MEMORIA APLICACIÓN SUBTE

Ciencia de datos e Inteligencia artificial

2.º Curso

Grupo 1

Pablo de Tarso Pedraz García

Manuel Arce Losada

Diego José García Callejas

Pedro Álvaro Martínez Gutiérrez

Mario Martín Muñoz

Héctor Fernández Cano (coordinador)

Índice

INTRODUCCIÓN

La práctica a realizar consiste en la elaboración de una aplicación de usuario con interfaz gráfica de la parte del metro de Buenos Aires (subte) mediante el uso del algoritmo A* visto en clase y que ha sido aportado por el enunciado de la práctica. La finalidad de la aplicación es proveer un recurso que nos permita averiguar el trayecto óptimo entre dos estaciones.

En particular, se utilizará el algoritmo de búsqueda A* (A-estrella) para determinar la ruta más eficiente, considerando, además de las distancias previamente calculadas, una serie de parámetros como los transbordos, el tiempo de viaje o el día y la hora en la que se realizará el trayecto, para así tener en cuenta la frecuencia con la que circulan los metros.

Asimismo, además del código, el usuario dispondrá de una interfaz gráfica muy intuitiva con el código implementado que apoyará que la búsqueda de la ruta más óptica sea lo más cómodo posible, teniendo a la vista todas las estaciones y mostrando en ella la ruta calculada por el algoritmo.

Los objetivos que queremos alcanzar con este proyecto son:

1. El desarrollar un algoritmo de búsqueda eficiente (A*) que calcule la ruta óptima entre dos estaciones, teniendo en cuenta las variables mencionadas.

2.Crear una interfaz gráfica de usuario (GUI) que facilite la interacción con el sistema, permitiendo a los usuarios seleccionar estaciones de inicio y destino, y visualizar la ruta más adecuada.

3. Optimizar la visualización de la ruta en el mapa, mostrando claramente las estaciones, transbordos y el camino elegido.

4.Implementar el sistema en un entorno cooperativo, fomentando el trabajo en equipo y la colaboración en el desarrollo de la aplicación.

Este proyecto tiene un enfoque práctico, permitiendo a los estudiantes aplicar conceptos de algoritmos de búsqueda, programación orientada a objetos, y diseño de interfaces gráficas en el contexto de una aplicación de uso real.

DESCRIPCIÓN DETALLADA DE LA PRÁCTICA

Hemos dividido la practica en tres partes, en primer lugar, hemos realizado una tarea de búsqueda en donde queríamos obtener información relacionada con las horas de inicio y de fin de las líneas, y hemos encontrado que todas las líneas tenían su horario dividido en tres formas, una que es de lunes a viernes, donde el horario empieza a las 5:30 y acababa a las 23:30. Por otro lado, los sábados donde su horario empezaba a las 6:00 y acababa a las 23:30 y por último los domingos donde el horario empieza a las 8:00 y termina a las 22:32.

```
Horarios_inicio_dict = {  # Horarios de inicio de las líneas
    "A": {"L-V": time(5, 30), "S": time(6, 0), "D": time(8, 0)},
    "B": {"L-V": time(5, 30), "S": time(6, 0), "D": time(8, 0)},
    "C": {"L-V": time(5, 30), "S": time(6, 0), "D": time(8, 0)},
    "D": {"L-V": time(5, 30), "S": time(6, 0), "D": time(8, 0)},
    "E": {"L-V": time(5, 30), "S": time(6, 0), "D": time(8, 0)},
```

```
Horario_fin_dict = {# Horarios de cierre de las líneas
    "A": {"L-V": time(23, 30), "S": time(23, 30), "D": time(22, 32)},
    "B": {"L-V": time(23, 30), "S": time(23, 30), "D": time(22, 32)},
    "C": {"L-V": time(23, 30), "S": time(23, 30), "D": time(22, 32)},
    "D": {"L-V": time(23, 30), "S": time(23, 30), "D": time(22, 32)},
    "E": {"L-V": time(23, 30), "S": time(23, 30), "D": time(22, 32)}
}
```

