Algoritmo de Dijkstra

José Eduardo López Corella - C24343

Universidad de Costa Rica

2 de Junio de 2024

Introducción

El algoritmo de Dijkstra es un algoritmo que nos ayuda a encontrar la ruta más corta entre en un grafo, lo que lo convierte en una herramienta fundamental en la teoría de grafos y en diversas aplicaciones prácticas. Este algoritmo, llamado así por su desarrollador Edsger Dijkstra es especialmente útil en diversos campos en donde la optimización de rutas es crucial. En esta presentación, exploraremos el funcionamiento del algoritmo de Dijkstra, su implementación en Python y algunos ejemplos de su aplicación en la vida real.

Objetivo General

Desarrollar la búsqueda de rutas optimas entre dos nodos de un grafo en escenarios complejos mediante la implementación del algoritmo de Dijkstra en Python.

Objetivos especificos

- Explicar el funcionamiento del algoritmo de Dijkstra.
- Implementar el algoritmo de Dijkstra para resolver el problema de la ruta más corta.
- Ejemplificar los usos que puede llegar a tener este algoritmo en la vida real.

Funcionamiento del Algoritmo de Dijkstra

- **Inicialización**: Se crean dos diccionarios, uno para registrar las distancias provisionales desde el vértice de inicio a todos los demás vértices, y otro para registrar los predecesores de cada vértice en la ruta más corta. Todas las distancias se inicializan a infinito, excepto la distancia al vértice de inicio, que se establece en 0. Los predecesores se inicializan en None.
- **Selección del vértice actual**: En cada iteración, se selecciona el vértice no explorado con la menor distancia provisional. Esta selección se realiza mediante la función `minimo`, que encuentra la clave (vértice) con el menor valor en el diccionario de distancias provisionales.

Funcionamiento del Algoritmo de Dijkstra

- **Actualización de distancias**: Para cada vecino del vértice actual, se calcula una distancia tentativa sumando la distancia provisional del vértice actual y el peso de la arista que lo conecta con el vecino. Si la distancia tentativa es menor que la distancia provisional almacenada para ese vecino, se actualiza la distancia provisional y se registra el vértice actual como predecesor del vecino.
- **Eliminación del vértice actual**: Después de actualizar las distancias a todos los vecinos, el vértice actual se elimina del conjunto de vértices no explorados.
- **Finalización**: El algoritmo termina cuando se han explorado todos los vértices o cuando se ha alcanzado el vértice destino. Para reconstruir la ruta más corta, se sigue el rastro de los predecesores desde el vértice destino hasta el vértice de inicio.

Aplicaciones del Algoritmo de Dijkstra

- **Mapas de navegación**: Para encontrar la ruta más rapida para llegar de un punto A a un punto B.
- **Aviación**: Podría implementarse para encontrar las escalas optimas que se deberían hacer con el fin de maximizar ganancias.
- **Logística y Cadena de Suministro**: Se utiliza para optimizar rutas de entrega y recolección en sistemas de transporte en empresas de reparto.
- **Telecomunicaciones y Sistemas de energía**: Ayuda en el diseño eficiente de redes de telecomunicaciones o energía con el fin de minimizar el costo de instalación y operación.
- **Infraestructura Urbana**: Ayuda en la planificación y desarrollo de infraestructura urbana eficiente, como la optimización en la creación de nuevas rutas.

