INSTITUTO TECNOLÓGICO AUTÓNOMO DE MÉXICO

Investigación de Operaciones | Proyecto final – Reporte

Juan Carlos Sigler Alonso Martinez Rebeca Angulo Carlos Galeana
Jacqueline Lira

Índice general

1 Marco teórico

1.1 Formulación como problema de programación lineal

Planteamos el problema de encontrar un *tour*, es decir una ruta cerrada que pasa por todas las ciudades, sin repetir ninguna y regresando a la ciudad de origen como un problema de minimización. Para el acercamiento mediante algoritmos genéticos planteamos lo siguiente para un individuo dado.

Sea $G = [g_1, g_2, \ldots, g_n]$ el *genoma* del individuo. El genoma se puede representar como una lista ordenada de números g_i con $g_1 \leq g_i \leq g_n$ que representan el índice dado de una ciudad. Cada ciudad tiene un índice único y lo usamos como su nombre. El conjunto C es el conjunto de los índices de todas la ciudades.

Entonces, podemos formular el problema como el siguiente problema de programación lineal en forma estándar:

$$\min \sum_{i=1}^{n-1} \|C_i - C_{i+1}\| + \|C_n - C_1\| \tag{1}$$

Sujeto a:

$$\begin{aligned} \sum_{i \in G} 1 &= |C| \text{ Se deben visitar todas las ciudades} \\ g_i \neq g_j \quad \forall i &\neq j \text{ No se repite ninguna ciudad en el tour} \end{aligned}$$

1.2 Algoritmos genéticos

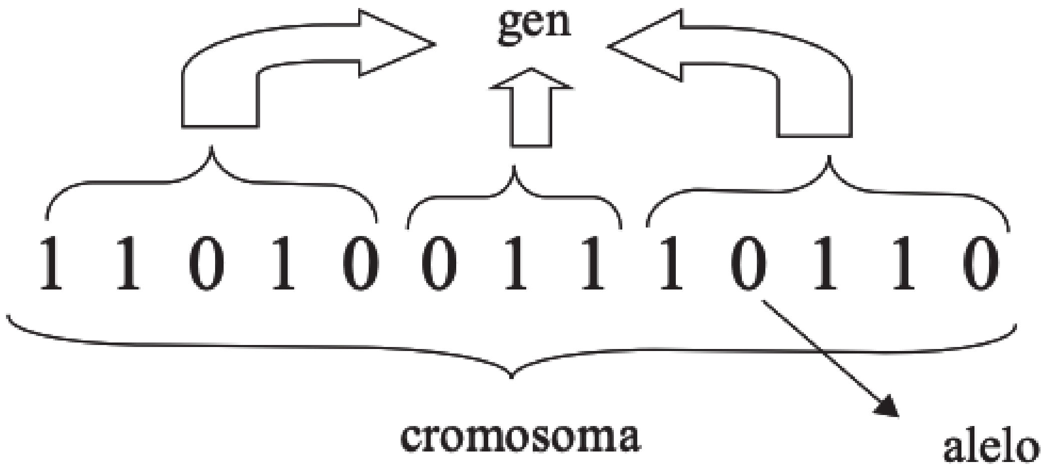
Los algoritmos genéticos son algoritmos de optimización, búsqueda y aprendizaje inspirados en los procesos de evolución natural y evolución genética. La evolución es un proceso que opera sobre los cromosomas. La selección natural, expuesta en la teoría de la evolución biológica por Charles Darwin (1859), es un mecanismo que relaciona los cromosomas (genotipo) con el fenotipo (caracteres observables) y otorga a los individuos más adaptados un mayor número de oportunidades de reproducirse, lo cual aumenta la probabilidad de que sus características genéticas se repliquen.

Los procesos evolutivos tienen lugar durante la etapa de reproducción, algunos de los mecanismos que afectan a la reproducción son la mutación, causante de que los cromosomas en la descendencia sean diferentes a los de los padres y el cruce que combina los cromosomas de los padres para producir una nueva descendencia.

En un algoritmo genético para alcanzar la solución a un problema se parte de un conjunto inicial de individuos, llamado población, el cual es generado de manera aleatoria. Cada uno de estos individuos representa una posible solución al problema. Se construye una función objetivo mejor conocida como función *fitness*, ya definida en la ecuación (1), y se definen los *adaptive landscapes*, los cuales son evaluaciones de la función objetivo para todas las soluciones candidatas. Por medio de una función de evaluación, se establece una medida numérica, la cual permite controlar en número de selecciones, cruces y copias. En general, esta medida puede entenderse como la probabilidad de que un individuo sobreviva hasta la edad de reproducción.

1.2.1 Representación

Para trabajar con las características genotípicas de una población dotamos a cada individuo de un *genotipo*. En nuestra implementación éste se representa como una lista de índices de ciudades. En general, el genotipo es se puede representar como una cadena de bits que se manipula y muta.



