Instituto Tecnológico Autónomo de México (2024), : 1–19

ESTUDIO DE CASO

Resolución del problema del agente viajero con algoritmos genéticos y búsqueda tabú

Alejandro Naranjo Caraza

Investigación de operaciones, Instituto Tecnológico Autónomo de México.

Contenido

1. Introducción al problema

El Problema del Agente Viajero es un problema clásico de optimización combinatoria y teoría de grafos. El problema parte del siguiente escenario: un agente viajero desea visitar **n** ciudades una sola vez y regresar a su punto de partida. Un viaje entre dos ciudades tiene asociado un costo (o distancia). El agente busca minimizar el costo total durante su trayecto.

Este problema se puede representar fácilmente con un grafo:

Sea $V = \{v_1, \ldots, v_n\}$ el conjunto de nodos.

Sea la matriz $(n \times n)$ de pesos:

$$D = \begin{pmatrix} d_{11} & d_{12} & \cdots & d_{1n} \\ d_{21} & d_{22} & \cdots & d_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ d_{n1} & d_{n2} & \cdots & d_{nn} \end{pmatrix}$$

donde $D_{t_i t_j}$ es la distancia de la ciudad t_i a la ciudad t_j.

Entonces, buscamos una trayectoria de la forma $T = (t_1, t_2, \ldots, t_n, t_1)$ donde $t_i \neq t_j \, \forall i, j = 1, \ldots, n$ con $i \neq j$, que minimiza $\left(\sum_{i=1}^{n-1} d_{t_i t_{i+1}}\right) + d_{t_n t_1}$.

Históricamente, se ha abordado el problema de diversas maneras: fuerza bruta (para encontrar una solución exacta), programación dinámica (algoritmo de Held-Karp) y métodos aproximados y metahurísticas.

En el presente texto, se exploran métodos de aproximación por metahurísticas, en particular, dos algoritmos:

1. Algoritmos genéticos
2. Búsqueda tabú

Para resolver el problema, se utiliza una base de datos con las coordenadas de 929 ciudades en Kenya y se definen las distancias entre dichas ciudades con la distancia euclidiana. Se implementan ambos métodos (y algunas de sus variaciones) por medio de Python, para examinar la eficiencia comparativa entre los algoritmos.

Cabe destacar que estos métodos son de *aproximación* y, por tanto, se comparan los modelos por medio de la *mejor* solución arrojada.

2. Notas previas sobre la implementación de los algoritmos

En realidad, el planteamiento del problema se puede hacer meramente con una matriz de distancias $D_{n\times n}$, pues a partir de esta matriz, podemos definir nuestro conjunto de ciudades $V = \{1, 2, \ldots, n\}$. De esta forma, cualquier trayectoria será una permutación de dichas ciudades $T = [t_1, \ldots, t_n, t_1]$, donde $t_i \in V \, \forall i$ y $t_i \neq t_j \, \forall i \neq j$.

Por ende, en el ejemplo considerado, ambos algoritmos parten de esta matriz. En el siguiente bloque de código, vemos la definición adecuada de las distancias a partir de nuestra matriz de coordenadas.

```
import numpy as np
# definimos objecto para matriz de distancias
#y calculo de distancias totales de trayectorias
class Distancias:
    def __init__(self, path):
        try:
            coord_mat = np.loadtxt(path, usecols=(1, 2), dtype=float)
        except ValueError as e:
            raise ValueError(f"Error al leer el archivo {path}: {e}")
        if coord_mat.ndim != 2 or coord_mat.shape[1] != 2:
            raise ValueError("Error. Config de archivo.")
        dist_mat = np.linalg.norm(coord_mat[:, np.newaxis] - coord_mat, axis=2)
        dist_mat = np.round(dist_mat).astype(int)
        if (not isinstance(dist_mat, np.ndarray) or
                not np.allclose(dist_mat, dist_mat.T)):
            raise ValueError("Debe ser matriz sim trica.")
        elif not dist_mat.dtype == int:
            raise ValueError("Debe ser matriz de enteros.")
        else:
            self.dist_mat = dist_mat
            self.size = dist_mat.shape[0]
    def distancia_total(self, t):
        #calcula distanica total recorrida por una trayectoria
        if not isinstance(t, list):
            raise TypeError("Error. Dtype trayectoria.")
        if len(t)-1 != self.size:
            raise ValueError("Error. Tama o de trayectoria.")
        res = sum(self.dist_mat[t[i], t[i-1]] for i in range(1, len(t)))
        return res
```

A partir de la definición de la clase **Distancias**, podemos definir un objeto que tiene asociado: **1.** Una matriz de distancias y **2.** Un método para obtener la distancia total recorrida por una trayectoria. Nótese que al definir un objeto global, evitamos saturar el uso de la memoria del programa y optimizamos los tiempos de ejecución.

En las siguientes secciones, usaremos un objeto global de esta clase para realizar todos los cálculos de las distancias totales recorridas por nuestras trayectorias. A partir de nuestra matriz de distancias, podremos definir cualquier trayectoria por medio de *listas*, bajo las restricciones mencionadas anteriormente.

3. Algoritmo genético

A grandes rasgos, implementamos el algoritmo genético con la siguiente lógica.

1. **Inicializamos los parámetros**: Definimos el número de ciudades, la población, el número de generaciones y la proporción de mutaciones.
2. **Generamos la población inicial**: Creamos una población de trayectorias aleatorias (listas de ciudades), asegurándonos de que cada trayectoria sea válida (sin repeticiones y cada ciudad esté incluida).
3. **Ordenamos la población**: Ordenamos la población según la distancia total recorrida por cada trayectoria (menor distancia = mejor).
4. **Seleccionamos a los padres para la función de enlace**: Seleccionamos a los mejores individuos de la población (la mitad superior) para formar la próxima generación.
5. **Realizamos la función de enlace de los padres**: Aplicamos una función de enlace (como single-point, order, o PMX) para generar dos hijos a partir de dos padres seleccionados.
6. **Realizamos mutaciones**: Aplicamos mutaciones a los hijos generados, cambiando aleatoriamente las posiciones de las ciudades en la trayectoria con una cierta probabilidad.
7. **Generamos la nueva población**: Combinamos a los padres y los hijos mutados para formar la nueva población.
8. **Repetimos el proceso**: Repetimos los pasos de ordenación, selección, función de enlace y mutación por el número de generaciones especificado.
9. **Obtenemos la solución óptima**: Después de completar todas las generaciones, seleccionamos la trayectoria con la menor distancia total como la solución óptima al TSP.
10. **Devolvemos el resultado**: Mostramos la trayectoria óptima y su distancia total.

Para implementar el algoritmo genético en Python, partimos de un objeto distancias (visto en la sección 2). Entonces, distancias.mat_dist es nuestra matriz de distancias y distancias.distancia_total(trayectoria) nos da la distancia total recorrida por una trayectoria.