Implementación en Python

Paquetes necesarios para la implementación de dicho algoritmo en la clase Dijkstra que se desarrollo.

```
1 import networkx as nx # para la creacion de grafos
2 import matplotlib.pyplot as plt # para plotear
3 import random # para crear eventualmente un for que genere vertices
  ↪  y aristas aleatorias
```

Pseudocódigo del algoritmo

A continuación, se presenta un pseudocódigo basado en el código desarrollado en Python.

```
Clase DijkstraAlgorithm:
    Método __init__(vertices, rutas, inicio, destino):
        Inicializar atributos con parámetros
        Llamar al método dijkstra() para calcular la distancia
        ↪ mínima y la ruta
        Llamar al método mostrar_resultados()

    Método minimo(dict):
        Encontrar el vértice con la menor distancia en el
        ↪ diccionario

```

Pseudocódigo del algoritmo

```
1     Método dijkstra():
2         Inicializar distancia de todos los vértices a infinito
3         Establecer distancia del vértice de inicio a 0
4         Mientras haya vértices no explorados:
5             Seleccionar vértice con la menor distancia
6             Si el vértice seleccionado es el destino, romper el
              ↪  bucle
7             Para cada vecino del vértice seleccionado:
8                 Calcular distancia tentativa
9                 Si la distancia tentativa es menor, actualizar
                  ↪  distancia y predecesor
10            Marcar vértice como explorado
11        Reconstruir y devolver la ruta y la distancia mínima
12
13    Método mostrar_resultados():
14        Imprimir distancia mínima y la ruta
15        Crear y dibujar el grafo
16        Resaltar la ruta más corta en el grafo
17
18    Método __str__():
19        Devolver una representación legible del objeto
```

Resultados y visualización caso básico

Acá se presenta un ejemplo de uso, en donde se inicializan los vertices, las rutas, punto de inicio y destino pues estos son los parámetros que requiere la clase DijkstraAlgorithm para funcionar.

```
vertices = ['A', 'B', 'C', 'D', 'E']
rutas = {
    ('A', 'B'): 7,
    ('A', 'C'): 3,
    ('B', 'C'): 1,
    ('B', 'D'): 3,
    ('C', 'D'): 5,
    ('C', 'E'): 6,
    ('E', 'D'): 1,
}
inicio = 'A'
destino = 'D'

dijkstra_basico = DijkstraAlgorithm(vertices, rutas, inicio,
↪  destino)
```

```
La distancia mínima entre A y D es 7
La ruta es A -> C -> B -> D
```

