertos, y pasa el nodo evaluado a cerrados.

El algoritmo es una combinación entre búsquedas del tipo primero en anchura con primero en profundidad: mientras que g(n) tiende a primero en profundidad, h(n) tiende a primero en anchura. De este modo, se cambia de camino de búsqueda cada vez que existen nodos más prometedores.

A la hora de implementar este algoritmo nuestro grupo ha pensado que lo más idóneo es utilizar el lenguaje de programación Python, además utilizaremos una de sus librerías para su visualización

2.2-Ejecución

Para explicar cómo hemos programado el algoritmo y como finalmente hemos desarrollado la aplicación, primero debemos mostrar de donde hemos sacado los datos que nos han permitido llevar a cabo el proyecto.

El coste de ir de una estación a otra es decir la g(n) lo hemos tomado como tiempo, es decir, como en la aplicación lo que se busca es el recorrido mínimo en tiempo no espacio hemos decidido tomar el peso de cada arista del grafo del plano de Kiev, como el tiempo que se tarda de ir a una estación a la siguiente.

Hemos utilizado la app Metro Kyiv para sacar todos los tiempos entre las diferentes estaciones, así como información interesante que nos ha servido de ayuda , como las horas punta .

Aquí está la información principal:

https://docs.google.com/document/d/1GJ8UHOTusJSoPdcV7XXG3m5XqAHa_HNG-KYsiUW5e-Q/edit?usp=sharing

El valor heurístico del nodo a evaluar desde el nodo que este visitando en ese momento, es decir h(n) , lo hemos cogido de la api geocoding , que te muestra la distancia geográfica exacta de una estación a otra , pero como viene dada en distancia y nosotros operamos con tiempo, utilizamos la velocidad media del metro y la distancias obtenidas para conseguir el tiempo, según la página web oficial del metro de Kiev la velocidad media del metro es 36,1 km/h (10m/s) por lo que cualquier distancia la debemos dividir por 10 para conseguir el tiempo . Toda esta información la decidimos guardar en un archivo CSV para luego poder manejar los datos mejor.

Ahora debemos programar el algoritmo A*

Creamos una clase grafo, que lo que haces es crear un grafo entre todas las paradas poniendo de peso de las aristas el tiempo que se tarda de ir de una estación a otra, además tenemos que saber en todo momento cual son los vecinos de cada nodo, por lo que definimos una función que lo saque.

```python
from collections import deque
import pandas as pd
data = pd.read_csv(r'C:\Users\DAVID\Desktop\datos.csv',index_col=0)
tiempo_en_segundos = lambda x, y: x * 60 + y
tiempo_en_minutos = lambda x : str(x//60) +" minutos "+str(x%60)+" segundos "
class Graph:
    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list

    def get_neighbors(self, v):
        return self.adjacency_list[v]
```

Lo primero que hacemos es crear 2 listas, la primera es una lista de nodos que ya han sido visitados , pero cuyos vecino no han sido inspeccionados todos (lista abierta), comienza con el nodo inicial .La segunda lista es de nodos que ha sido visitados y cuyos vecinos ya hayan sido inspeccionados(lista cerrada) .

```python
def a_star_algorithm(self, start_node, stop_node,hora,dia):
    # open_list es una lista de nodos que han sido visitados,pero cuyos vecinos
    # no han sido inspeccionados todos, comienza con el nodo inicial
    # lista_cerrada es una lista de nodos que han sido visitados
    # y cuyos vecinos han sido inspeccionados
    open_list = set([start_node])
    closed_list = set([])
    peso = 0
```