2mm

Primero, definimos nuestro algoritmo para generar poblaciones. Nótese que, dado una número de poblaciones específicado **num_poblacion** y un número de ciudades **num_ciudades**, generamos poblaciones compuestas por trayectorias que recorren el número de ciudades especificado.

```
def generar_poblacion(num_poblacion,num_ciudades):
    poblacion=[]
    for i in range (num_poblacion):
        t=list(np.random.permutation(num_ciudades))
        t.append(t[0])
        poblacion.append(t)
    return poblacion
```

Ahora, definimos nuestras funciones de enlace de acuerdo a los algoritmos vistos anteriormente: single point, order crossover y pmx crossover.

```
# Funciones de enlace
def single_point(t1,t2): #single point crossover
    if not (isinstance(t1,list) and isinstance(t2,list)):
        print("Error. Dtype de trayectorias.")
        res=None
    elif len(t1)!=len(t2):
        print("Error. Trayectorias de distinta longitud.")
        res=None
    elif len(t1)==0 or len(t2)==0:
        print("Error. Trayectorias de longitud 0.")
        res=None
    else:
        punto_enlace=np.random.randint(1,len(t1)-1)
        hijo1=t1[:punto_enlace]
        hijo2=t2[:punto_enlace]
        hijo1.extend([i for i in t2 if i not in hijo1])
```

```
        hijo2.extend([i for i in t1 if i not in hijo2])
        res=hijo1,hijo2
    return res

def order_crossover(t1,t2): #order crossover
    n=len(t1)
    punto_enlace1=np.random.randint(0,n//2)
    punto_enlace2=np.random.randint(n//2,n)
    hijo1,hijo2=[-1]*n,[-1]*n
    hijo1[punto_enlace1:punto_enlace2]=t1[punto_enlace1:punto_enlace2]
    hijo2[punto_enlace1:punto_enlace2]=t2[punto_enlace1:punto_enlace2]

    def llenar_hijo(hijo,padre,punto_enlace2):
        indice=punto_enlace2 % n
        for i in padre:
            if i not in hijo:
                hijo[indice]=i
                indice=(indice+1)%n
        return hijo
    hijo1=llenar_hijo(hijo1,t2,punto_enlace2)
    hijo2=llenar_hijo(hijo2,t1,punto_enlace2)
    res=hijo1,hijo2
    return res

def pmx_crossover(t1,t2): #pmx crossover
    n=len(t1)
    punto_enlace1=np.random.randint(0,n//2)
    punto_enlace2=np.random.randint(n//2,n)
    hijo1,hijo2=t1,t2
    for i in range(punto_enlace1,punto_enlace2):
        hijo1[i],hijo2[i]=t2[i],t1[i]
    def reparar_hijo(hijo,padre,punto_enlace1,punto_enlace2):
        for i in range (punto_enlace1,punto_enlace2):
            if hijo[i] in hijo[:punto_enlace1] or hijo[i] in hijo[punto_enlace2:]:
                remplazo=padre[i]
                while remplazo in hijo[:punto_enlace1]
                            or remplazo in hijo[punto_enlace2:]:
                    temp=np.random.randint(-5,5)
                    remplazo=(padre[hijo.index(remplazo)]+temp)%n
                hijo[i]=remplazo
        return hijo
    hijo1=reparar_hijo(hijo1,t2,punto_enlace1,punto_enlace2)
    hijo2=reparar_hijo(hijo2,t1,punto_enlace1,punto_enlace2)
    return hijo1,hijo2
```

Finalmente, la última función que debemos definir se encarga de las mutaciones entre trayectorias. Esto es, dada una probabilidad de mutación p y una trayectoria, el algoritmo le realiza un cambio entre ciudades aleatorias a la trayectoria el $p\%$ de las veces.

```
def mutacion(t,proporcion_mutaciones):
    if not isinstance(t,list):
        print("Error. Dtype de trayectoria.")
        res=None
    if len(t)==0:
        print("Error. Trayectoria de longitud 0.")
        res=None
    else:
        for i in range (len(t)):
            if np.random.rand()< proporcion_mutaciones:
                k=np.random.randint(len(t))
                t[k],t[i]=t[i],t[k]
        res=t
    return res
```

El algoritmo genético implementado en Python utiliza las funciones definidas anteriormente para aproximar la trayectoria óptima.

```
def algoritmo_genetico(distancias,num_poblacion,
                                iteraciones,proporcion_mutaciones,enlace):
    if not isinstance(distancias,Distancias):
        print("Error. Dtype Distancias.")
        return None
    elif not (isinstance(num_poblacion,int) and isinstance(iteraciones,int)
              and isinstance(proporcion_mutaciones,float)):
        print("Error. Dtype")
        return None
    else:
        num_ciudades=len(distancias.dist_mat)
        poblacion=generar_poblacion(num_poblacion,num_ciudades)
        ordenacion=lambda x: distancias.distancia_total(x)
        for i in range(iteraciones):
            print("Iteracion",i+1)######################
            poblacion=sorted(poblacion,key=ordenacion)
            poblacion_siguiente=poblacion[:num_poblacion // 2]
            while len(poblacion_siguiente)<num_poblacion:
                indice1, indice2 = np.random.choice(len(poblacion),2,replace=False)
                t1,t2=poblacion[indice1],poblacion[indice2]
                hijo1,hijo2=enlace(t1,t2)
                hijo1=mutacion(hijo1,proporcion_mutaciones)
                hijo2=mutacion(hijo2,proporcion_mutaciones)
                poblacion_siguiente.extend([hijo1,hijo2])
            poblacion=poblacion_siguiente
        res=poblacion[0]
    return res
```

4. Búsqueda tabú

A grandes rasgos, implementamos el algoritmo de búsqueda tabú de la siguiente forma:

1. **Generamos una trayectoria inicial aleatoria:**
   - Generamos una trayectoria inicial de ciudades de forma aleatoria, asegurándonos de que el recorrido empiece y termine en la misma ciudad.
2. **En cada iteración, mejoramos la distancia con búsqueda tabú:**
   - En cada iteración, buscamos el *subviaje óptimo*. Cada *subviaje* es el resultado de eliminar y agregar dos aristas.
   - Durante la búsqueda, mantenemos una lista de *movimientos tabú* de aristas que no podemos eliminar. Al encontrar un *subviaje óptimo*, actualizamos esta lista con las aristas que agregamos para formarlo.
3. **Iteramos:**
   - Repetimos el último proceso iterativamente, tomando cada *suviaje óptimo* de la iteración anterior como el *subviaje inicial* de la actual.
   - Después de un número determinado de iteraciones, nos detenemos.
   - Almacenamos la mejor trayectoria encontrada.
4. **Devolvemos el resultado final:**
   - Al terminar las iteraciones devolvemos la *mejor trayectoria encontrada* y su *distancia total*.