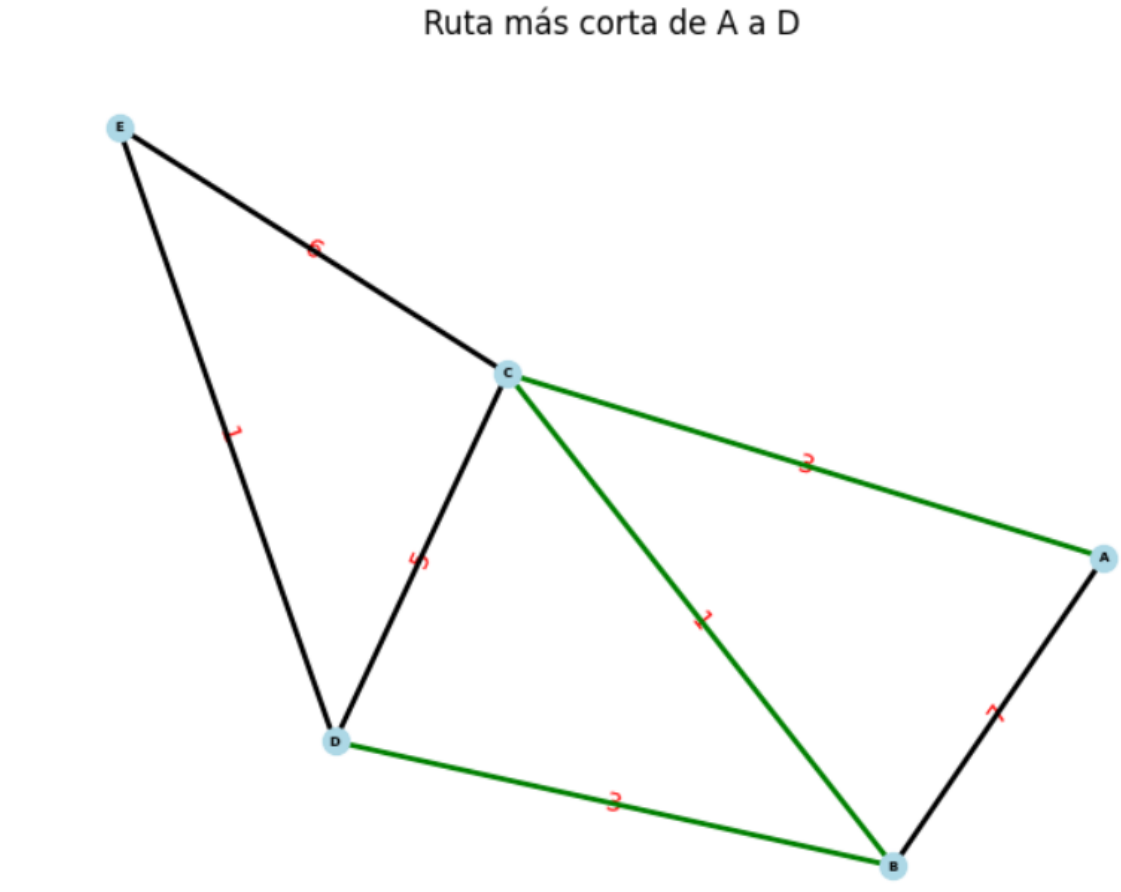


Figura: Gráfico del output obtenido del Ejemplo de uso

Resultados y visualización caso random con 40 vertices

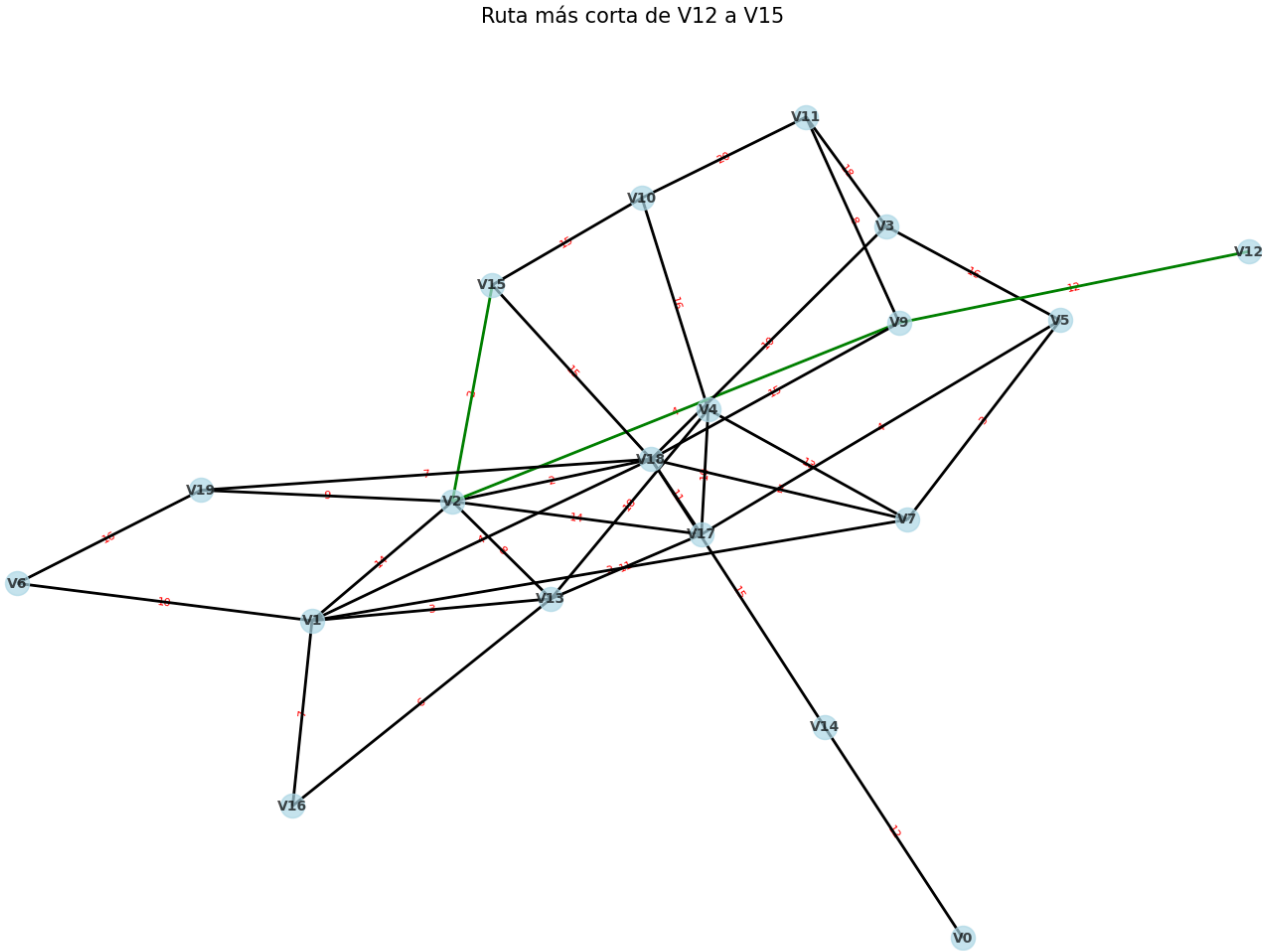
```
# Generar un conjunto de 40 vértices y 100 rutas aleatorias (aristas)
vertices_random = [f'A{i}' for i in range(40)]
rutas_random = {}
for _ in range(100):  # Generar 100 rutas aleatorias
    origen_random, trayecto_random = random.sample(vertices_random, 2)
    tiempo_random = random.randint(1, 20)  # Longitud aleatoria entre 1 y 20
    rutas_random[(origen_random, trayecto_random)] = tiempo_random

inicio_random = random.choice(vertices_random)
destino_random = random.choice(vertices_random)

dijsktrar40 = DijkstraAlgorithm(vertices_random, rutas_random, inicio_random, destino_random)
```

```
La distancia mínima entre A14 y A7 es 24
La ruta es A14 -> A19 -> A24 -> A11 -> A1 -> A7
```