En la presentación del proyecto ya habíamos dicho que había que tener en cuenta múltiples factores, como podía ser las diferentes horas del día, así como los propios días de la semana. Por lo que hemos decidido sumarle proporcionalmente, según la información que hemos sacado de la aplicación que hemos nombrado anteriormente, cierto tiempo a cada peso a cada arista por la cual transita el camino mínimo.

```python
    closed_list = set([])
    peso = 0
    linea1 = ['Akademmistechko','Zhytomyrska','Sviatoshyn','Nyvky','Beresteiska','Shuliavska','Polit
    linea2 = ['Heroiv_Dnipra','Minska','Obolon','Pochaina','Tarasa_Sevchenka','Kontraktova_Polscha',
    linea3 = ['Syrets','Dorogozhychi','Lukianivska','Zoloti_Vorota','Palats_Sportu','Klovska','Peche
    frecuencias1 = [tiempo_en_segundos(2,0),tiempo_en_segundos(3,30),tiempo_en_segundos(3,30)]
    frecuencias2 = [tiempo_en_segundos(2,0),tiempo_en_segundos(5,0),tiempo_en_segundos(4,15)]
    frecuencias3 = [tiempo_en_segundos(2,15),tiempo_en_segundos(4,10),tiempo_en_segundos(6,0)]
    diario = ['Monday','Tuesday','Wednesday','Thursday','Friday']
    weekend = ['Saturday','Sunday']
```

```
        if 6>hora>0 or hora>24:
            raise ValueError
        if dia in diario:
            if 7<hora<10 or 17<hora<20 :
                if start_node in linea1:
                    peso += frecuencias1[1]
                if start_node in linea2:
                    peso += frecuencias2[1]
                if start_node in linea3:
                    peso += frecuencias3[1]
            if 6<hora<7 or 10<hora<17 or 17< hora<24:
                if start_node in linea1:
                    peso += frecuencias1[0]
                if start_node in linea2:
                    peso += frecuencias2[0]
                if start_node in linea3:
                    peso += frecuencias3[0]
        elif dia in weekend:
            if start_node in linea1:
                peso += frecuencias1[2]
            if start_node in linea2:
                peso += frecuencias2[2]
            if start_node in linea3:
                peso += frecuencias3[2]
        else:
            raise ValueError
```

Metemos en g todas los pesos que hay del nodo principal al resto de nodos, si no lo encuentra es infinito.

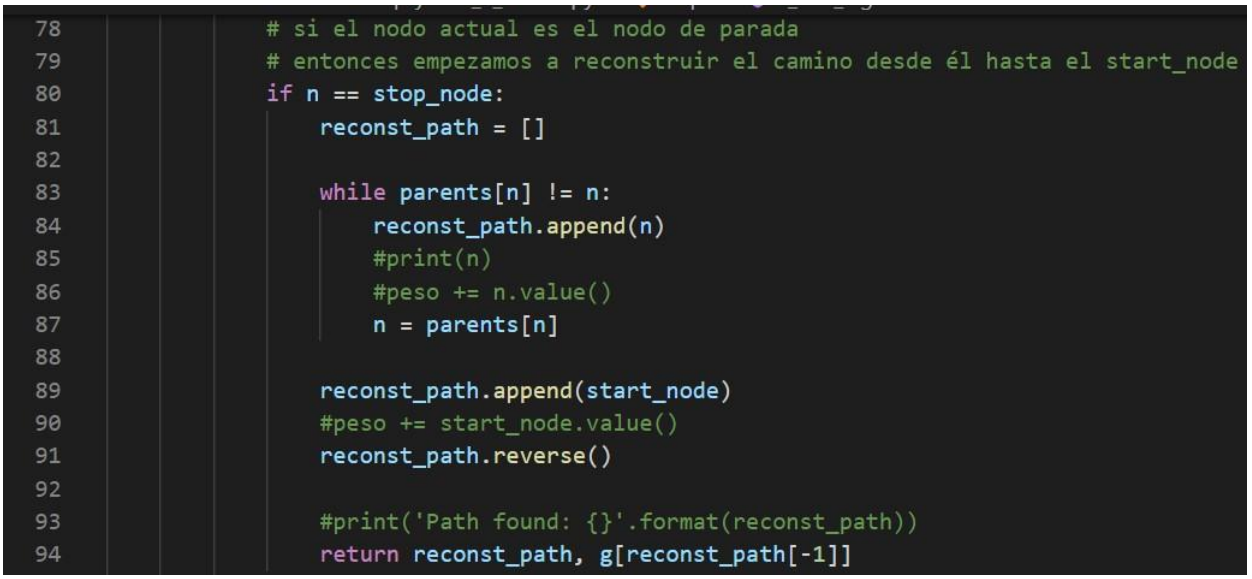
```
# g contiene las distancias actuales desde el nodo_inicial a todos los demás nodos
# el valor por defecto (si no se encuentra en el mapa) es +infinito
g = {}

g[start_node] = 0

# parents  contiene un mapa de adyacencia de todos los nodos
parents = {}
parents[start_node] = start_node
```