Importamos nuestras bibliotecas.

```
from collections import deque
import numpy as np
import copy
```

Primero, definimos una función para generar una sola trayectoria mediante una permutación aleatoria (esta será nuestra trayectoria inicial).

```
def generar_trayectoria(num_ciudades):
    t=list(np.random.permutation(num_ciudades))
```

```
    t.append(t[0])
    return t
```

Ahora, la siguiente función encuentra una sub trayectoria óptimo a partir de trés criterios:

- Trayectoria inicial
- Objeto para calcular distancias
- Lista de artistas que no se pueden eliminar (Tabú)

```
def subviaje_optimo(t,distancias,tabu):
    subviaje_optimo=None
    distancia_optima=distancias.distancia_total(t)
    n=len(t)
    for i in range(1,n-1):
        print(i)
        for j in range(i+1,n-1):
            t_sub=copy.copy(t)
            t_sub[i:j]=t[i:j][::-1]
            lig_x_eliminar1={t[i-1],t[i]}
            lig_x_eliminar2={t[j],t[j+1]}
            dist=distancias.distancia_total(t_sub)
            if dist<distancia_optima and lig_x_eliminar1
                    not in tabu and lig_x_eliminar2 not in tabu:
                distancia_optima=dist
                subviaje_optimo=t_sub
                lig_agregada1={t[i-1],t[j]}
                lig_agregada2={t[j+1],t[i]}
                tabu.append(lig_agregada1)
                tabu.append(lig_agregada2)
    return subviaje_optimo
```

Finalmente, definimos nuestro algoritmo de búsqueda tabú a partir con base en las funciones definidas anteriormente. Nótese que en cada iteración, podemos obtener una trayectoria mejor o peor (mayor o menor distancia total). Después de un número dado de iteraciones, el algoritmo se detiene.

```
def tabu_search(distancias,iteraciones):
    tabu=deque(maxlen=4)
    t=generar_trayectoria(distancias.dist_mat.shape[0])
    t_optima=t
    distancia_optima=distancias.distancia_total(t)
    for i in range(iteraciones):
        t_sub=subviaje_optimo(t,distancias,tabu)
        if t_sub==None:
            break
        t=t_sub
        dist=distancias.distancia_total(t)
        if dist<distancia_optima:
            distancia_optima=dist
            t_optima=t
    return t_optima,distancia_optima
```

5. Resultados

5.1. Algrotimo genético

En el caso del algoritmo genético, tenemos tres parámetros que podemos configurar: el tamaño de la población, el número de generaciones y el porcentaje de mutaciones.

Entonces, con base en el algoritmo, **hipotetizamos** lo siguiente:

- La distancia de la trayectoria óptima obtenida es mínima cuando se implementa un mayor número de iteraciones (generaciones), aunque con mejoras decrecientes.
- Las distancias obtenidas son menores para una proporción óptima de mutaciones.
- Las distancias obtenidas son menores para un tamaño óptimo de la población.

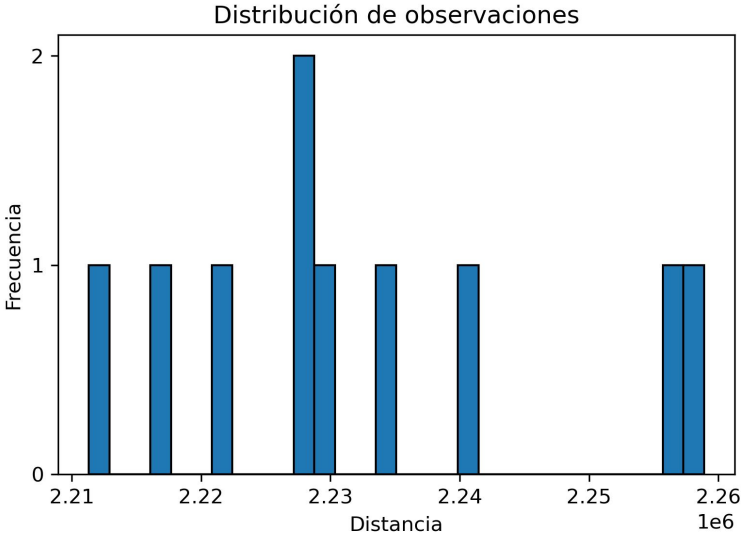
Para corroborar estos supuestos, ejecutamos el algoritmo $n = 10$ veces y analizamos los tiempos de ejecución del programa, así como las distancias óptimas obtenidas.

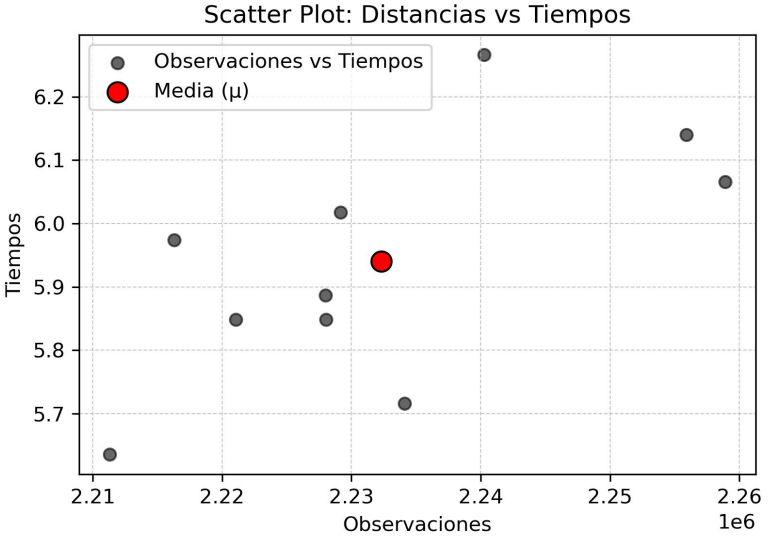
5.1.1. Variando generaciones

Fijamos el tamaño de la población en 100 y la proporción de mutaciones en 0.05 y vemos cómo se comportan las distancias obtenidas al correr el algorimo.

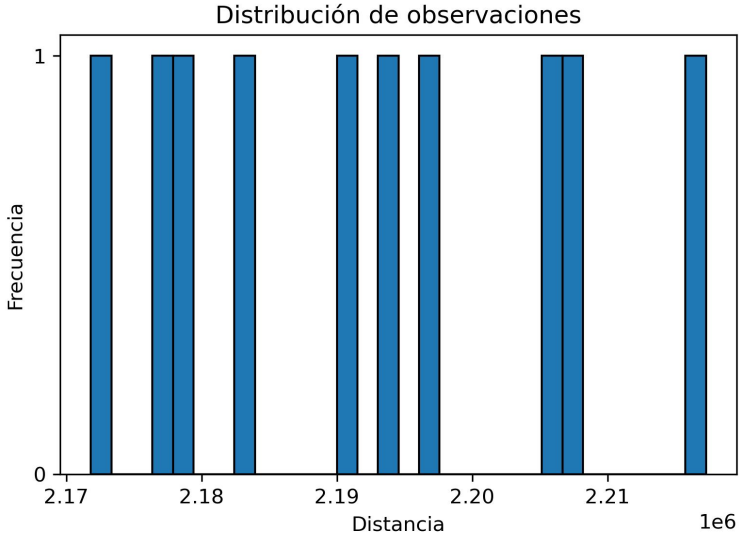
Con 10 generaciones, obtenemos una distancia promedio **2,232,299** y tiempo de ejecución promedio de **5.94** segundos.