Por otra parte, hemos buscado las estaciones y sus coordenadas y la línea a la que pertenecen las estaciones, las cuales las almacenamos en un fichero csv llamado estaciones_buenos_aires_contildes.

```
Estacion,Longitud,Latitud,Linea
Alberti,-58.401207534233,-34.6098335784398,A
Pasco,-58.3984269918123,-34.6094569170521,A
Congreso,-58.3926688246648,-34.6092256843174,A
Sáenz Peña,-58.3867771940873,-34.6094125865027,A
Lima,-58.3822324010181,-34.609098655619,A
Piedras,-58.3790851530908,-34.608881721215,A
Perú,-58.3742677264304,-34.608559738532,A
Plaza de Mayo,-58.3709648989674,-34.6088103961689,A
Pasteur,-58.399474256679,-34.6046429679193,B
Callao B,-58.3923142350887,-34.6044915426087,B
Uruguay,-58.3872961135408,-34.6040953531057,B
Carlos Pellegrini,-58.3807148471409,-34.6036371051817,B
Florida,-58.3750715182636,-34.6032972855775,B
Leandro N.Alem,-58.3699928501224,-34.6029969466332,B
Constitución,-58.3814434393943,-34.6276194522548,C
San Juan,-58.3799211788651,-34.6219617322081,C
Independencia C,-58.3801736104752,-34.6181255993293,C
Moreno,-58.38044446966,-34.6126177298037,C
Avenida de Mayo,-58.3806107179579,-34.6089833148827,C
Diagonal Norte,-58.3759299800739,-34.6048437399143,C
Lavalle,-58.3781557828244,-34.6017699230114,C
General San Martín,-58.37781910509867,-34.5959574047792,C
Retiro,-58.3740128164816,-34.5911938883113,C
Facultad de Medicina,-58.3972253755734,-34.59975708076,D
Callao D,-58.3932151890727,-34.599639552419,D
Tribunales,-58.3851423580813,-34.6015876151394,D
9 de Julio,-58.3805743428896,-34.6044520296929,D
Catedral,-58.3795360956911,-34.6032843642899,D
Pichincha,-58.3970680746626,-34.6227166621278,E
Entre Ríos,-58.3851485496128,-34.6223394919367,E
San José,-58.3815349411722,-34.617937397972,E
Independencia E,-58.3851395411272,-34.6128491058186,E
Belgrano,-58.3778808506406,-34.6138994512063,E
Bolívar,-58.3736842241685,-34.6092424288863,E
```

Por otra parte, hemos buscado información sobre las frecuencias de los trenes, de la misma forma que los horarios todas las líneas lo tenían dividido en tres formas:

- De lunes a viernes:
    - Línea A: Tiene una frecuencia de 3 minutos.
    - Línea B: Tiene una frecuencia de 3 minutos.
    - Línea C: Tiene una frecuencia de 3 minutos.
    - Línea D: Tiene una frecuencia de 3 minutos.
    - Línea E: Tiene una frecuencia de 4:30 minutos.
- Los sábados:
    - Línea A: Tiene una frecuencia de 7 minutos.
    - Línea B: Tiene una frecuencia de 7 minutos.
    - Línea C: Tiene una frecuencia de 6 minutos.
    - Línea D: Tiene una frecuencia de 7 minutos.
    - Línea E: Tiene una frecuencia de 8:35 minutos.
- Los domingos:
    - Línea A: Tiene una frecuencia de 8 minutos.
    - Línea B: Tiene una frecuencia de 8:50 minutos.
    - Línea C: Tiene una frecuencia de 7:50 minutos.

- Línea D: Tiene una frecuencia de 7 minutos.
- Línea E: Tiene una frecuencia de 8:35 minutos.

```
Frecuencias_dict={   # Frecuencias de las líneas
    "A": {"L-V": 3, "S": 7, "D": 8},
    "B": {"L-V": 3, "S": 7, "D":8.5},
    "C": {"L-V": 3, "S": 6, "D": 7.5},
    "D": {"L-V": 3, "S": 7, "D": 7},
    "E": {"L-V": 4.30, "S": 8.35, "D": 8.35},
}
```

Por otro lado, hemos buscado información sobre las distancias reales, que en este caso hemos querido buscar esta información basándonos en la distancia que recorre el tren al hacer al recorrer las líneas. Posteriormente, hemos buscado la velocidad media de los trenes para dividir esta distancia real.

Para terminar la parte de búsqueda, hemos buscado información sobre los transbordos, como cuantas escaleras tienen, si tiene accesibilidad para los minusválidos, cuanto se tarda en realizar dichos transbordos.

La segunda parte ha sido realizar el código Python de la práctica, esto lo hemos realizado dividiendo el código en varias partes:

1. En primer lugar, hemos realizado un grafo, donde los nodos de este eran las estaciones y cada arista que las unía las líneas. Para ello hemos realizado el código creando las aristas a las cuales les hemos asignado un color que es el mismo que el de las líneas y también les hemos asignado un peso, el cual era la distancia real dividido por la velocidad media de los trenes. Para definir estas aristas lo que hemos hecho ha sido establecer los dos nodos, que como dije antes eran las estaciones. Finalmente, construíamos el grafo mediante la librería de NetworkX.
   Hemos hecho esto para todas las líneas.

```
G = nx.Graph()
# Representacion de la linea A
G.add_edge('Alberti', 'Pasco', color= line_colors['A'], weight=232/VELOCIDAD_MEDIA)
G.add_edge('Pasco', 'Congreso',color= line_colors['A'],weight=528/VELOCIDAD_MEDIA)
G.add_edge('Congreso', 'Sáenz Peña',color= line_colors['A'],weight=567/VELOCIDAD_MEDIA)
G.add_edge('Sáenz Peña', 'Lima',color= line_colors['A'],weight=382/VELOCIDAD_MEDIA)
G.add_edge('Lima', 'Piedras',color= line_colors['A'],weight=365/VELOCIDAD_MEDIA)
G.add_edge('Piedras', 'Perú',color= line_colors['A'],weight=383/VELOCIDAD_MEDIA)
G.add_edge('Perú', 'Plaza de Mayo',color= line_colors['A'],weight=313/VELOCIDAD_MEDIA)
```