Generamos un bucle que intenta buscar en menor valor de f() , como vemos g(v) o g(n) se refiere a el coste real del camino recorrido para llegar a dicho nodo, y data[n][v] , por ejemplo , muestra el valor heurístico del nodo a evaluar desde el actual , es decir es h(n).

A continuación, verificamos si el camino que queríamos tomar existe o no .

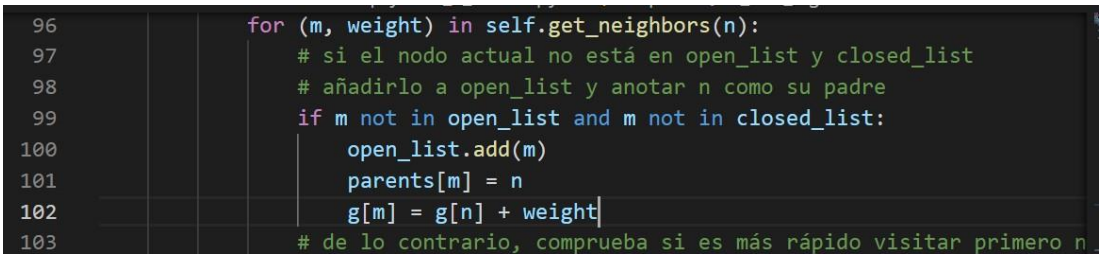
```
while len(open_list) > 0:
    n = None

    # encontrar un nodo con el menor valor de f() - función de evaluación
    for v in open_list:
        if n == None or g[v] + data[v][stop_node] < g[n] + data[n][stop_node]:
            n = v

    if n == None:
        print('Path does not exist!')
        return None
```

La siguiente parte del programa, confirma que no haya acabado el camino mínimo para devolver todo los nodos por los que ha pasado, así la suma de todos los pesos de las aristas por las que pasa el camino mínimo .



```
78              # si el nodo actual es el nodo de parada
79              # entonces empezamos a reconstruir el camino desde él hasta el start_node
80              if n == stop_node:
81                  reconst_path = []
82
83                  while parents[n] != n:
84                      reconst_path.append(n)
85                      #print(n)
86                      #peso += n.value()
87                      n = parents[n]
88
89                  reconst_path.append(start_node)
90                  #peso += start_node.value()
91                  reconst_path.reverse()
92
93                  #print('Path found: {}'.format(reconst_path))
94                  return reconst_path, g[reconst_path[-1]]
```