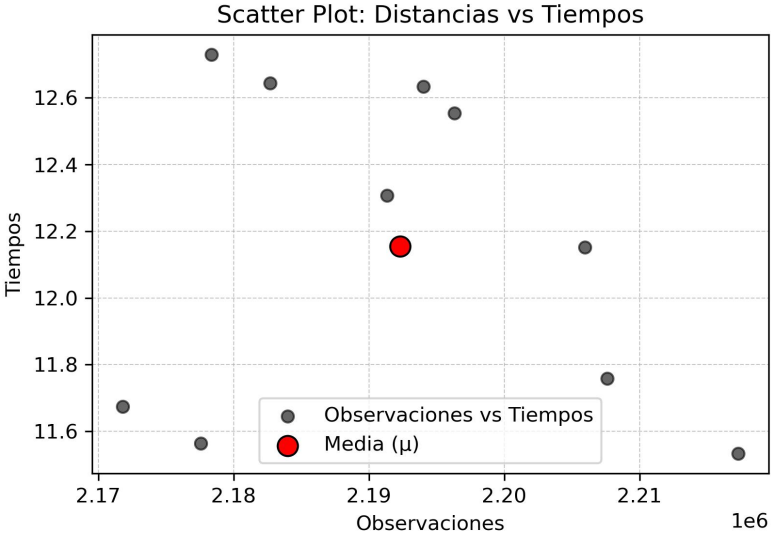
Se pueden observar las distribuciones en las siguientes figuras.



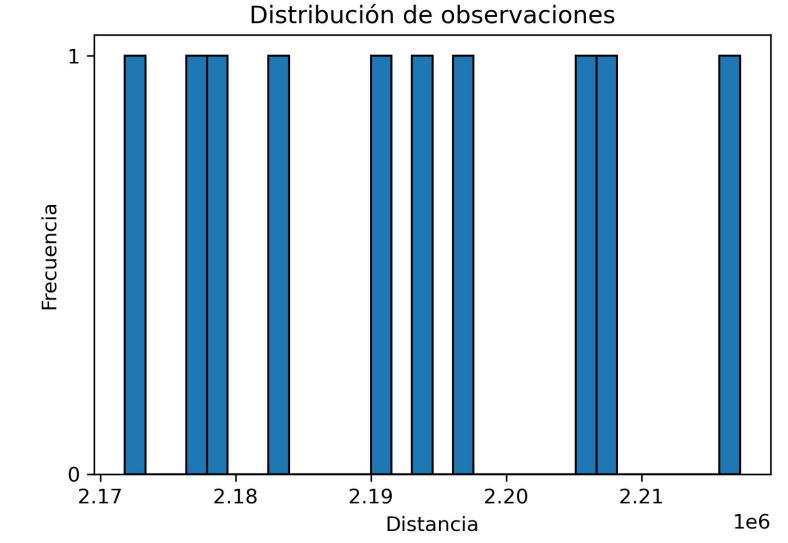


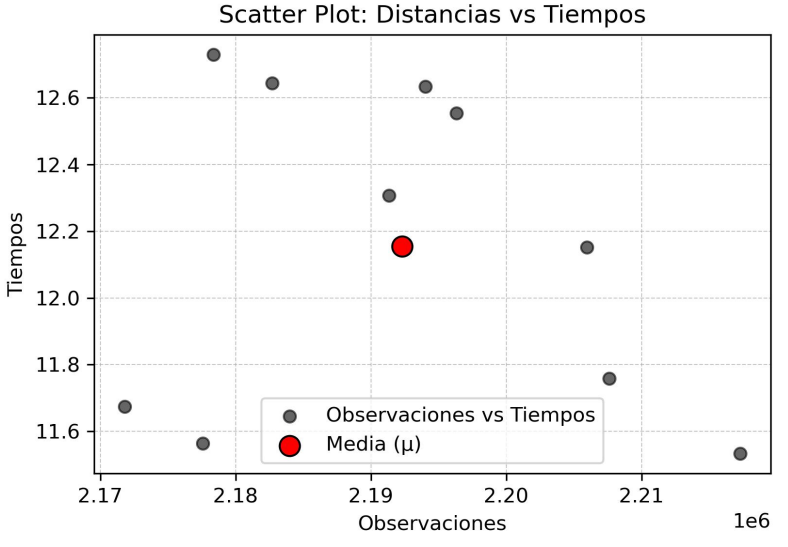
Con 10 generaciones, obtenemos una distancia promedio **2,192,299** y tiempo de ejecución promedio de **12.16** segundos.



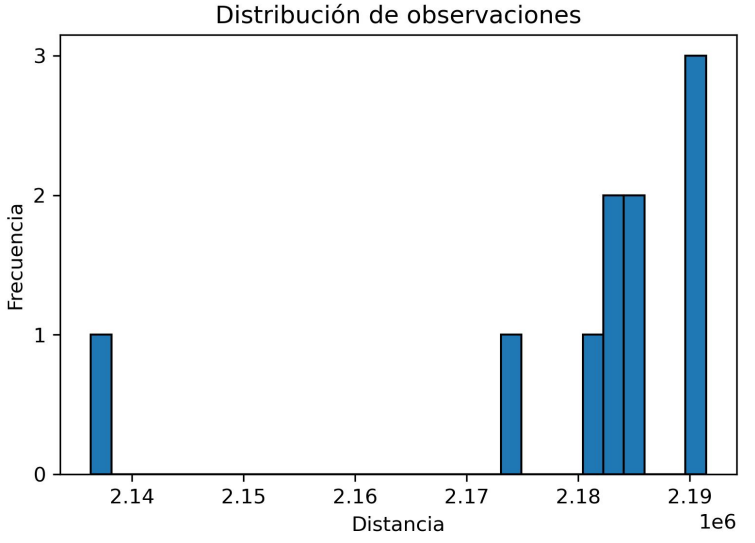


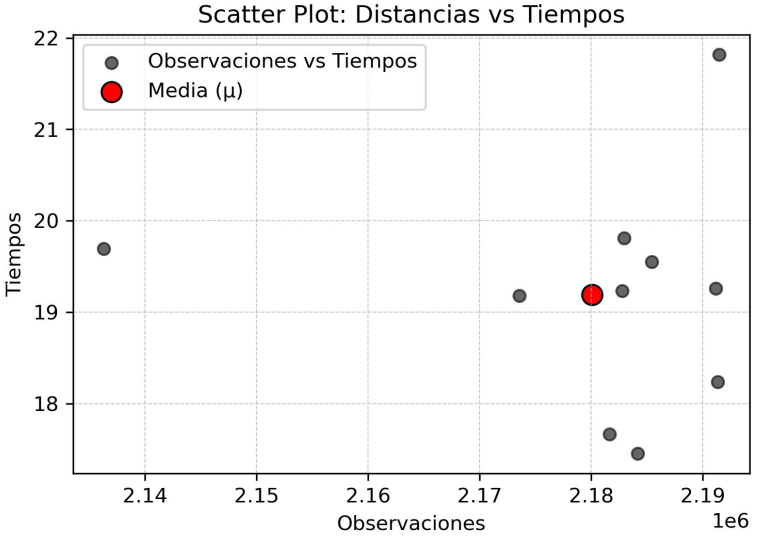
Con 20 generaciones, obtenemos una distancia promedio **2,180,077** y tiempo de ejecución promedio de **19.19** segundos.