1.3 Operadores genéticos

Una generación se obtiene a partir de la anterior por medio de operadores, mejor conocidos como operadores genéticos. Los más empleados son los operadores de selección, cruce, copia y mutación, los cuales vamos a utilizar en la implementación del algoritmo.

1.3.1 Selección:

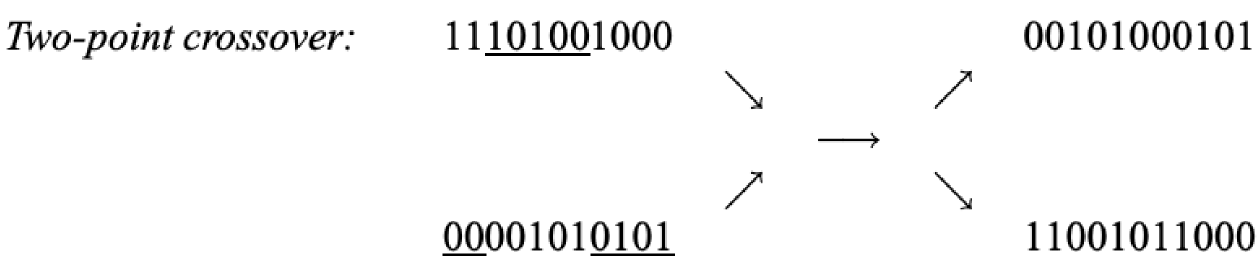
Es el mecanismo por el cual son seleccionados los individuos que serán los padres de la siguiente generación. Se otorga un mayor número de oportunidades de reproducción a los individuos más aptos. Existen diversas formas de realizar una selección, por ejemplo: 1. Selección por truncamiento 2. Selección por torneos 3. Selección por ruleta 4. Selección por jerarquías

Los algoritmos de selección pueden ser divididos en dos grupos: probabilísticos, en este grupo se encuentran los algoritmos de selección por ruleta, y determinísticos, como la selección por jerarquías.

En nuestro algoritmo utilizamos la selección por ruleta, donde cada padre se elige con una probabilidad proporcional a su desempeño en relación con la población.

1.3.2 Cruce:

Consiste en un intercambio de material genético entre dos cromosomas de dos padres y a partir de esto se genera una descendencia. Existen diversas formas de hacer un cruce, en nuestro algoritmo utilizamos el cruce de dos puntos.



La idea principal del cruce se basa en que si se toman dos individuos correctamente adaptados y se obtiene una descendencia que comparta genes de ambos, al compartir las características buenas de dos individuos, la descendencia, o al menos parte de ella, debería tener una mayor bondad que cada uno de los padres.

1.3.3 Mutación:

Una mutación en un algoritmo genético causa pequeñas alteraciones en puntos determinados de la codificación del individuo, en otras palabras, produce variaciones de modo aleatorio en un cromosoma. Por lo general primero se seleccionan dos individuos de la población para realizar el cruce y si el cruce tiene éxito entonces el descendiente muta con cierta probabilidad.

1.3.4 Copia:

Consiste simplemente en la copia de un individuo en la nueva generación. Un determinado número de individuos pasa directamente a la siguiente generación sin sufrir variaciones.

1.4 Implementación

A continuación presentamos el pseudocódigo del algoritmo que implementaremos. Nos basamos principalmente en [M K19] y en [Tae21].

Algoritmo 1: GA(n, χ, μ)

Result: individuo más apto de P_k

Inicializamos generación 0;
$k := 0$
$P_k :=$ población de n individuos generados al azar;
Evaluar P_k :
do
  Crear generación $k + 1$;
  1. Copia:;
  Seleccionar $(1 - \chi) \times n$ miembros de P_k e insertar en P_{k+1}
  2. Cruce $k + 1$;
  Seleccionar $\chi \times n$ miembros de P_k; emparejarlos; producir descendencia; insertar la descendencia en P_{k+1}
  3. Mutar:;
  Seleccionar $\mu \times n$ miembros de P_{k+1}; invertir bits seleccionados al azar
  Evaluar P_{k+1};
  Calcular $fitness(i)$ para cada $i \in P_k$
  Incrementar: $k := k + 1$;
while *el fitness del individuo más apto en* P_k *no sea lo suficientemente bueno*;

n es el número de individuos en la población. χ es la fracción de la población que será reemplazada por el cruce en cada iteración. $(1 - \chi)$ es la fracción de la población que será copiada. μ es la tasa de mutación.

En cuanto a los criterios de terminación de nuestro algoritmo, nosotros indicamos que debe detenerse cuando alcance el número de generaciones máximo especificado.

El código de *python* utilizado para los resultados se adjunta al final de este documento como un anexo en interés de la brevedad y legibilidad de este reporte.