Y finalmente la creación del grafo.

```python
def dibujar_grafo(G:nx.Graph):
    np.random.seed(30)
    plt.figure(figsize=(12, 12))
    pos = nx.spring_layout(G)  # Cambiar a un layout más intuitivo

    # Dibujar las aristas con los colores asignados
    edges = G.edges(data=True)

    # Extraer colores de cada arista utilizando el atributo 'color'
    colors = [edge[2]['color'] if 'color' in edge[2] else 'black' for edge in edges]  # Color o negro por defecto

    # Dibujar el grafo
    nx.draw(G, pos, with_labels=True, node_size=500, node_color="lightblue", font_size=8, font_weight="bold", edge_color=colors)
    plt.title("Grafo del Subte de Buenos Aires")
    plt.show()
```

2. Posteriormente, hemos realizado una función que calcula las heurísticas desde una estación inicial a una estación final. Para ello, hemos utilizado un DataFrame, en nuestro caso df_coordenas que almacena las estaciones y sus coordenadas geográficas en grados (latitud y longitud). Estas coordenadas nos permiten calcular la distancia geodésica entre las dos estaciones utilizando la fórmula de Haversine. Esta función lo que hace es convertir estas coordenadas de grados a radianes, posteriormente se calculan las diferencias angulares entra la estación de origen y la de fin, a continuación, calculamos la fórmula de Haversine y por último hacemos una estimación del tiempo dividiendo la distancia que nos sale en el paso anterior por la velocidad media de los trenes.

```python
def calcular_heuristica_entredos ( current:str,target:str)->float:
    RADIO_TIERA=6378100  # Necesitamos el radio de la Tierra para calcular la heurística
    if current == target: # Si el inicio es igual al destino la heurística será 0
        return 0
    # Calculamos los datos para aplicar la fórmula
    longitud_current:float=math.radians(df_coordenas.loc[current]["Longitud"])
    longitud_target:float=math.radians(df_coordenas.loc[target]["Longitud"])
    latitud_current:float=math.radians(df_coordenas.loc[current]["Latitud"])
    latitud_target:float=math.radians(df_coordenas.loc[target]["Latitud"])
    dif_lon:float=longitud_target - longitud_current
    dif_lat:float=latitud_target - latitud_current
    interior:float= math.sqrt(math.sin(dif_lat/2)**2+ math.cos(latitud_current)*math.cos(latitud_target)*math.sin(dif_lon/2)**2)
    #Calculamos la distancia
    distancia:float= 2*RADIO_TIERA*math.asin(interior)
    # Devolvemos la heurística expresadda en minutos
    return distancia//VELOCIDAD_MEDIA
```

3. A continuación, hemos realizado una función que devuelva el día de la semana de la fecha introducida, para ello hemos utilizado una lista llamada DIAS_SEMANA que tiene como claves, en los días de los lunes y viernes: L-V, la de los sábados: S y los domingos: D.

```python
#Funcion que devuelve el dia de la ficha elegidad
def calcular_fecha(fecha:date)->str:
        #Devuelve dia accediendo al indice que devuelve weekday en la lista DIAS_SEMANA
        nombre_dia:str = DIAS_SEMANA[fecha.weekday()]
        return nombre_dia
```

4. Para calcular el tiempo que esperamos en cada estación, hemos implementado una función llamada tiempo_espera_en_estacion. Esta función determina el tiempo de espera en minutos para un tren en una estación específica del metro, utilizando la hora de llegada, el nombre de la estación y la fecha como parámetros. Identifica la línea de metro asociada a la estación a partir de df_coordenas, consulta la frecuencia de paso en Frecuencias_df y los horarios de inicio y fin del servicio en Horarios_inicio_df y Horarios_fin_df. Si la hora de llegada está fuera del horario operativo, devuelve None. Calcula los minutos desde el inicio del servicio y, con la frecuencia correspondiente, estima el tiempo restante para el próximo tren, devolviendo este valor como resultado.

```python
#Funcion que devuelve el tiempo que esperamos en la estacion indicada
def tiempo_espera_en_estacion(hora_llegada:time, nombre_estacion:str,fecha:str)->int:
    linea_metro:str=df_coordenas.loc[nombre_estacion,"Linea"]
    frecuencia:float =Frecuencias_df.loc[fecha,linea_metro]
    horario_inicio:time=Horarios_inicio_df.loc[fecha,linea_metro]
    horario_fin:time=Horarios_fin_df.loc[fecha,linea_metro]
    # Verificamos si la línea está operativa
    if not (horario_inicio <= hora_llegada <= horario_fin):
        return None  # Tren fuera de servicio
    # Calvualmos la hora a la que llegamos y la hora de inicio del servicio en minutos
    minutos_llegada:int = hora_llegada.hour * 60 + hora_llegada.minute
    minutos_inicio:int = horario_inicio.hour * 60 + horario_inicio.minute
    # Calculamos los minutos que llevamos desde que se inicio el servicio
    minutos_desde_inicio:int = minutos_llegada - minutos_inicio
    # Dividimos por la frecueencia de ese dia y ese dato se lo restamos a la frecuencia para saber cuanto tenemos que esperar
    tiempo_espera:int = (frecuencia - (minutos_desde_inicio % frecuencia)) % frecuencia
    #Devuelve el tiempo de espera
    return tiempo_espera
```