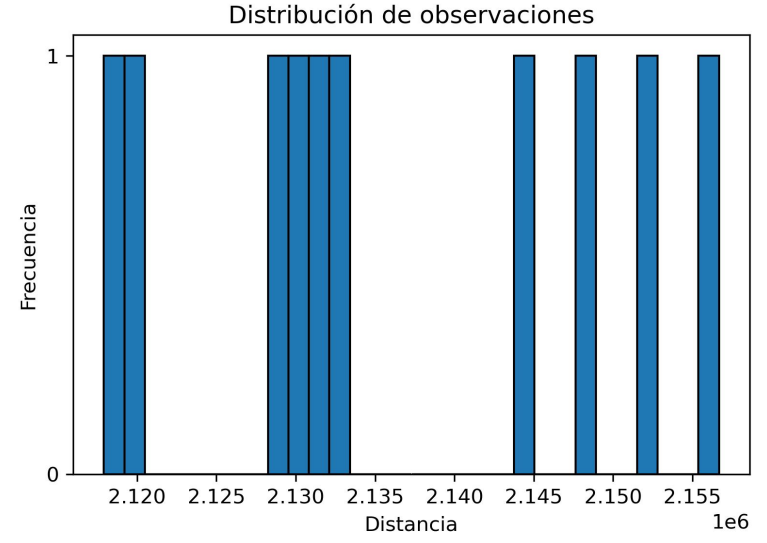


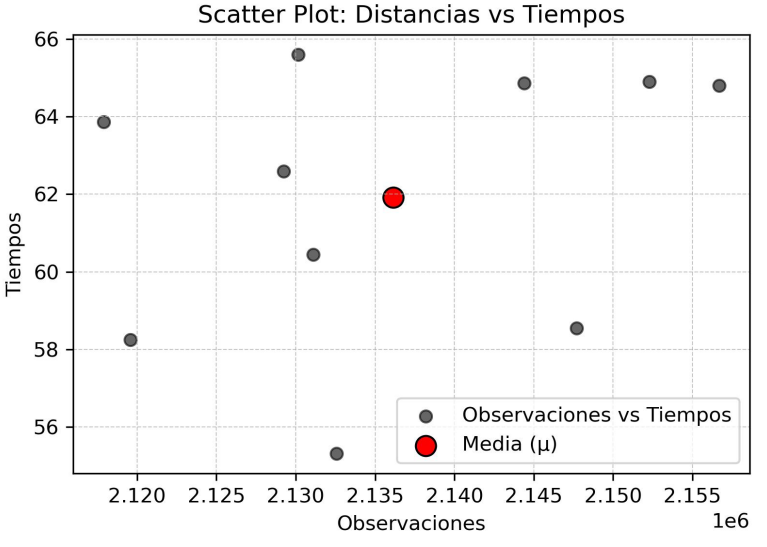
Con 30 generaciones, obtenemos una distancia promedio **2,180,159** y tiempo de ejecución promedio de **19.16** segundos.





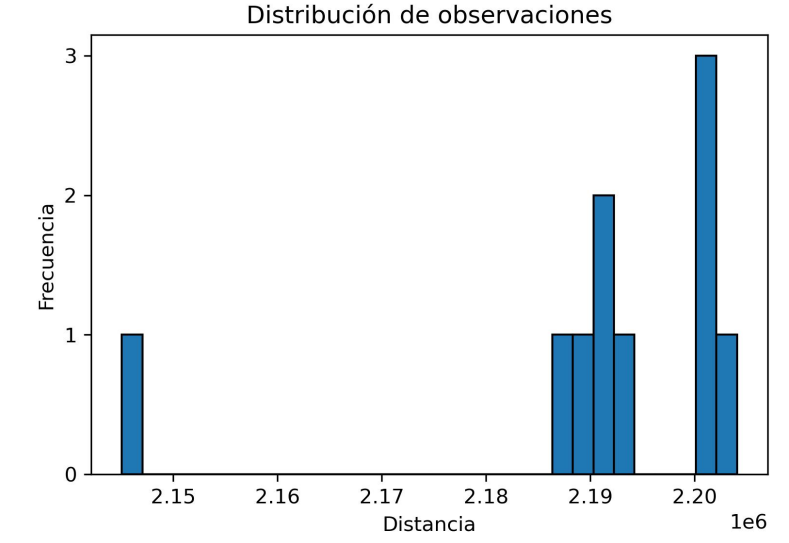
Con 90 generaciones, obtenemos una distancia promedio **2,136,159** y tiempo de ejecución promedio de **61.91** segundos.