1.5 Vecinos más cercanos

El algoritmo de Vecinos más cercanos es otra heurística pensada para resolver el problema del agente viajero mediante una estrategia *greedy*. A diferencia de un algoritmo genético, vecinos más cercanos planea una ruta mediante un criterio simple: si se minimiza la distancia recorrida al recorrer una ciudad más, es sensato pensar que se minimiza la distancia total del tour. Entonces, se selecciona una ciudad al azar para empezar el tour, y se calculan las ciudades más cercanas sucesivamente hasta que no quede ninguna.

Esta idea queda retratada en el siguiente algoritmo presentado como pseudocódigo:

Algoritmo 2: Algoritmo vecinos más cercanos

Result: Ruta elegida con vecinos más cercanos a partir de ciudad inicial

Comenzamos con un conjunto de ciudades por visitar y un conjunto de visitados
$c_0 \leftarrow$ ciudad elegida al azar.
$c_a \leftarrow c_0$ fijamos la ciudad actual.
$V \leftarrow \emptyset$ ciudades visitadas
$C \leftarrow \{c_1, \ldots, c_n\}$ ciudades por visitar
while $|V| \neq |C|$ **do**
  $V \leftarrow V \cup \{c_a\}$
  $c^* \leftarrow \min\{d(c_a, c_i) \mid c_i \in C \backslash V\}$
  $c_a \leftarrow c^*$

Cabe mencionar que para este trabajo, implementamos un algoritmo genético y otro híbrido con el propósito de comparar su desempeño en los datos del país de Qatar.

El algoritmo híbrido que desarrollamos consta en una mezcla de las estrategias de vecinos más cercanos con algoritmos genéticos. Nuestro algoritmo difiere de uno genético en que la población inicial no se genera solo aleatoriamente. Damos la posibilidad de elegir qué porcentaje de esa población son individuos generados con la estrategia de vecinos más cercanos. Después de la generación de esta población inicial se deja que el algoritmo continue con el proceso de mutación y cruza como lo haría sin modificaciones; solo interferimos en la creación de la población inicial.

La idea detrás de este algoritmo híbrido es fomentar una convergencia más rápida a un óptimo auténtico introduciendo un "super gen" que generación a generación se irá haciendo más común en la población por la ventaja que da. Además, esperamos que la estrategia de mutación permitiera mejorar paulatinamente una solución que ya era en si muy buena, y llegar a un óptimo global real y no solo uno local. En la siguiente sección hablamos con más detalle de lo que sucedió realmente.

2 Resultados

Para dar contexto, presentamos primero un mapa de el país seleccionado: Qatar.

Seleccionamos este país porque su baja densidad permite ver sin mucho esfuerzo qué rutas son más óptimas que otras. Por ejemplo, cruces de un lado a otro del mapa indican rutas sub-óptimas.

2.1 Pruebas

Para los resultados que se presentan a continuación se corrieron 10,000 generaciones del algoritmo genético tanto en su versión estándar como la versión híbrida. Para asegurar la reproducibilidad de estos resultados se fijó el *seed* de el generador de números aleatorios. De esta manera aseguramos que las versiones del algoritmo híbrido que comparamos más tarde comenzaron en la misma ciudad.

En la figura siguiente, presentamos cómo evoluciona la distancia total del tour a medida que avanzan las generaciones.

Algunas cosas resultan aparentes de la figura. Por ejemplo: se puede notar que el algoritmo genético es efectivo y si logra reducir la distancia total recorrida generación a generación. Es decir el algoritmo está bien implementado. Dicho eso, también se puede ver con claridad que hay varias puntos en la gráfica en los cuales la disminución de distancia total entre generaciones sucesivas es cero. El algoritmo se estanca.

Otra detalle a notar es que la distancia total recorrida en el caso del algoritmo híbrido es casi constante. Se empieza con una distancia muy buena, y la mutación y cruza puede hacer muy poco para mejorarla. Incluso después de 10 mil generaciones.

En la siguiente tabla se presentan los resultados de la distancia total y las rutas propuestas en concreto.

Algoritmo	Ciudad de inicio	Ciudad final	Distancia total
Algoritmo genético	61	111	32,855.6
Algoritmo híbrido	35	0	11,330.3