Figura: Gráfico del output obtenido con 40 vertices

Juegos de hallar el camino más corto

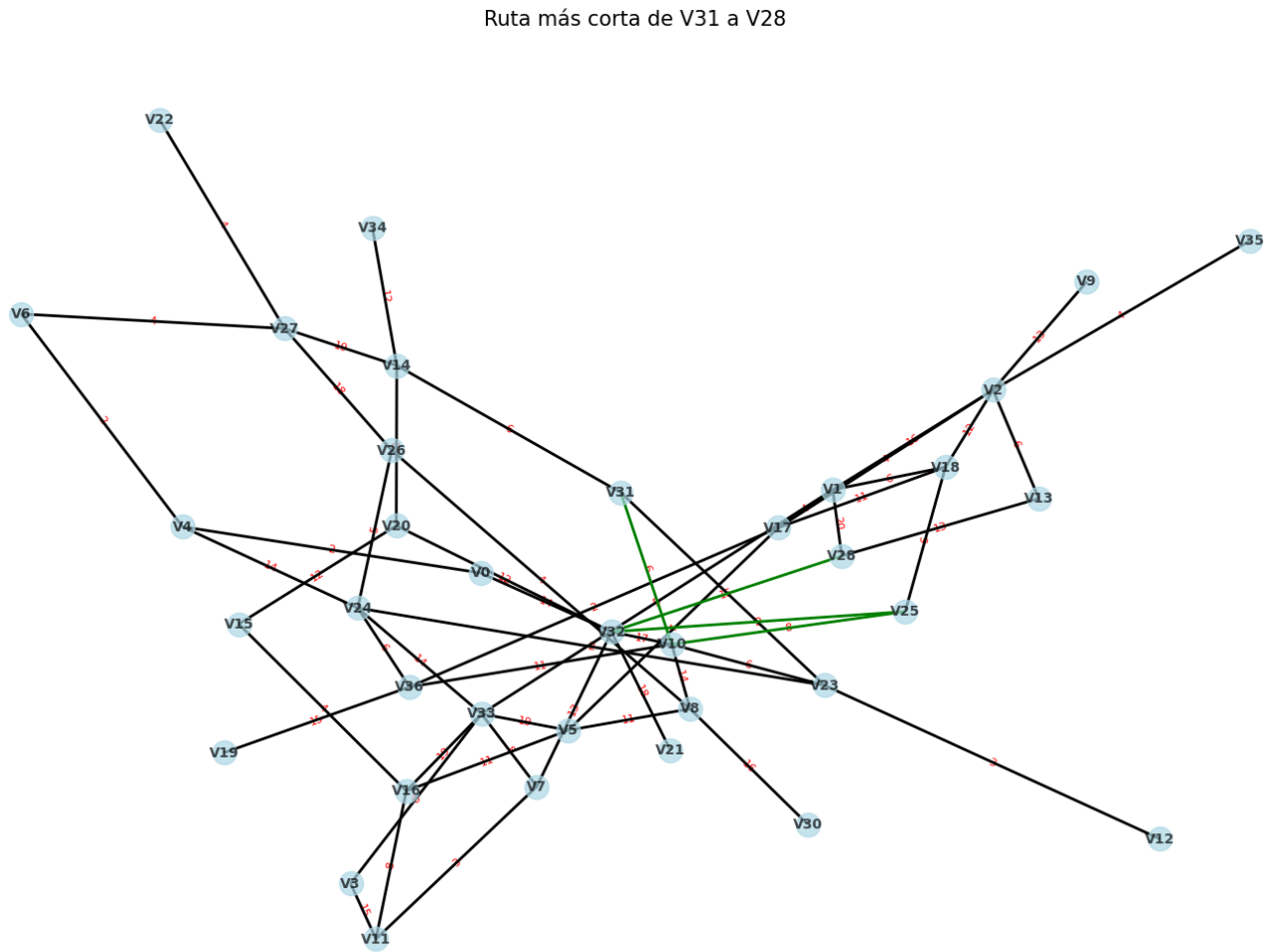
Solución Juego 1: 20 vértices



La distancia mínima entre V12 y V15 es 18

$$V12 \rightarrow V9 \rightarrow V2 \rightarrow V15$$

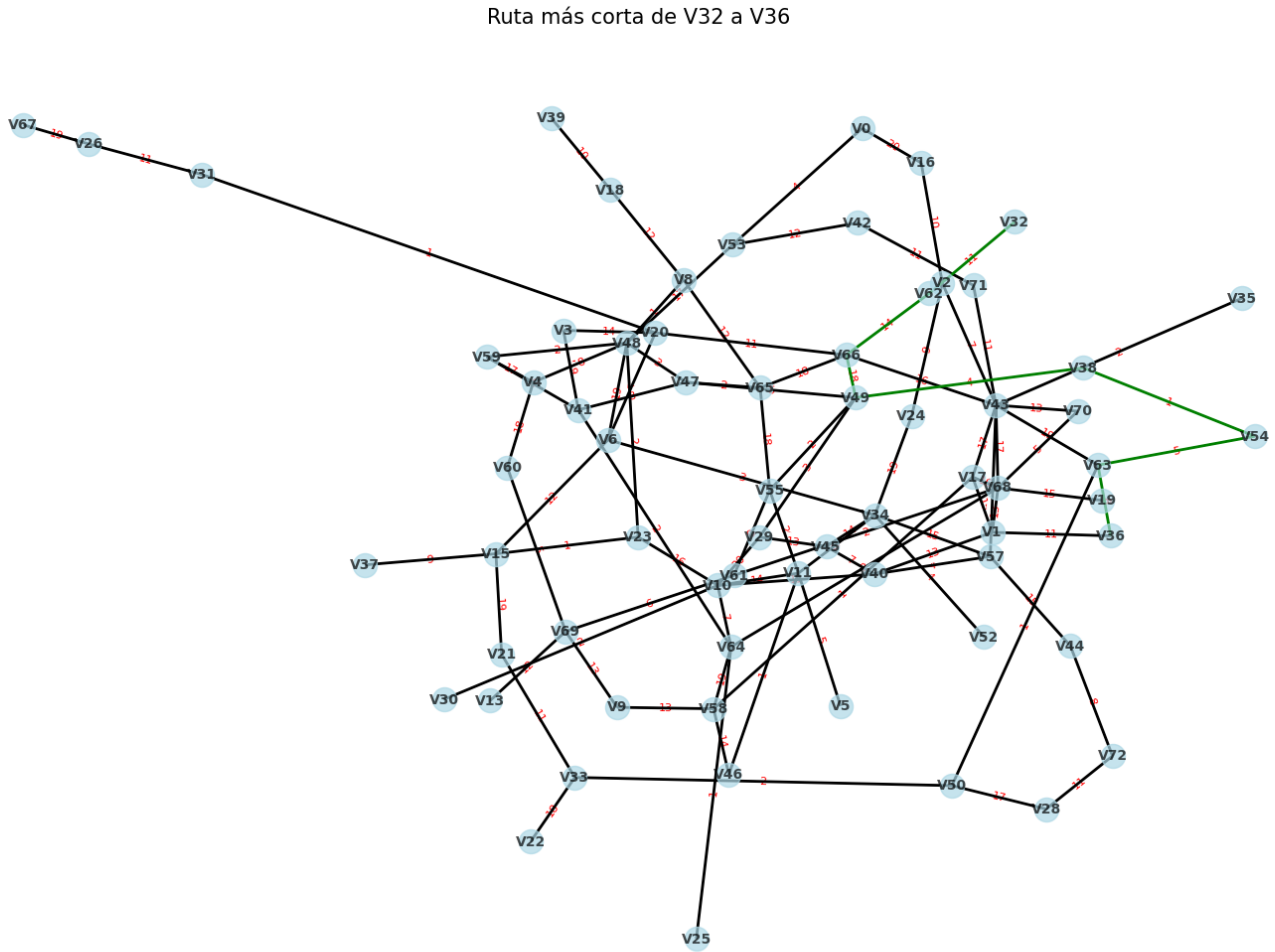
Solución Juego 2: 37 vértices



La distancia mínima entre V31 y V28 es 31

$$V31 \rightarrow V10 \rightarrow V25 \rightarrow V32 \rightarrow V28$$