5. En otra función, hemos calculado el camino de estaciones a partir del algoritmo A* proporcionado, para las heurísticas que necesita este algoritmo hemos utilizado la función anterior para calcular las heurísticas y con las distancias reales que como dijimos es el weight de las aristas del grafo.

```python
def calcular_camino(inicio:str, fin:str  )->List[str]: # Aplicamos el algoritmo A*
    camino:List[str]= nx.astar_path(G,inicio, fin, heuristic=calcular_heuristica_entredos, weight="weight")
    return camino
```

6. Para saber a qué hora llegamos a la estación destino, hemos creado una función que calcula la hora exacta de llegada a partir de la hora actual y el tiempo estimado de viaje en minutos. La función utiliza datetime.combine para unir la fecha actual con la hora proporcionada, creando un objeto datetime que representa el momento inicial. Luego,

suma el tiempo de viaje en minutos usando time delta, lo que permite calcular la hora exacta de llegada. Finalmente, devuelve solo la hora de llegada como un objeto time.

```python
#Funcion que calcula la hora de llegada dada la hora de inicio y el tiempo del trayecto
def hora_llegada(hora:time,tiempo:int)->datetime:
    hora_actual:datetime = datetime.combine(datetime.today(), hora)
    hora_delta:datetime = hora_actual + timedelta(minutes=tiempo)
    return hora_delta.time()
```

7. Para indicar el número de transbordos que se realizan a lo largo del trayecto hemos hecho una función que accede a la última columna del DataFrame de las coordenadas y hemos accedido a la última columna, la cual contiene la letra de la línea. Lo que hace esta función es recorrer las estaciones del camino comparando las letras de las líneas viendo que si son distintas incrementamos en 1 el contador de transbordos inicializado a 0.

```python
def numero_transbordos(camino: list) -> List[tuple]: # Devolvemos lista de tuplas de transborodos
    #Creamos la listavacia
    num_transbordos=[]
    ultima_columna = df_coordenas.iloc[:, -1]  # Última columna del DataFrame

    for i in range(len(camino) - 1):  # Recorrer hasta el penúltimo elemento
        if ultima_columna[camino[i]] != ultima_columna[camino[i + 1]]:
            num_transbordos.append((camino[i],camino[i+1]))

    return num_transbordos
```

8. Por último, para calcular el tiempo total que lleva realizar el trayecto, hemos implementado una función denominada calcular_tiempo_camino. Esta función suma los tiempos de espera iniciales en la primera estación, los tiempos de viaje entre estaciones y los tiempos de transbordo, si los hay. Utiliza información del grafo de la red de metro, como el tiempo de viaje entre estaciones y el cambio de línea, además de considerar los horarios operativos y la frecuencia de los trenes. Si en algún punto el trayecto no es viable por horarios, la función devuelve none. Finalmente, calcula el tiempo total incluyendo un pequeño ajuste por cada transbordo y lo retorna como resultado.

```python
#Recibe la hora de llegada, el dia y el camino a recorrer
def calcular_tiempo_camino(Hora:time,fecha:str,camino:List[str])->int:
    #Calculas el tiempo que hay que esperar en la estacion
    tiempo_espera_inicio:int=tiempo_espera_en_estacion(Hora,camino[0],fecha)
    #Si el tiempo de espera es None significa que estamos fuera del horario establecido
    if tiempo_espera_inicio is None:
        return None
    #Actualizamos la hora y sumamos el tiempo que va pasando
    tiempo_actual:datetime = datetime.combine(datetime.today(), Hora) + timedelta(minutes=int(tiempo_espera_inicio))
    tiempo_total:int = tiempo_espera_inicio
    #Hacemos la  misma accion durante todo el camino
    for i in range(1,len(camino)):
        nodo_A:str=camino[i-1]
        nodo_B:str=camino[i]
        arista =G.get_edge_data(nodo_A,nodo_B)
        tiempo_segmento=arista.get('weight')
        tiempo_total += tiempo_segmento

        tiempo_actual += timedelta(minutes=int(tiempo_segmento))
        if i < len(camino) - 1:  # No es el último nodo
            nodo_siguiente:str = camino[i + 1]
            color_actual:str = G[nodo_A][nodo_B].get('color', None)
            color_siguiente:str = G[nodo_B][nodo_siguiente].get('color', None)

            if color_actual != color_siguiente:  # Si cambia la línea, hay un transbordo
                # Calculos tiempo de espera para el siguiente tren de la nueva linea de metro
                tiempo_espera_transbordo = tiempo_espera_en_estacion(tiempo_actual.time(), nodo_B, fecha)
                if tiempo_espera_transbordo is None:
                    return None     #Si el tiempo de espera es None significa que estamos fuera del horario establecido

                tiempo_total += tiempo_espera_transbordo
                tiempo_actual += timedelta(minutes=int(tiempo_espera_transbordo))

    #Agisnamos todo a una variable final
    tiempo_final = (tiempo_total)
    return tiempo_final + (len(camino)-2)*0.5
```