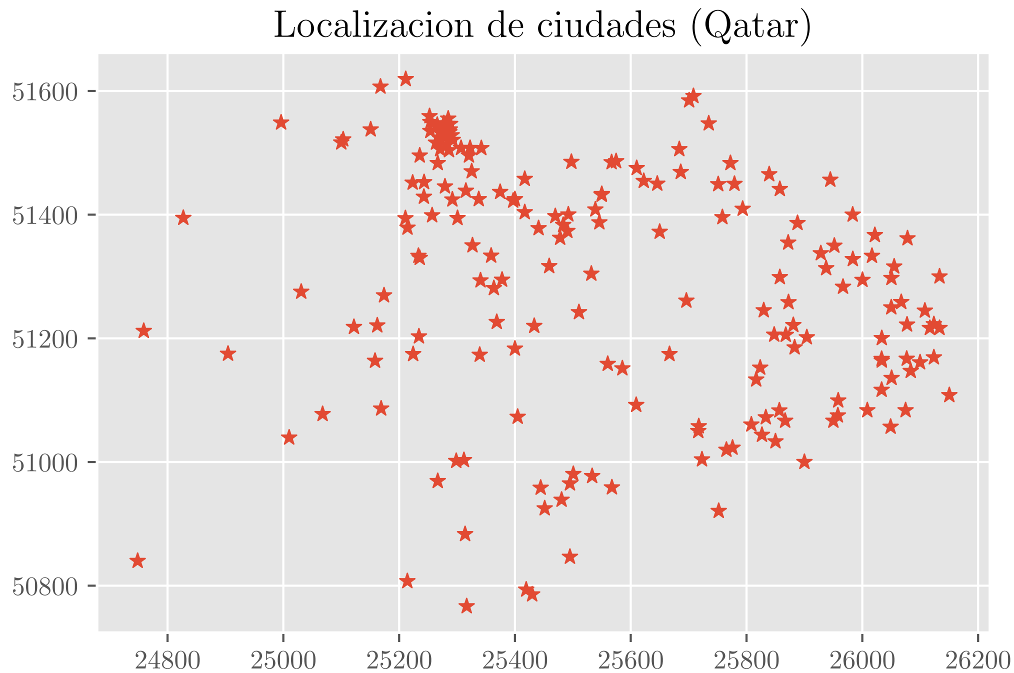


Figura 1: Qatar

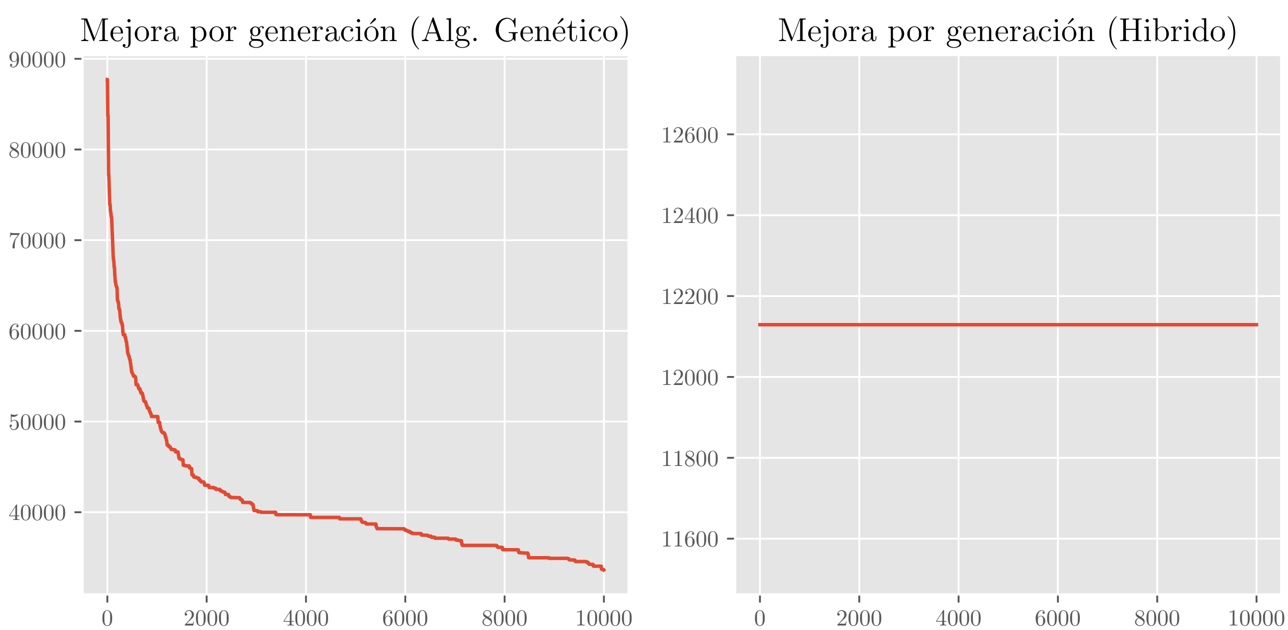


Figura 2: Disminución de distancia de tour

La siguiente figura es una comparación directa de la distancia del tour que proponen el algoritmo genético y el híbrido. Es clara la diferencia abismal.