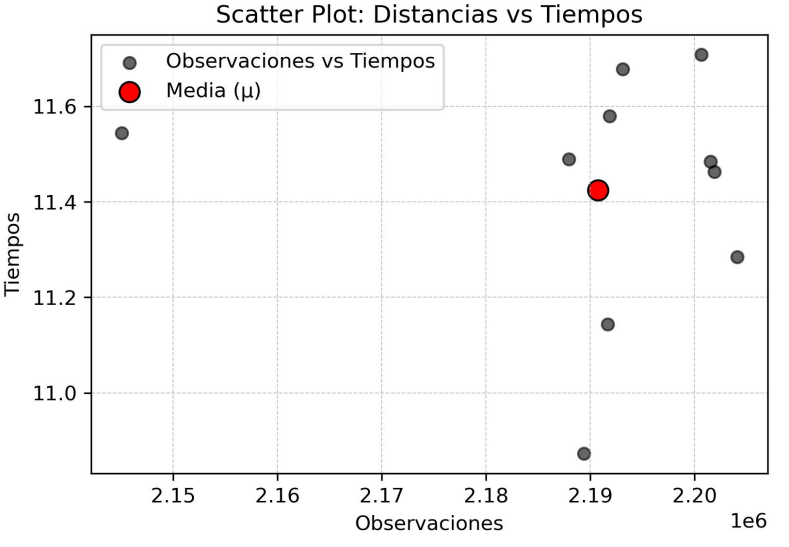




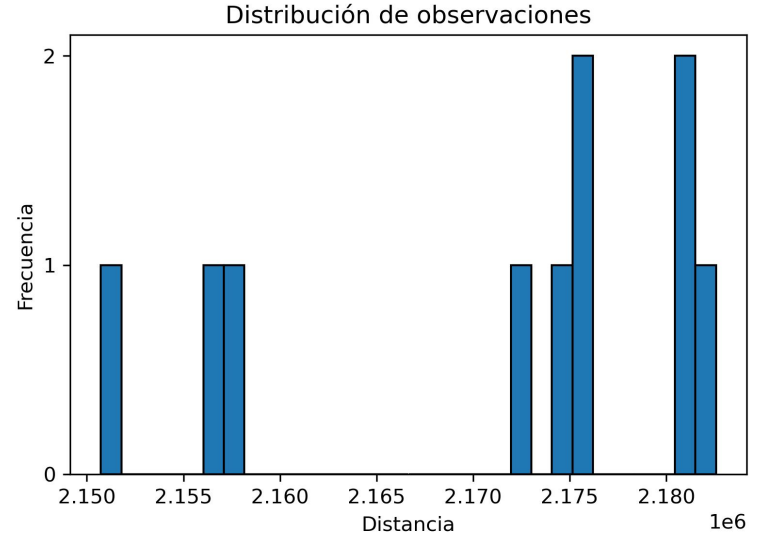
5.1.2. Variando tamño de población

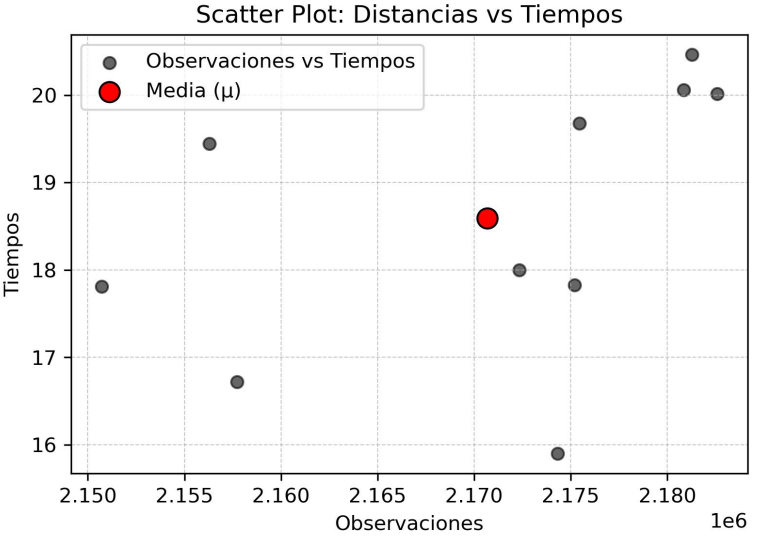
Con 60 trayectorias por población, obtenemos una distancia promedio **2,190,717** y tiempo de ejecución promedio de **11.42** segundos.



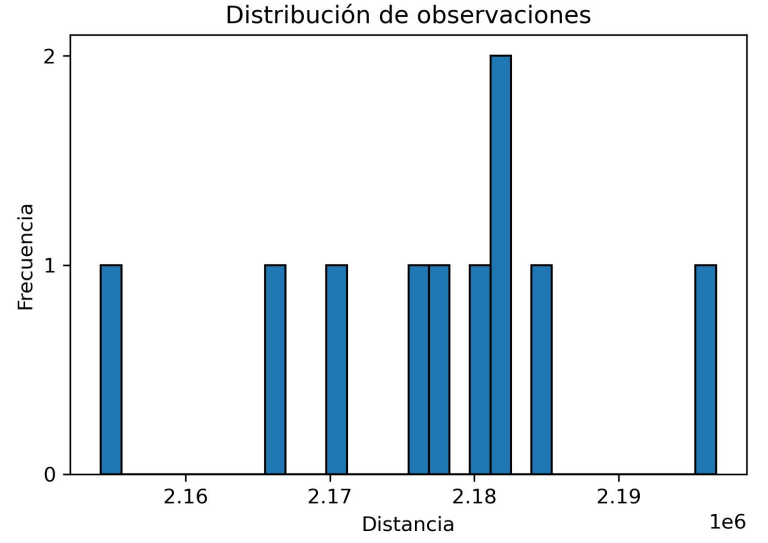


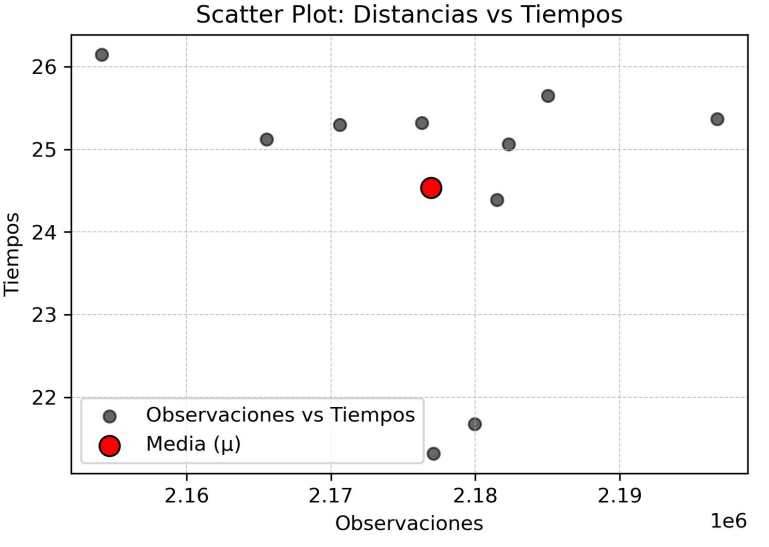
Con 90 trayectorias por población, obtenemos una distancia promedio **2,170,675** y tiempo de ejecución promedio de **18.59** segundos.





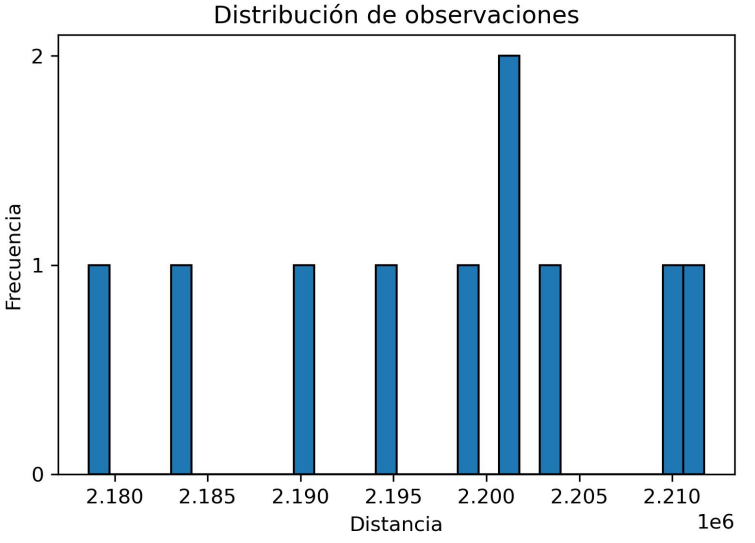
Con 120 trayectorias por población, obtenemos una distancia promedio **2,176,925** y tiempo de ejecución promedio de **24.53** segundos.