La tercera y última parte ha sido desarrollar la interfaz. En esta parte de la práctica, se ha implementado una interfaz gráfica de usuario (GUI) utilizando la librería Tkinter, que permite a los usuarios interactuar con un sistema para planificar su viaje en el metro de Buenos Aires. A continuación, se describe el proceso detallado en cada sección del código.

1. Inicialización de la interfaz principal: La ventana principal de la aplicación se crea usando Tkinter. Se especifica un tamaño fijo para la ventana de 900x850 píxeles, centrada en la pantalla del usuario. Además, se establece un icono personalizado y se configura el título de la ventana. También se cargan imágenes (como el logo del subte y una imagen de tren) que se muestran en la parte superior de la ventana.

```python
# Creamos la ventana principal
app = tk.Tk()
app.title("Subte - Metro Buenos Aires")

# Dimensiones de la ventana
window_width = 900
window_height = 850

# Obtenemos dimensiones de la pantalla
screen_width = app.winfo_screenwidth()
screen_height = app.winfo_screenheight()

# Calculamos  posición centrada
center_x = int(screen_width / 2 - window_width / 2)
center_y = int(screen_height / 2 - window_height / 2)

# Configuramos  la geometría centrada
app.geometry(f"{window_width}x{window_height}+{center_x}+{center_y}")
app.minsize(window_width, window_height)
app.maxsize(window_width, window_height)
```

2. Desplegables para selección de estaciones: Se utilizan dos Combobox para permitir al usuario seleccionar la estación de origen y destino de su viaje. Las estaciones disponibles se extraen de una lista llamada `estaciones`. Los valores predeterminados de estos campos son “Estación de Origen” y “Estación de Destino”.

```python
# Establecemos el desplegable de la estación de origen
entry_from = ttk.Combobox(frame, values=estaciones, width=30, state="readonly")
entry_from.pack(pady=5, anchor="w")
entry_from.set("Estación de Origen")  # Texto antes de elegir la estacion deseada

# Establecemos el desplegable de la estación de destino
entry_to = ttk.Combobox(frame, values=estaciones, width=30, state="readonly")
entry_to.pack(pady=5, anchor="w")
entry_to.set("Estación de Destino")   # Texto antes de elegir la estacion deseada
```

```python
#Nombre de las estaciones
estaciones= ["9 de Julio", "Alberti", "Avenida de Mayo", "Belgrano", "Bolívar", "Callao B", "Callao D", "Carlos Pellegrini", "Catedral", "Congreso",
"Constitución","Diagonal Norte", "Entre Ríos", "Facultad de Medicina", "Florida", "General San Martín", "Independencia C", "Independencia E",
"Lavalle", "Leandro N.Alem", "Lima", "Moreno","Pasco", "Pasteur", "Perú", "Pichincha", "Piedras", "Plaza de Mayo", "Retiro", "Sáenz Peña","San José",
"San Juan", "Tribunales", "Uruguay"]
```

3. Selección de fecha y hora: Se incluye un botón para seleccionar la fecha mediante un calendario emergente. La fecha seleccionada se muestra en un campo de texto readonly

para evitar la edición manual y así evitar errores. Además, se implementan dos Combobox para seleccionar la hora y el minuto de salida, con valores predeterminados que corresponden a la hora y minuto actuales.

```
# Configurar fecha por defecto
today = datetime.today()  # Obtener la fecha de hoy
entry_day = tk.StringVar()
entry_day.set(today.strftime("%Y-%m-%d"))  # Establecer la fecha actual

# Crear campo de entrada para la fecha
entry_day_widget = ttk.Entry(frame_day, width=20, textvariable=entry_day, state="readonly")
entry_day_widget.pack(side=tk.LEFT, padx=5)

# Botón para abrir el calendario
calendar_btn = ttk.Button(frame_day, text=" 🗓 ", command=open_calendar)  # Llamamos a la función calendario
calendar_btn.pack(side=tk.LEFT)


# Creamos la cuarta fila donde se podra seleccionar las horas
hour_var = tk.StringVar()
minute_var = tk.StringVar()

#Establecemos las posibles horas y minutos
hours = [f"{h:02}" for h in range(24)]
minutes = [f"{m:02}" for m in range(0, 60, 1)]

#ELegimos la hora actual y lo ponemos como valor predetermiando
current_hour = today.strftime("%H")
current_minute = int(today.strftime("%M"))
```