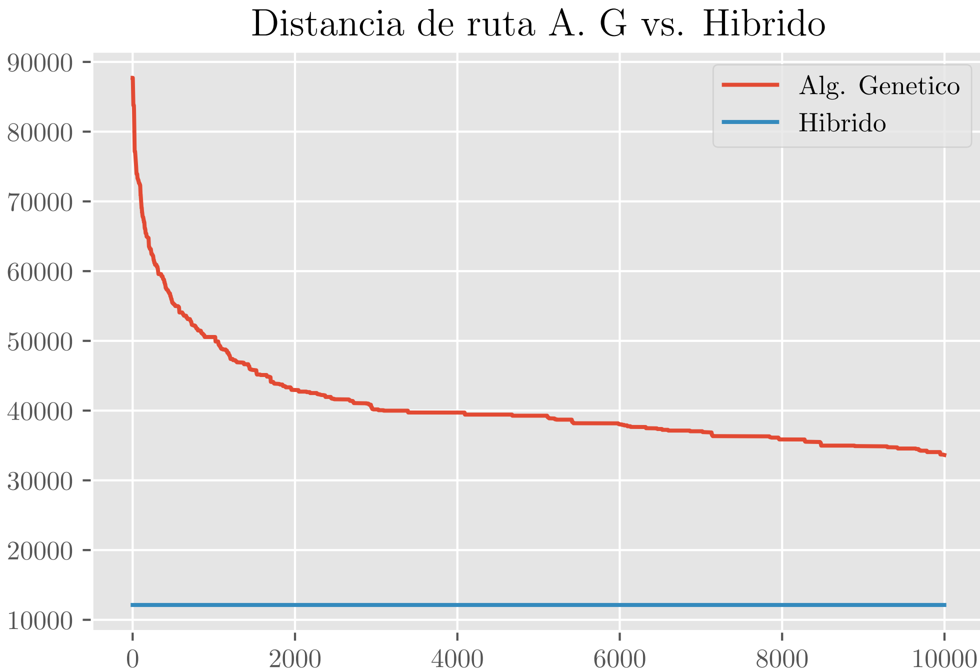


Figura 3: Comparación directa de las distancias

Finalmente, para hacer claro cómo se comparan en desempeño ambos algoritmos presentamos la siguiente gráfica que está dividida en 4 subgráficas. Como sugieren los títulos, en la primera fila se encuentra la comparación de los tours propuestos por el algoritmo genético (izquierda) contra el algoritmo híbrido (derecha) ambos al final de 10,000 generaciones. En la fila de abajo, el análogo pero para apenas 10 generaciones.

El ver los tours graficados directamente sobre el mapa de el país permite ver con claridad qué ruta es mejor, ya que es fácil ver que una ruta de apariencia menos caótica es más eficiente en la distancia total.

Esta figura resulta ilustrativa de dos fenómenos importantes: Primero que nada, se nota que el algoritmo genético si está encontrando rutas cada vez mejores, pero lo hace a costa de mucho cómputo pesado y tiempo. Después, se puede ver que el algoritmo híbrido tiene rutas por mucho superiores a su contraparte, y además es claro que estas no mejoran de manera obvia bajo mutación incluso a través de varios miles de generaciones. Lo cual sugiere que eran rutas muy aceptables desde el inicio.

Como adicional incluimos el tour completo a continuación.

```
array([ 35,  58,  61,  81,  79,  86,  75,  74,  77,  71,  73,  68,  59,
        56,  44,  28,  21,  27,  32,  17,  20,  23,  25,  16,  13,  10,
        12,  22,  24,  70, 101, 102,  90,  92,  95,  94,  96,  91,  87,
        82,  80,  78,  76,  69,  63,  67,  65,  66,  60,  57,  55,  52,
        51,  47,  45,  40,  37,  39,  42,  46,  50,  38,  33,  30,  31,
        29,  34,  41,  48,  54,  53,  43,  49,  36,  26,  11,   9,   8,
         4,  14,  18,  72,  83,  99, 109, 111, 114, 115, 116, 120, 119,
       127, 122, 123, 132, 134, 128, 130, 135, 147, 142, 154, 150, 146,
       151, 152, 149, 143, 153, 156, 140, 138, 137, 141, 145, 148, 144,
       139, 136, 133, 131, 126, 124, 125, 113, 112, 108, 118, 121, 117,
       105, 104, 106, 107, 157, 158, 161, 166, 169, 170, 165, 159, 184,
       179, 177, 180, 176, 183, 187, 190, 188, 191, 189, 186, 185, 182,
       178, 171, 168, 175, 181, 193, 173, 172, 174, 167, 164, 192, 163,
       162, 160, 155, 129, 110, 103, 100,  98,  93,  89,  88,  97,  85,
```