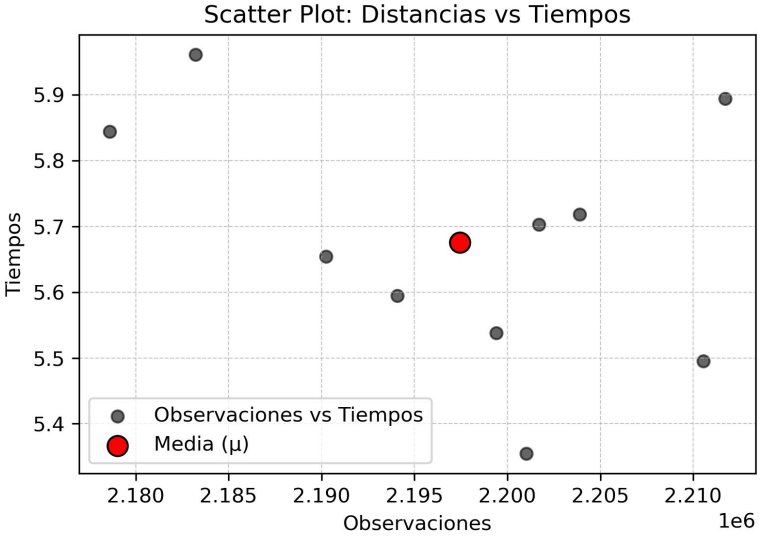




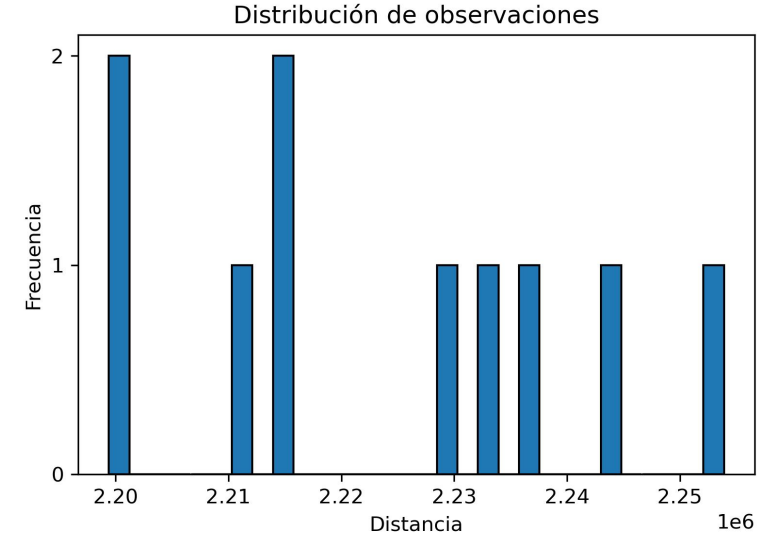
5.1.3. Variando proporción de mutaciones

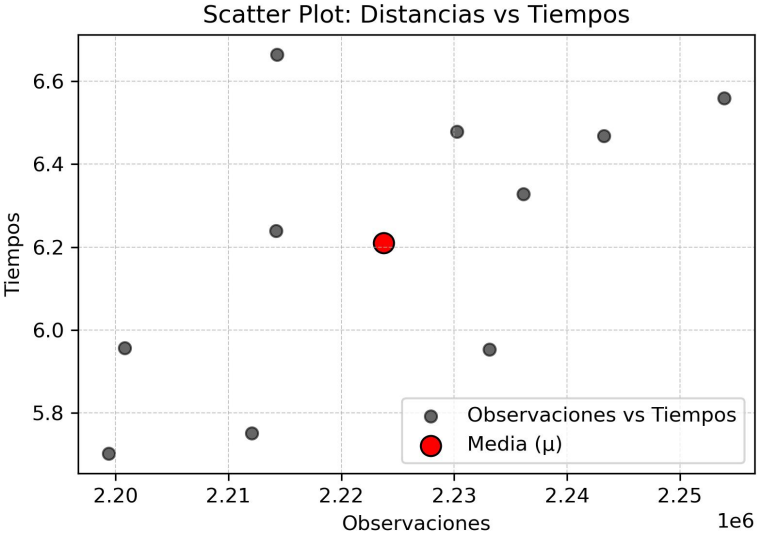
Con una **probabilidad de mutación de 0.05**, obtenemos una distancia óptima de **2,197,442** en promedio en aproximadamente **5.6** segundos.



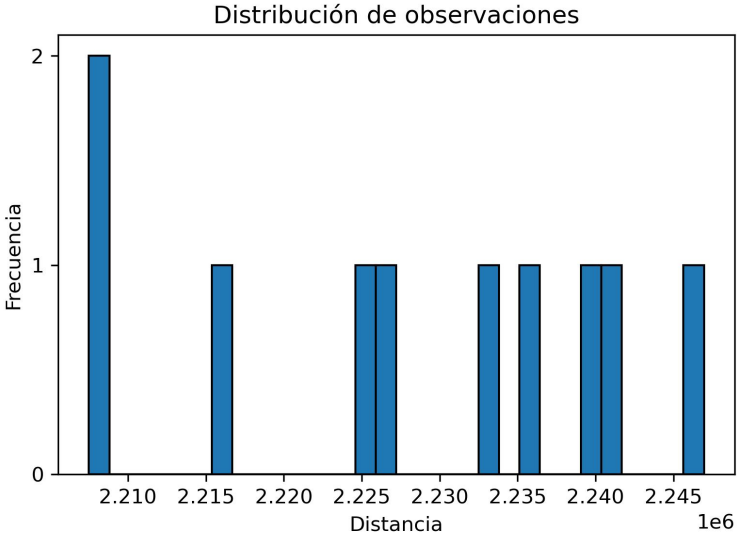


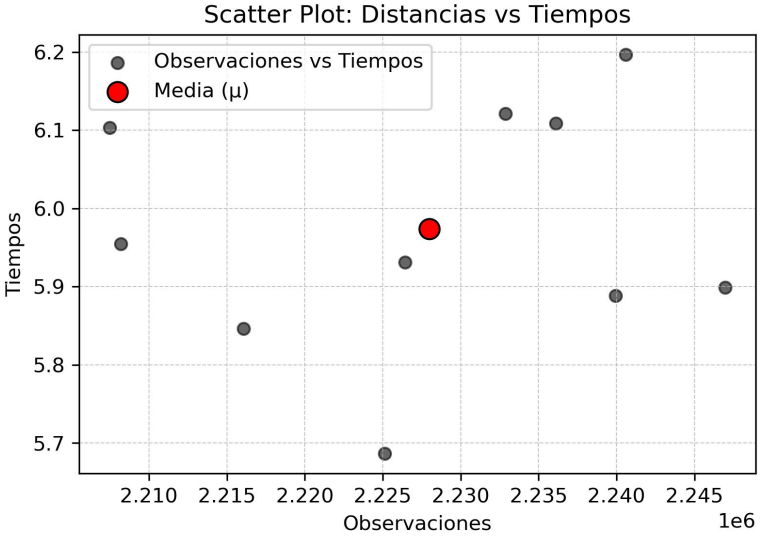
Con una **probabilidad de mutación de 0.10**, obtenemos una distancia óptima de **2,223,749** en promedio en aproximadamente **6.2** segundos.