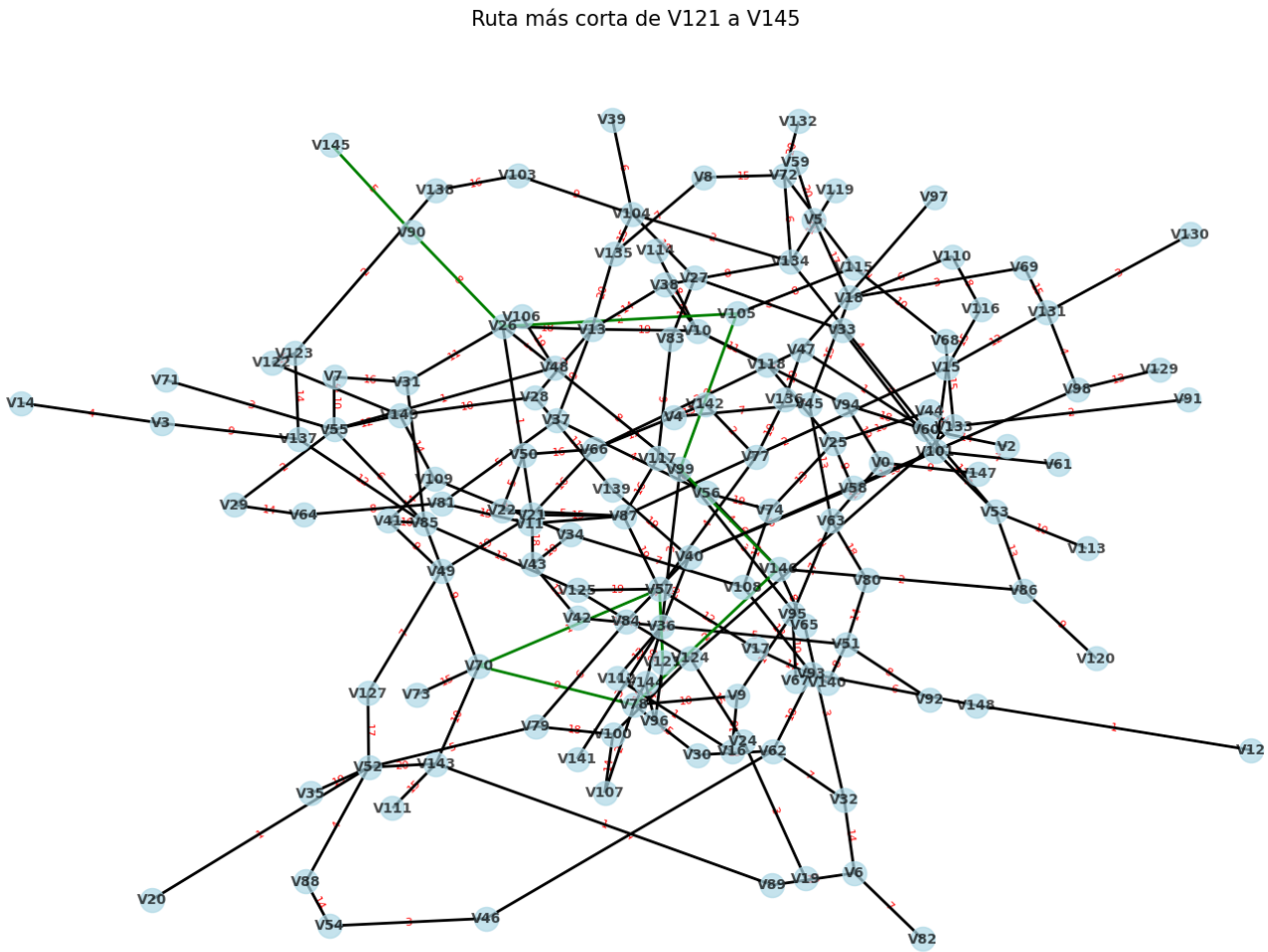
Solución Juego 3: 73 vértices



La distancia mínima entre V32 y V36 es 61

$$V32 \rightarrow V62 \rightarrow V66 \rightarrow V49 \rightarrow V38 \rightarrow V54 \rightarrow V63 \rightarrow V36$$

Solución Juego 4: 150 vértices



La distancia mínima entre V121 y V145 es 50

$$V121 \rightarrow V57 \rightarrow V70 \rightarrow V78 \rightarrow V146 \rightarrow V99 \rightarrow V105 \rightarrow V26 \rightarrow V90 \rightarrow V145$$

Conclusiones

- El algoritmo de Dijkstra es eficiente para encontrar la ruta más corta entre dos vertices en un grafo cuyas aristas tienen diferentes pesos.
- La implementación en Python permite visualizar gráficamente la utilidad de este algoritmo y evidenciar la importancia que tiene cuando la cantidad de aristas y vertices crecen y convierten la tarea de encontrar la ruta más corta entre dos vertices en algo complicado de resolver de forma primitiva.