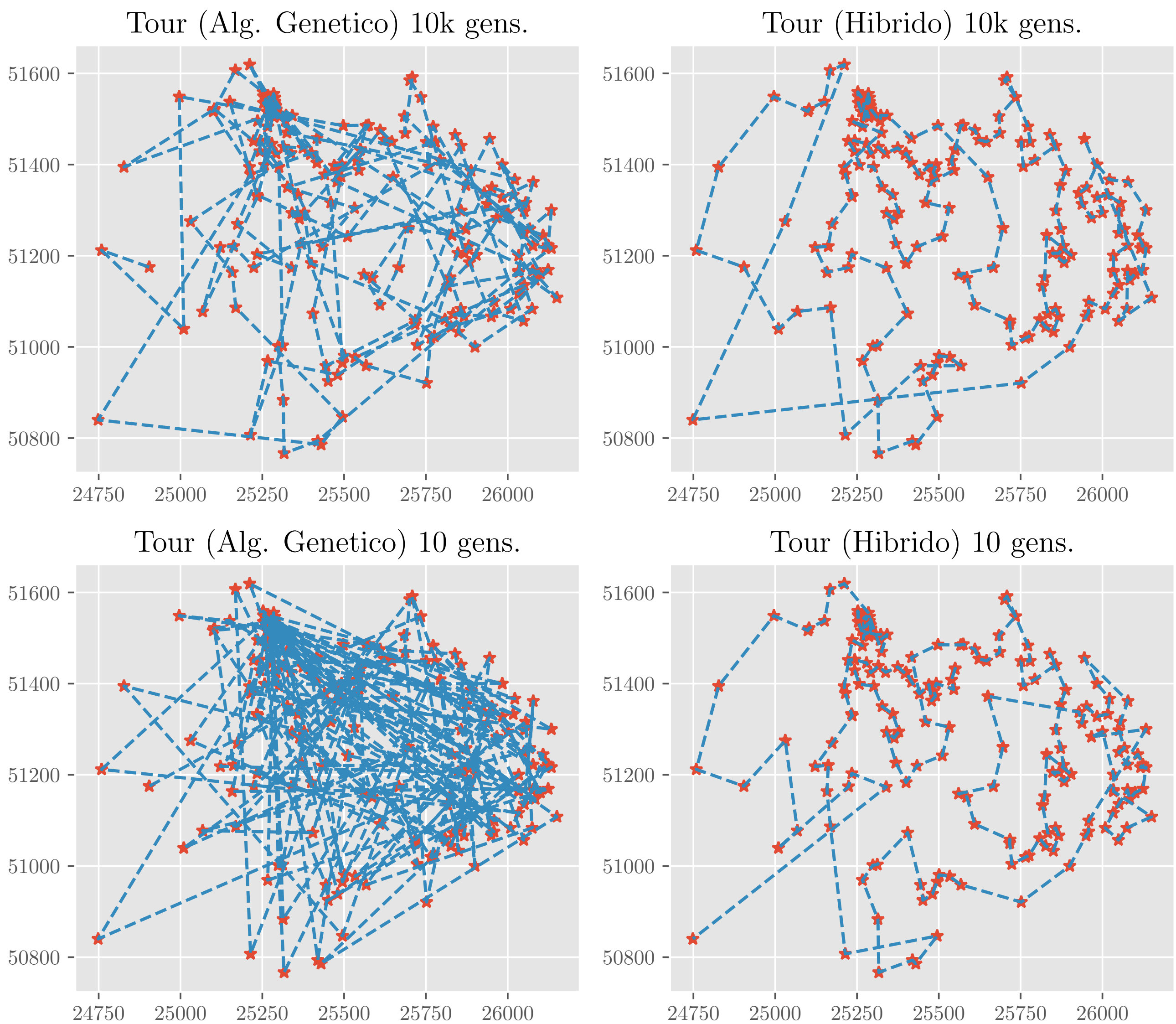


Figura 4: Tours

```
        84,  64,  19,  62,  15,   7,   5,   3,   1,   2,   6,   0])
```

Parece ser que hay cierto orden a como se nos proporcionaron los datos de ciudades. El tour muestra una preferencia a escoger ciudades con índices consecutivos, lo cual sugiere que los datos estaban ordenados desde un inicio.

3 Conclusión

De los resultados anteriores podemos ver el comportamiento de ambos algoritmos con mucha claridad. En particular vale la pena hacer notar lo siguiente:

1. El algoritmo genético es muy caro computacionalmente hablando y en general propone rutas poco óptimas
2. El algoritmo híbrido llega muy rápido a un óptimo local y no cambia mucho después de eso. Lo cual podría sugerir que la solución propuesta es de inicio muy buena o que el componente genético no es lo suficientemente bueno como para privilegiar las ventajas tan pequeñas que podría dar la mutación.

En resumen, se podría decir que bajo estas condiciones particulares y la implementación específica de ambos algoritmos, resulta mucho más conveniente resolver el problema del agente viajero mediante una estrategia greedy como vecinos más cercanos y el algoritmo híbrido que hacerlo mediante un algoritmo genético. En este caso particular, parece ser que el algoritmo híbrido es poco efectivo, porque llega a un óptimo desde la primera iteración y por el resto de las generaciones y mutaciones no mejora. Entonces no hace falta el componente genético y se puede lograr una solución satisfactoria con una sola aplicación del algoritmo de vecinos más cercanos.