4. Funcionalidad del calendario emergente: La función open_calendar() permite abrir una ventana emergente con un calendario interactivo. Los usuarios pueden seleccionar una fecha y esta se inserta automáticamente en un campo de texto readonly. El calendario se cierra después de la selección de la fecha y muestra el nuevo valor en el campo de texto.

```
# Función que nos permite  el calendario y seleccionar una fecha
def open_calendar():
    global ventana

    def select_date():
        selected_date = cal.selection_get()
        entry_day_widget.config(state="normal")  #Cambiamos a modo normal para poder introducir la fecha
        entry_day_widget.delete(0, tk.END)
        entry_day_widget.insert(0, selected_date.strftime("%Y-%m-%d"))
        entry_day_widget.config(state="readonly") #Volvemos a modo leer para que no se pueda cambiar la fecha escribiendo directameatne
        ventana.destroy()
```

5. Popup emergente: Mediante la función popup_route_and_transfer() creamos un popup mediante Toplevel (ventana hija de la ventana principal) el cual contendrá información sobre el trayecto y los transbordos que tendrá que realizar el usuario en caso de que sea posible.

```python
def popup_route_and_transfer(text):
    #Creamos el popu
    popup = tk.Toplevel(app)
    popup.title("Información del trayecto") #Añadimos titulo
    popup.geometry("500x400")
    popup.transient(app)
    popup.grab_set()  #Asi obligamos a que el usuario cierre el popup antes de seguir
    label = tk.Label(
        popup,
        text="Detalles del trayecto:",
        font=("Arial", 12),
        justify="center"
    )
    label.pack(pady=20)
    #Cuadro donde escribiremos tanto la ruta como los tranbordos
    text_box = tk.Text(popup, wrap="word", font=("Arial", 12), height=10, width=60)
    text_box.pack(padx=20, pady=20)
    text_box.insert("1.0", text)
    text_box.config(state="disabled")  # Solo puede leerse y no modficarse
    # Para cerrar el boton
    close_button = ttk.Button(popup, text="Cerrar", command=popup.destroy)
    close_button.pack(pady=10)
```

6. Función de búsqueda de ruta: Cuando el usuario hace clic en el botón "Buscar ruta", se ejecuta la función when_button_click(), que realiza los siguientes pasos:
    - Recoge los datos de las estaciones de origen y destino, fecha y hora de viaje.
    - Calcula el día de la semana a partir de la fecha seleccionada.
    - Llama a funciones adicionales para calcular el camino entre las estaciones, el tiempo de viaje, la hora de llegada y el número de transbordos necesarios.
    - Si el camino es una lista de un elemento, quiere decir que ya estamos en la estación deseada.
    - Si el horario de viaje está fuera del servicio, muestra un mensaje informando al usuario de que debe esperar a que inicie la jornada de trenes.
    - En caso de que el trayecto sea válido, muestra el resultado con información detallada sobre el viaje: estación de origen, destino, fecha, hora de salida, duración del trayecto, hora de llegada y número de transbordos. Del mismo modo, genera un pop con la información del trayecto a recorrer y los transbordos que hay que hacer

```python
#Las operaciones que realizaran cuando hagos click en el boton
def when_button_click():
    from_st = entry_from.get() #Estacion de origen
    to_st = entry_to.get() #Estacion de fin
    day = entry_day.get() #Dia de viaje
    hour = hour_var.get() #Hora de viaje
    minute = minute_var.get() #Minuto d e viaje
    date_day = day.split("-") #Separamos la fecha para tenerla en formato fehca
    final_date = date(int(date_day[0]), int(date_day[1]), int(date_day[2])) #Pasamos a foramto fecha
    final_hour = time(int(hour), int(minute.zfill(2))) #Pasamos a formato time


    try:
        #Calculammos el dia de la semana
        week_day = calcular_fecha(final_date)
        #Avergiamos el camino
        way_to_st: List = calcular_camino(from_st, to_st)

        if len(way_to_st) == 1: #Si la longitud es una tenemos la misma estacion
            result_label.config(text='Ambas estaciones introducidas son la misma')
        else:
            time_to_st= calcular_tiempo_camino(final_hour, week_day, way_to_st)
            transfer = None

            if time_to_st is None: #El horario de viaje esta fuera del servicio
                week_day = None
                way_to_st = None
                time_to_st = None
                result_label.config(text="No hay trenes a estas horas,\ntiene que esperar a que comience la jornada de trenes")
            else:
                time_to_st = round(time_to_st)
                hour_at_st = hora_llegada(final_hour, time_to_st)
                transfer = numero_transbordos(way_to_st)
            #Mostramos el resultado

            result_text = (
                f"🚆 TRAYECTO 🚆\n\n📍 {from_st} -> {to_st}\n\n"
                f"📅 {final_date} - {final_hour.strftime('%H:%M')}\n\n"
                f"⏱️ Duración del trayecto: {time_to_st} min\n\n🕐 Hora de llegada -> {hour_at_st.strftime('%H:%M')}\n\n"
                f"🔄 Transbordos: {len(transfer)}\n\n🚉 Paradas: {len(way_to_st)}\n\n"
                # f"🗺️ Ruta:\n\n{' - '.join(map(str, Camino))}"
            )

            result_label.config(text=result_text)
            popup_route_and_transfer(f"Trayecto:{way_to_st}\n\nTransbordos:{transfer}")
        #Coloreamos los botones de las estaciones
        light_floating_button(app, station_positions, way_to_st, new_color="yellow")

    except Exception as e:
        print(e)
```