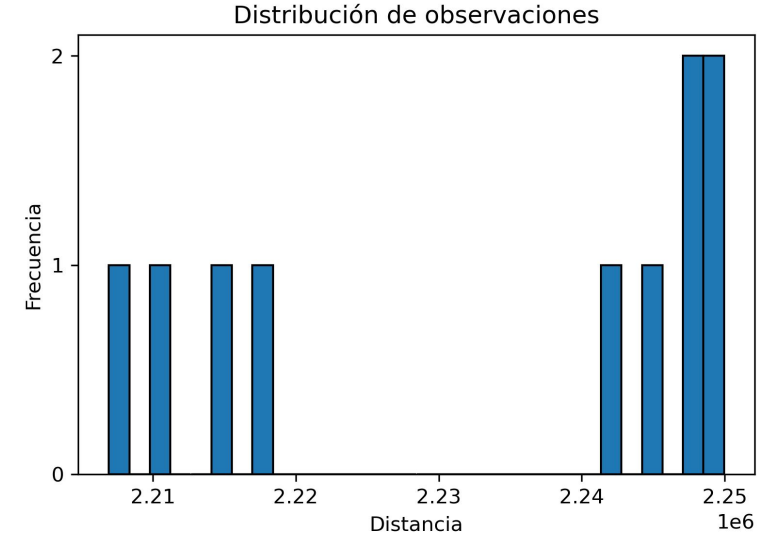


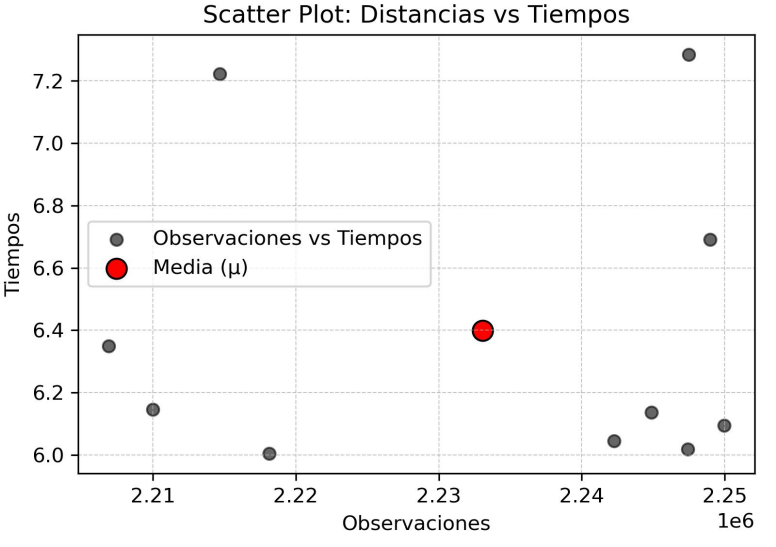
Con una **probabilidad de mutación de 0.15**, obtenemos una distancia óptima de **2,227,980** en promedio en aproximadamente **6.0** segundos.





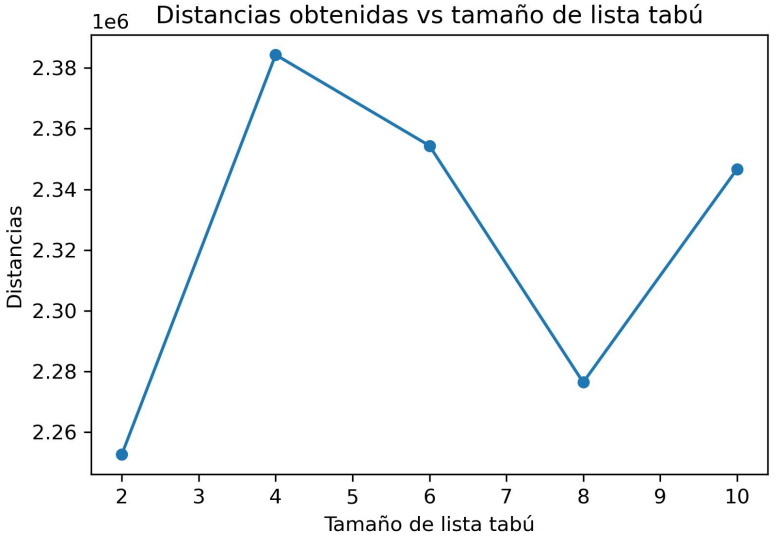
Con una **probabilidad de mutación de 0.20**, obtenemos una distancia óptima de **2,233,067** en promedio en aproximadamente **6.4** segundos.





5.2. Búsqueda tabú

En el caso de la búsqueda tabú, los tiempos de ejecución del algoritmo fueron de más de 180 segundos para todos los tamaños de la lista tabú que se probaron.
En la siguiente figura, mostramos las distancias obtenidas para cada tamaño de la lista tabú. Nótese que la mínima distancia se obtuvo con una lista tabú de dos aristas.



6. Optimización de los parámetros

Debido al fundamento aleatorio del algoritmo (nótese que generamos las trayectorias iniciales, las mutaciones y los enlaces de manera aleatoria), podemos suponer que la distancia óptima (mínima) obtenida al aplicarlo se comporta como una variable aleatoria cuya distribución desconocemos y cuya función de probabilidad general depende de los tres parámetros mencionados anteriormente: $p(x|\theta = (\textit{tamaño de población, proporción de mutaciones}))$.
Entonces, podemos considerar el siguiente problema de optimización estocástica:

$$g(\textit{tamaño de población, proporción de mutaciones}) = \mathbb{E}[X|\theta] \tag{6.1}$$

En caso de tener *tamaños de población* y *proporción de mutaciones* óptimos, si se cumple (como podríamos hipotetizar) que la función g es convexa, podemos considerar la posibilidad de aplicar algoritmos de optimización convexa para optimizar los parámetros descritos.

Bibliografía

Goodrich, Michael T., Tamassia, Roberto, and Goldwasser, Michael H. Data Structures and Algorithms in Python. Wiley, 2013.
Hillier F.S. and Liberman G.J. "Introducción a la Investigación de Operaciones", McGraw-Hill, 9a Edición (2010)
Widodo, Abel K. "Exploring Heuristics Analytics with Python."Medium, Medium, 28 July 2023, medium.com/@abelkrw/exploring-heuristics-analytics-with-python-31b5c6e7e526.
Talbi, El-Ghazali. Metaheuristics: From Design to Implementation. John Wiley and Sons, 2009.