4 Código en python

Listing 1: Implementación de algoritmos descritos

```python
# +
import matplotlib
import tikzplotlib
import matplotlib.pyplot as plt

matplotlib.use("pgf")

plt.rcParams.update(
    {"pgf.texsystem": "pdflatex", "font.family": "serif", "font.serif": []}
)

plt.style.use("ggplot")

# +
import numpy as np
import numpy.linalg as la

# matplotlib.use("module://matplotlib-backend-kitty")

# np.set_printoptions(threshold=0)
np.random.seed(42)

# %matplotlib inline
# -

# Obtenemos las ciudades de los datos dados para el proyecto procesados
cities = np.genfromtxt("csv/Qatar.csv", delimiter=",")
cities = cities[1:, 1:]
n_cities = cities.shape[0]
```

```python
n_cities

# +
# Mapa del país
plt.scatter(cities[:, 0], cities[:, 1], marker="*")
plt.title("Localizacion de ciudades (Qatar)")

plt.savefig("qatar.pdf")


# -
def d(i, j):
    """
    Función de distancia entre las ciudades i,j calculada como la norma de la diferencia
    entre los vectores de coordenadas de las ciudades.

    Args:
        i,j: Ints que funcionen como índices válidos de cities
    """
    return la.norm(cities[i, :] - cities[j, :])


class Individual:
    """
    Representación de un individuo en la población. Tiene como características su genoma y
    los métodos para calcular fitness, llevar a cabo mutaciones y el two-point crossover
    """

    def __init__(self, genome):
        self.genome = genome
        self.fitness = sum(
            [d(genome[i], genome[i + 1]) for i in range(0, len(genome) - 1)]
        ) + d(genome[len(genome) - 1], genome[0])

    # Muta al individuo
    def mutate(self):
        genome = np.copy(self.genome)
        i, j = np.random.choice(len(genome), size=2, replace=False)
        genome[i], genome[j] = genome[j], genome[i]
        return Individual(genome)

    # Two point crossover
    def cross(self, q):
        child = np.copy(self.genome)
        start, end = np.sort(np.random.choice(len(child), size=2, replace=False))
        child[:start] = child[end + 1 :] = -1
        child[child == -1] = np.setdiff1d(q.genome, child, assume_unique=True)
        return Individual(child)

    def __lt__(self, other):
        return self.fitness < other.fitness

    def __gt__(self, other):
        return self.fitness > other.fitness

    def __repr__(self):
        return "Individual(genome: {0}, fitness: {1})".format(
            self.genome.__str__(), self.fitness
        )


i = Individual(np.array([0, 5, 10, 15]))
```

```
i.fitness

o = i.mutate()
o.genome, o.fitness, i.genome, i.fitness

a = Individual(np.array([1, 2, 3, 4, 5]))
b = Individual(np.array([2, 1, 3, 5, 4]))
c = a.cross(b)
a.genome, b.genome, c.genome


def greedy_popuation(n_population, ratio=1 / 2):
    """
    Genera población parte aleatoria parte generada mediante vecinos más cercanos.

    Args:
        n_population: Tamaño total de la población
        ratio: Qué porcentaje de la población se genera greedily. 0 -> toda random
    """
    # Guardamos la población generada en population
    population = []

    # Primero generamos los de vecinos más cercanos
    n_greedy = round(n_population * ratio)
    n_random = n_population - n_greedy

    for _ in range(n_greedy):
        # Anotamos las ciudades visitadas modificando una copia de la lista de ciudades. Las
        # visitadas se vuelven nan. Asi se excluyen del cálculo de distancias.
        visitados = []
        index = range(len(cities))
        # Elegimos ciudad al azar
        curr_city = np.random.randint(0, len(cities))
        visitados.append(curr_city)
        city_record = np.copy(cities)

        while len(visitados) != len(index):
            city_record[curr_city] = np.nan

            distancias = la.norm(city_record - cities[curr_city], axis=1)
            # Tomamos la ciudad más cercana como actual
            curr_city = np.nanargmin(distancias)
            visitados.append(curr_city)

        # Añadiendo el individuo a la población
        population.append(Individual(visitados))

    # Generamos el resto de la población aleatoriamente
    population = population + [
        Individual(np.random.permutation(n_cities)) for _ in range(n_random)
    ]
    return np.array(population)


def calculate_wheel_probability(population):
    """
    Calcula la probabilidad de reproducción para cada individuo en `population`. El de menor
    fitness es el que tiene más probabilidades de reproducirse
    """
    fitnesses = np.array([p.fitness for p in population])
    fitnesses = np.max(fitnesses) + 1 - fitnesses
    s = np.sum(fitnesses)