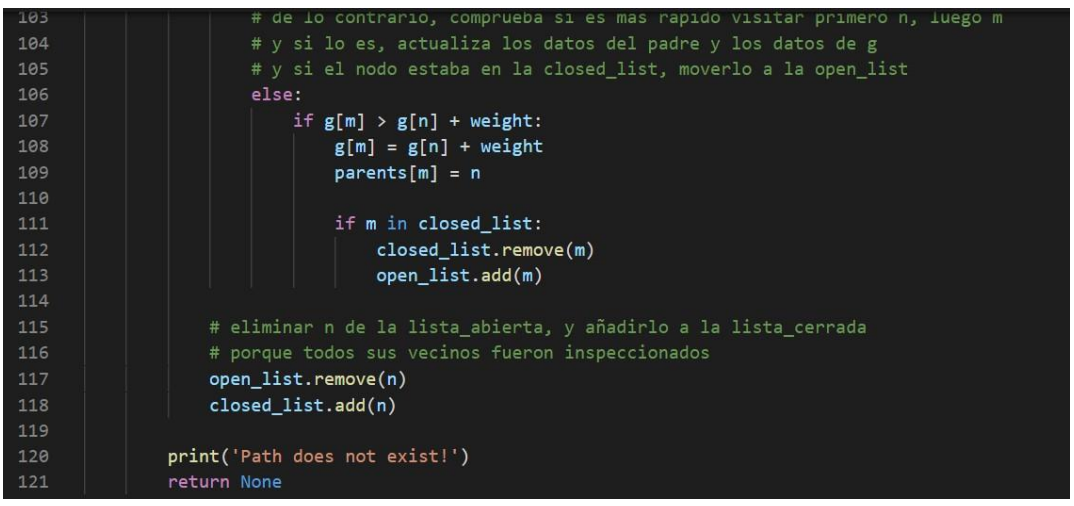
Si no has acabado el camino mínimo, entonces ves si n está en alguna de las 2 listas que hemos creado al principio, si esta no está en ninguna, se le mete en la primera(lista abierta), la lista de nodos que ya han sido visitados , pero cuyos vecino no han sido inspeccionados todos. Y toma a n como su padre



```
96              for (m, weight) in self.get_neighbors(n):
97                  # si el nodo actual no está en open_list y closed_list
98                  # añadirlo a open_list y anotar n como su padre
99                  if m not in open_list and m not in closed_list:
100                     open_list.add(m)
101                     parents[m] = n
102                     g[m] = g[n] + weight
103                 # de lo contrario, comprueba si es más rápido visitar primero n
```

Si por el contrario sí que está en alguna de las listas, comprueba si es más rápido visitar a un nodo y luego a otro , si es así actualiza los datos y si el nodo estaba en la segunda lista(lista cerrada) lo pasa a la primera(lista abierta).

Después de que todos los vecinos hayan sido inspeccionados entonces el nodo se lemete en la segunda lista (lista cerrado) y se repite el proceso otra vez.



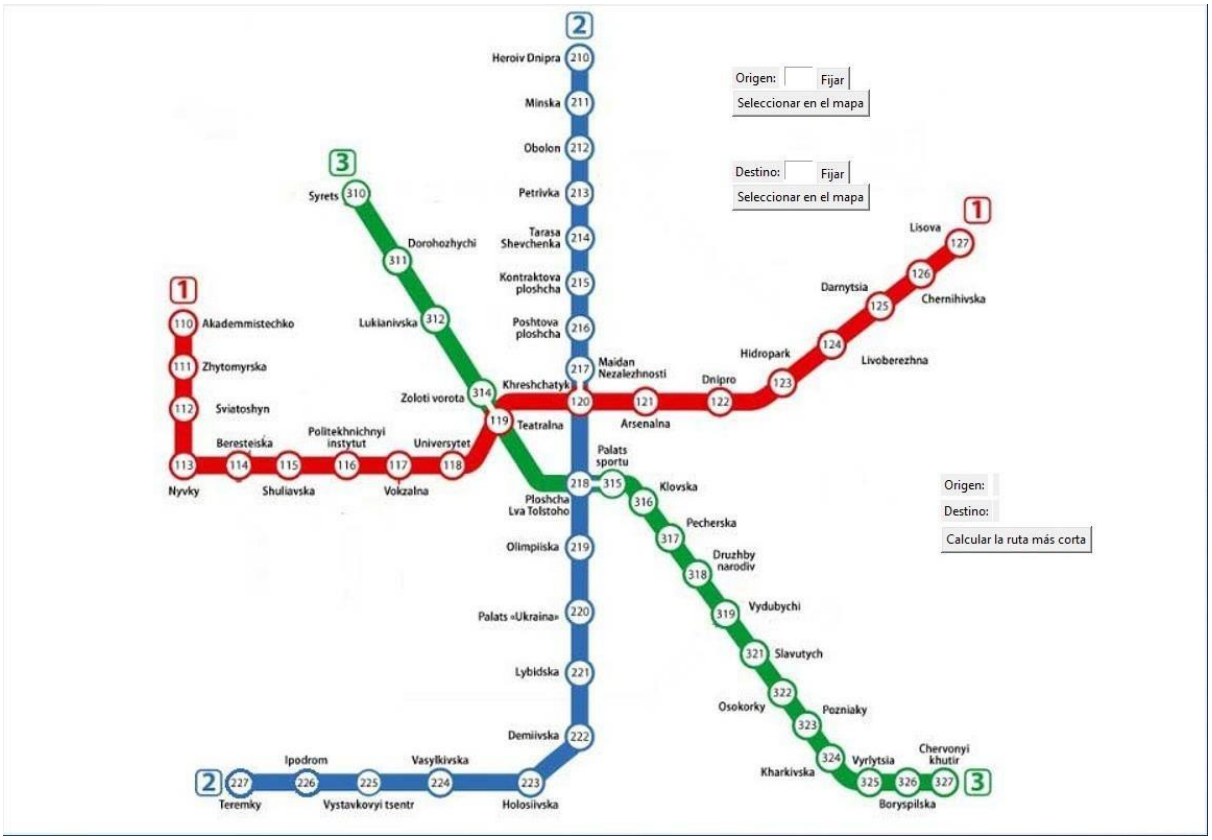
```
103             # de lo contrario, comprueba si es más rápido visitar primero n, luego m
104             # y si lo es, actualiza los datos del padre y los datos de g
105             # y si el nodo estaba en la closed_list, moverlo a la open_list
106             else:
107                 if g[m] > g[n] + weight:
108                     g[m] = g[n] + weight
109                     parents[m] = n
110
111                     if m in closed_list:
112                         closed_list.remove(m)
113                         open_list.add(m)
114
115         # eliminar n de la lista_abierta, y añadirlo a la lista_cerrada
116         # porque todos sus vecinos fueron inspeccionados
117         open_list.remove(n)
118         closed_list.add(n)
119
120     print('Path does not exist!')
121     return None
```