7. Resaltar estaciones en el camino: La función light_floating_button( ) resalta las estaciones que forman parte del camino calculado, cambiando el color de los botones flotantes correspondientes. Este cambio visual ayuda al usuario a identificar fácilmente las estaciones que debe tomar en su viaje.

```python
#Funcion que nos permite coloear los botones  para indicar el camino
def light_floating_button(self, station_positions, Camino, new_color="yellow"):
    for station, data in station_positions.items():
        x, y = data["coords"]  # Obtener las coordenadas de cada estación
        fg_color = new_color if station in Camino else "white"  # Si está en el camino, usamos el color amarillo
        bg_color = data["bg_color"] #Con esto mantenemos el color de fondo de la estacion

        customtkinter.CTkButton(
            self,
            width=14,
            height=14,
            corner_radius=15,
            fg_color=fg_color,
            bg_color=bg_color,
            hover_color="yellow",
            text="",
            command=lambda station=station: fill_field(station)
        ).place(x=x, y=y)
```

8. Botones flotantes para las estaciones: Cada estación en el mapa tiene un botón flotante que, al hacer clic, rellena automáticamente el campo de origen o destino en función de cuál esté vacío. Estos botones son creados dinámicamente usando las coordenadas y los colores de fondo de las estaciones. Los botones flotantes son visibles y activos en la interfaz gráfica.

```python
#Funcion que crea un botón flotante con texto, posición x e y, y color de fondo.
def create_floating_button(self, text, x, y, bg_color="white"):
    customtkinter.CTkButton(
        self,
        width=14,  # Tamaño pequeño
        height=14,
        corner_radius=15,  # Hacerlo circular
        fg_color="white",  # Color del botón
        bg_color=bg_color,  # Usamos el colon de fondo especificado
        hover_color="yellow",  # Color al pasar el ratón
        text="",
        command=lambda: fill_field(text)
    ).place(x=x, y=y)
```

9. Fondo de la aplicación: Se establece una imagen de fondo en la ventana principal, con una imagen que se ajusta al tamaño adecuado para cubrir una parte de la interfaz. La imagen se coloca en la esquina superior derecha de la ventana.

```python
bg_path = os.path.join(BASE_DIR, 'assets', 'images', 'fondo.jpg')
bg = Image.open(bg_path)
bg = bg.resize((450,775))
background = ImageTk.PhotoImage(bg)

# Etiqueta para mostrar la imagen
bg_label = tk.Label(app, image=background)
bg_label.place(relx=1.0, y=1, anchor="ne")  # Posiciona en la esquina superior derecha
```

10. Funcionamiento de la aplicacion: El flujo de la aplicación comienza con la inicialización de la ventana principal, donde el usuario puede seleccionar las estaciones de origen y destino, la fecha y hora de salida. Al hacer clic en el botón "Buscar ruta", el sistema calcula la ruta óptima y muestra los detalles del trayecto. Si es necesario, resalta las estaciones correspondientes en el mapa y permite al usuario ver la información clave sobre su viaje, como la duración, la hora de llegada y el número de transbordos, como ya hemos mencionado anteriormente.

Una parte adicional, ha sido crear el ejecutable, el cual es para una aplicación de escritorio. Para esto hemos utilizado la librería de Pyinstaller. Para realizar esta operación es necesario acceder a la ruta donde se encuentra nuestra interfaz gráfica en nuestro caso se denomina app.py. Posteriormente escribir la siguiente línea:

pyinstaller--onefile--noconsole--icon=assets/icons/logo_subte.ico--add-data "data/estaciones_buenos_aires_contildes.csv;data" --add-data "assets;assets" app.py. La cual

empaqueta el archivo principal app.py, junto con todos los recursos y dependencias necesarias, en un único archivo ejecutable independiente. Este ejecutable incluye tanto los datos adicionales requeridos (como el archivo CSV y los recursos gráficos en la carpeta assets), como una configuración personalizada, como el icono de la aplicación. Además, elimina la necesidad de mostrar una consola durante la ejecución, lo que resulta ideal para aplicaciones de escritorio con interfaz gráfica. Dicho ejecutable se puede encontrar dentro de la carpeta de dist, la cual esta dentro de la carpeta SUBTE.