```

```
    return fitnesses / s


calculate_wheel_probability([i, o])


def GA(
    n_population=100, n_generation=1000, cross_rate=0.3, mutate_rate=0.2, greedy_rate=0,
):
    """
    Resuelve el problema del agente viajero mediante una versión modificada de la estrategia
    de algoritmos genéticos. Se genera una población del tamaño especificado y con las
    características de aleatoriedad deseadas.

    Args:
        n_population: Tamaño total de la población
        n_generation: Número de generaciones
        cross_rate:
        mutate_rate:
        greedy_rate: Porcentaje de la población inicial que se genera mediante vecinos
        verbose: Controla la cantidad de información que imprime el algoritmo
    """
    # Para la generación 0
    # Pk = random_population(n_population)
    history = []
    Pk = greedy_popuation(n_population, greedy_rate)
    best_individual = Pk[Pk.argmin()]
    for k in range(1, n_generation):
        # Creamos la siguiente generacion
        Pk_next = np.array([])
        # Para seleccionar usamos wheel roulette selection
        # Calculamos la wheel probability
        wheel_prob = calculate_wheel_probability(Pk)
        # 1. Copy: seleccionamos (1 - cross_rate) × n individuos de Pk y los insertamos en
        # Pk+1
        Pk_next = np.append(
            Pk_next,
            np.random.choice(
                Pk, round((1 - cross_rate) * n_population), p=wheel_prob, replace=True
            ),
        )
        # 2. Crossover: seleccionamos (cross_rate * n) parejas de Pk y los cruzamos para
        # añadirlos en Pk+1
        parejas = np.random.choice(
            Pk, 2 * round(cross_rate * n_population), p=wheel_prob, replace=True
        ).reshape(-1, 2)
        Pk_next = np.append(Pk_next, [p.cross(q) for p, q in parejas])

        # 3. Mutate: seleccionamos mutate_rate de la población Pk+1 y la mutamos
        mutate_index = np.random.choice(
            len(Pk_next), int(mutate_rate * len(Pk_next)), replace=True
        )
        Pk_next[mutate_index] = np.array([x.mutate() for x in Pk_next[mutate_index]])

        # Acualizamos la generación
        Pk = Pk_next
        if Pk[Pk.argmin()] < best_individual:
            best_individual = Pk[Pk.argmin()]

        history.append(best_individual)

    return best_individual, history
```

```


# + tags=[]
best_ga, hist_ga = GA(n_population=15, n_generation=10000)
best_nn, hist_nn = GA(n_population=15, n_generation=10000, greedy_rate=9 / 10)
# -

best_ga_sm, hist_ga_sm = GA(n_population=15, n_generation=10)
best_nn_sm, hist_nn_sm = GA(n_population=15, n_generation=10, greedy_rate=9 / 10)

# +
fig, axs = plt.subplots(2, 2, figsize=(8, 7))

# Tour GA
axs[0, 0].plot(cities[best_ga.genome][:, 0], cities[best_ga.genome][:, 1], "*")
axs[0, 0].plot(cities[best_ga.genome][:, 0], cities[best_ga.genome][:, 1], "--")
axs[0, 0].set_title("Tour (Alg. Genetico) 10k gens.")

# Tour NN
axs[0, 1].plot(cities[best_nn.genome][:, 0], cities[best_nn.genome][:, 1], "*")
axs[0, 1].plot(cities[best_nn.genome][:, 0], cities[best_nn.genome][:, 1], "--")
axs[0, 1].set_title("Tour (Hibrido) 10k gens.")

# Tour NN chico
axs[1, 0].plot(cities[best_ga_sm.genome][:, 0], cities[best_ga_sm.genome][:, 1], "*")
axs[1, 0].plot(cities[best_ga_sm.genome][:, 0], cities[best_ga_sm.genome][:, 1], "--")
axs[1, 0].set_title("Tour (Alg. Genetico) 10 gens.")

# Tour NN chico
axs[1, 1].plot(cities[best_nn_sm.genome][:, 0], cities[best_nn_sm.genome][:, 1], "*")
axs[1, 1].plot(cities[best_nn_sm.genome][:, 0], cities[best_nn_sm.genome][:, 1], "--")
axs[1, 1].set_title("Tour (Hibrido) 10 gens.")

fig.tight_layout()

plt.savefig("tours.pdf")

# +
fig, axs = plt.subplots(1, 2, figsize=(8, 4))

# Mejora GA
axs[0].plot(range(len(hist_ga)), [ind.fitness for ind in hist_ga])
axs[0].set_title("Mejora por generación (Alg. Genético)")

# Mejora NN
axs[1].plot(range(len(hist_nn)), [ind.fitness for ind in hist_nn])
axs[1].set_title("Mejora por generación (Hibrido)")


fig.tight_layout()

plt.savefig("mejora.pdf")
# -


# +
plt.plot(range(len(hist_ga)), [ind.fitness for ind in hist_ga], label="Alg. Genetico")
plt.plot(range(len(hist_nn)), [ind.fitness for ind in hist_nn], label="Hibrido")
plt.title("Distancia de ruta A. G vs. Hibrido")
plt.legend()

plt.savefig("comparacion.pdf")
```

```
# -
```

Referencias

[M K19] T.A. Wheeler M. Kochenderfer. *Algorithms for Optimization*. The MIT Press, 2019.

[Tae21] J Tae. “Traveling Salesman Problem with Genetic Algorithms”. En: (2021). URL: https://jaketae.github.io/study/genetic-algorithm/.