Una vez que mediante el algoritmo A* ya puedo obtener el camino mínimo, así como el tiempo que tardo en hacer el trayecto, lo único que debo hacer es dotar a la aplicación de una interfaz que permita poner los datos y que te muestre el resultado.

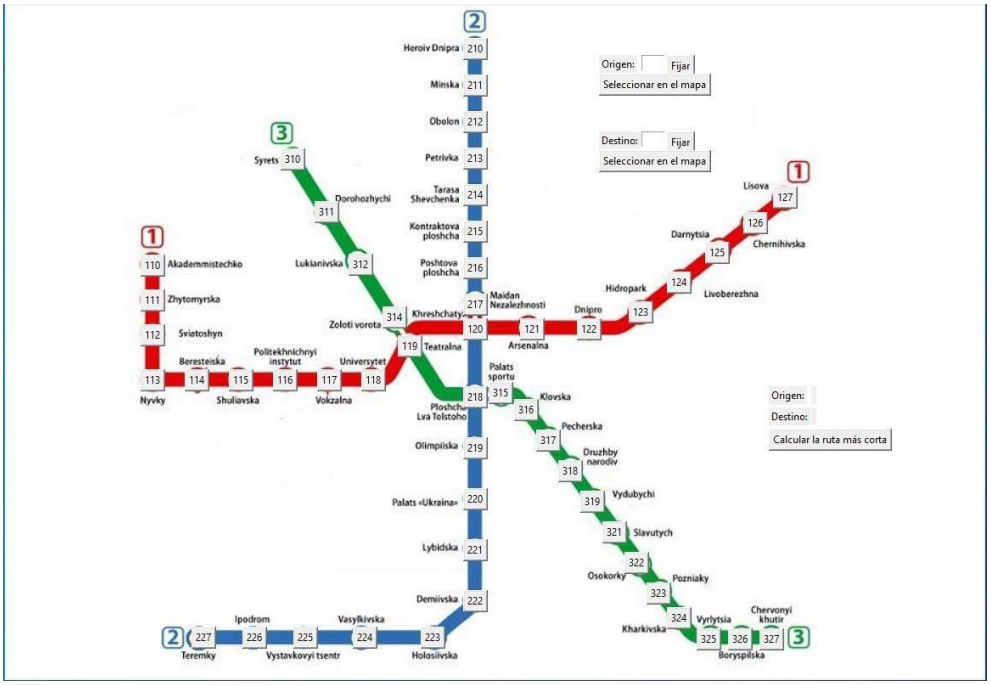
Como ya había comentado antes nuestro grupo había utilizado una librería de Python, se llama Tkinter y nos permitirá hacer todo lo que necesitamos