CONCLUSIONES TÉCNICAS Y GESTIONES DE GRUPO

A la hora de la elaboración del proyecto encontramos diferentes problemas que tuvimos que de manera eficiente y eficaz. Por un lado, empezamos teniendo dificultades para encontrar información técnica de las estaciones y de los trenes, ya que la información estaba repartida en numerosas páginas web. Y, por otro lado, centrándonos en el código el problema más notable fue el de elaborar una función que permitiese calcular de manera más exacta el horario de llegada, teniendo en cuenta no solo la frecuencia de los trenes iniciales, sino de aquellas líneas durante el trayecto, en el caso de que hubiese transbordos. Otra gran dificultad fue la elaboración de la interfaz gráfica debido a que ninguno de los componentes del grupo se había enfrentado a este tipo de retos.

Todos los problemas fueron resueltos gracias a la participación de todos los miembros del grupo, los cuales se han apoyado entre ellos en reuniones, a las cuales fueron todos los miembros, y de manera online, donde nuevamente ningún integrante fallo a su compromiso. Otro aspecto a resaltar del trabajo en grupo es el gran ambiente de armonía e implicación que se ha visto durante el desarrollo de dicha de práctica, donde ningún integrante ha mostrado desinterés y han aportado todas las grandes que han tenido.

Por último, hay que mencionar que durante la elaboración del trabajo nos hemos dado cuenta de que durante las reuniones presenciales es donde más se aprende y se disfruta trabajando, debido a que son momentos donde podemos ver las verdaderas cualidades de los demás y en momentos de frustración, existen apoyos que nos permiten seguir adelante y resolver los fallos y problemas

REFERENCIAS

- Modulos utilizados:
- Networkx
- Matplotlib
- Numpy
- Datetime
- Pandas
- Typing
- Math
- Tkinker

- Emova: https://emova.com.ar/
- Subte Buenos Aires: https://subtebuenosaires.com.ar/
- Wikipedia: https://es.wikipedia.org
- Buenos Aires Ciudad: https://buenosaires.gob.ar/inicio/
- Google Maps: https://www.google.es/maps/preview
- Moovit: https://moovitapp.com
- Data Sets Argentina: https://datos.gob.ar/dataset?tags=Buenos+Aires
- Calcular distancias: https://www.calcmaps.com/es/map-distance/#google_vignette

EXPLICACIÓN DE COMO HACER EJECUTABLE

Para crear la versión de usuario final del programa utilizaremos la herramienta Pyinstaller. Esta una herramienta de Python que permite convertir scripts de Python (.py) en ejecutables autónomos para diferentes sistemas operativos (Windows, macOS y Linux). Esto es especialmente útil para distribuir aplicaciones Python sin requerir que el usuario final tenga Python instalado en su máquina.

Las dos grandes ventajas de esta herramienta es que nos permite generar versiones completamente portables del programa, de manera que el usuario no tenga la necesidad de tener instalado Python en su equipo, así como librerías que puedan emplear los scripts. Además, el ejecutable es completamente multiplataforma, funciona en Windows, macOS y Linux. Por estas dos razones principales y otras muchas más, Pyinstaller se convierte en una excelente herramienta para generar el ejecutable de aplicaciones de escritorio.

¿CÓMO HEMOS APLICADO PYINSTALLER?

1. Instalar PyInstaller: Asegúrate de tener PyInstaller instalado. En una terminal o consola, ejecuta: pip install pyinstaller

2. Navegar hasta la carpeta que contiene el .py que ejecuta nuestra aplicación: cd "ruta"

3. Antes de crear el ejecutable debes asegurarte de que las rutas relativas de tu codigo esten bien definidas y sean compatibles con la nueva estructura de archivos que implementa Pyinstaller.

4. Emplea el comando pyinstaller junto con los parámetros que creas necesarios para tu ejecutable, en nuestro caso: pyinstaller --onefile --noconsole --icon=assets/icons/logo_subte.ico --add-data"data/estaciones_buenos_aires_contildes.csv;data"--add-data "assets;assets" app.py

    Explicacion del comando resumida: Convierte app.py en un único ejecutable (--onefile). Oculta la consola al ejecutar el programa (--noconsole). Establece un ícono personalizado (--icon). Incluye recursos adicionales como el CSV y la carpeta assets (--add-data). Resultado: Obtendrás un ejecutable (app.exe) en la carpeta dist.

Este ejecutable incluirá todos los archivos y recursos especificados y no necesitará dependencias externas para ejecutarse.

Enlace de descarga a la versión con ejecutable: https://drive.google.com/file/d/1gZkRHmCKjdcG2_1u_VYEZDro7oBG4c7f/view?usp=drive_link

En caso de que esto no funcionase, podrías simplemente entrar en el archivo de app.py y se ejecuta y se vería el resultado del ejecutable, pero sin tener el ejecutable.