2.3-Resultados

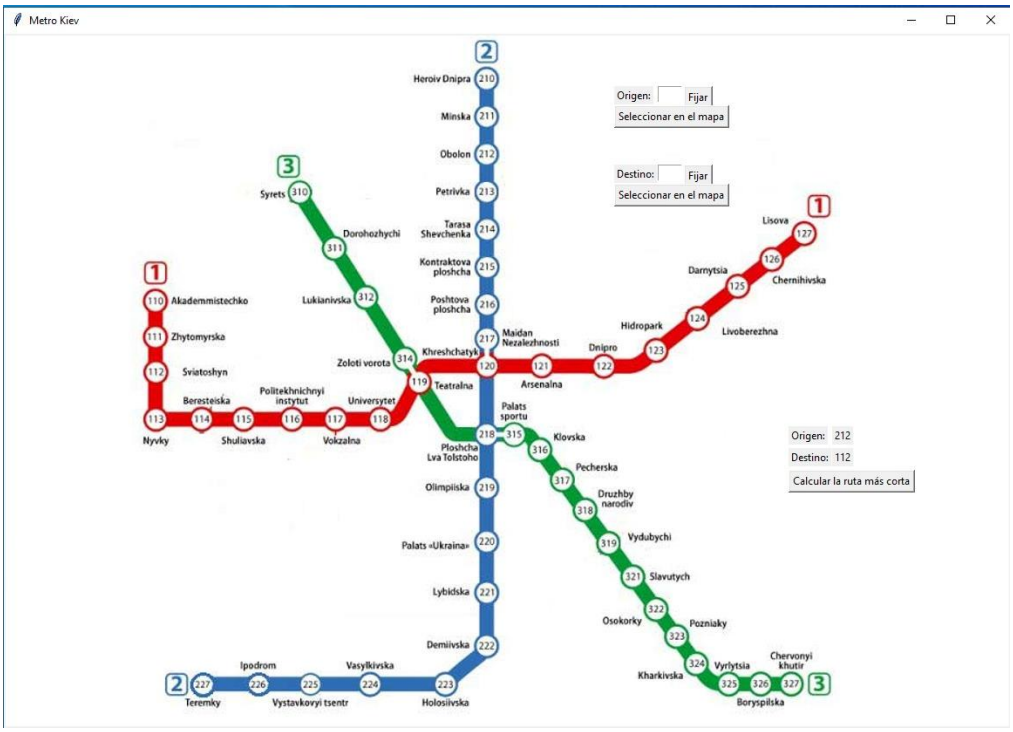
Aquí vemos como nuestra aplicación podemos introducir a mano la estación de origen y de destino, también podemos escoger la estación seleccionando en el mapa, para facilitar las cosas. La aplicación coje automáticamente la hora y día en la que se está ejecutando, y la utiliza en el algoritmo, ya que las frecuencias de los metros pueden cambiar dependiendo de este factor.



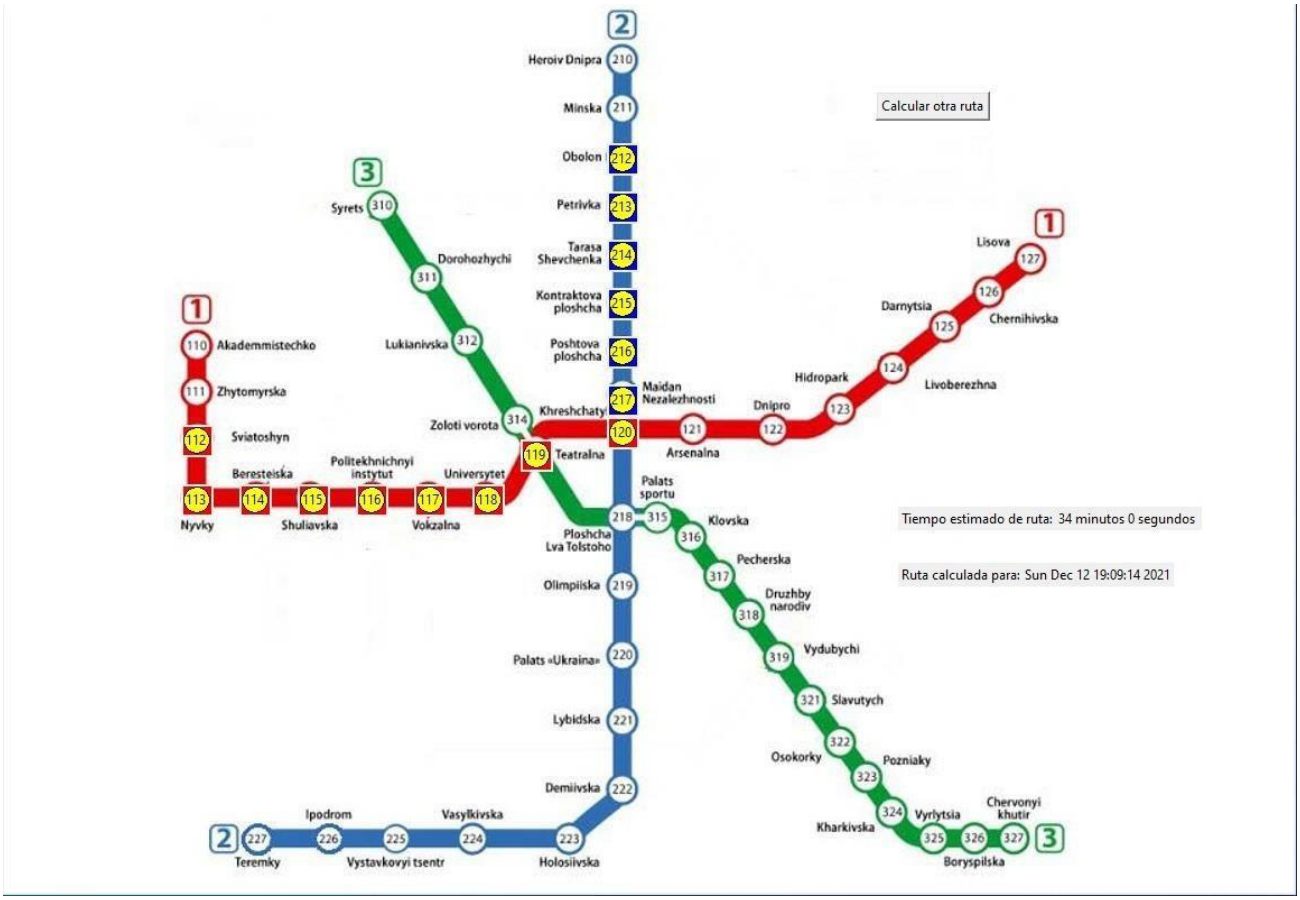
Al pulsar seleccionar en el mapa, se despliegan los botones con los que el usuario puede elegir la parada que desee.



Una vez seleccionadas las paradas origen y destino vemos que se han fijado en la parte de abajo, y el usuario podrá comprobar que está todo bien, y en caso de haberse equivocado podrá cambiar, volviendo a seleccionar las paradas en el mapa, o introduciéndolas a mano



Como vemos, nuestra aplicación devuelve el recorrido que tenemos que seguir estación a estación, puedes ver en el mapa como se colorea los nodos por los que pasa el recorrido mínimo, además, devuelve el tiempo que se tarda en realizarlo y la hora y fecha exacta en la que se ha ejecutado el algoritmo



Hemos intentado que la interfaz gráfica sea lo más sencilla y dinámica posible. Para ejecutar el programa, debes tener una carpeta “data” en el mismo directorio que el archivo .pyw con la imagen del plano del metro y el CSV con los datos de la función heurística. Teniendo esa carpeta, al ejecutar el .pyw se abrirá una ventana con la aplicación.