Evaluación Técnicas Meta-heurísticas para Optimización

Alumno: Inmaculada Perea Fernández

Agosto 2017

1 Descripción del problema

El problema consiste en encontrar N polígonos con un color y nivel de transparencia de manera que se aproximen lo mejor posible a una imagen objetivo (mona-lisa-head.png)

Se pide:

1) Implementar la generación de vecinos:

- Movimiento: mover un punto de uno de los polígonos
- Color: cambiar una de las componentes del color de un polígono
- Agregar: agregar un nuevo vértice a un polígono.
- Eliminación: eliminar un vértice de un polígono.

2) Usar el VND y/o VNS

¿Qué se valorará?

- Implementación de los ejercicios mínimos
- Prueba de parámetros: tamaño de la lista tabú, número máximo de iteraciones, tolerancia, etc.
- El score de la mejor solución encontrada.
- Aplicación de alguna mejora

2 Representación del problema

- *solución* = [(polígono, color), (polígono, color)...]
- *polígono* = [vértice1, vértice2, vértice3, ...] hasta un máximo de *max_edges*
- *color* = [R, G, B, A] con *R, G, B* y *A* en el intervalo [0, 255]
- *vértice* = (X, Y) con *X* en [0, *w*], *Y* en el intervalo [0, *h*]
- *fitness* = (100 * diff / (*w* * *h* * 3 * 255)) ** 2

Donde:

- *diff*: diferencia entre la imagen objetivo y la solución candidata
- *w*: anchura de la imagen objetivo
- *h*: altura de la imagen objetivo

Por tanto se trata de un problema de minimización, ya que queremos encontrar la solución que hace más pequeña la diferencia entre ambas imágenes.

Un ejemplo de solución sería:

[([(23, 40), (34, 67), (44, 76), (21,33), (17,38)], [2, 45, 56, 98]),

([(77, 49), (34, 67), (85, 77), (21,33)], [32, 150, 86, 200]) ...]

3 Implementación

La solución implementada está basada en el código visto en clase, y que utiliza la búsqueda Tabú.

A continuación el algoritmo a grandes rasgos:

1) Agrega un polígono aleatorio (número de vértices aleatorio en posiciones aleatorias)
2) Optimización de todas las variables (agregar vértice, cambiar color, eliminar vértice, mover vértice) hasta que converge, es decir, encaja en un óptimo local o alcanza el número de iteraciones de la tolerancia (repeticiones sin mejora) que se haya configurado
3) Se introduce un nuevo polígono y se optimiza, el polígono anterior no se modifica.
4) Cuando se tienen 2 polígonos se puede usar búsqueda por vecindades, que optimiza en cada paso (en este orden) movimiento de vértices, color, agregar vértices, eliminar vértices, y vuelve a empezar. En la búsqueda por vecindades se optimizan todos los polígonos (*polygon_list=None*)
5) Al terminar da una medida de la calidad, si es peor busca otro, si es mejor introduce el polígono a la solución candidata.
6) Se genera una solución inicial de una determinada longitud, y se vuelve a realizar una nueva búsqueda tabú.

A continuación detallaremos los principales cambios realizados sobre el código base que han sido implementados para resolver el problema planteado.

3.1 Métodos para la generación de vecinos

3.1.1 move__neighbor

Este método selecciona un vértice al azar del último polígono (o un polígono al azar si *polygon_list=None*) añadido a la solución candidata, y mueve aleatoriamente dicho vértice a una nueva posición.

```
def __move__neighbor(self, cand):
    if self.polygon_list:
        i = random.choice(self.polygon_list)
    else:
        i = random.randint(0, len(cand) - 1)

    cand = copy.deepcopy(cand)

    j = random.randint(0, len(cand[i][0])-1)
    cand[i][0][j] = self.__move_point(*cand[i][0][j])
    return cand
```

El método anterior utiliza la función *move_point*, que selecciona aleatoriamente una de sus coordenadas (*X* o *Y*) del vértice que recibe como entrada, y desplaza a lo largo del eje correspondiente una cierta cantidad aleatoria en el intervalo [-10, 10] sin que supere los límites de la imagen objetivo (*w*, *h*)

```
def __move_point(self, x, y):
    offset = 10
    if random.choice([True, False]):
        x = max(min(x + random.randint(-offset, offset), self.w), 0)
    else:
        y = max(min(y + random.randint(-offset, offset), self.h), 0)

    return x, y
```

3.1.2 remove__neighbor

Este método genera un vecino por la eliminación de un vértice del polígono, y devuelve el candidato.

Si el último polígono (o un polígono al azar si *polygon_list=None*) añadido a la solución candidata tiene más de 3 vértices, selecciona uno de ellos al azar y lo elimina. Si el polígono no tiene más de 3 lados no realiza ninguna acción, ya que de lo contrario al borrar uno de los vértices el polígono se convertiría en un triángulo (problema de los triángulos), una línea o un punto.

```
def __remove__neighbor(self, cand):
    if self.polygon_list:
        i = random.choice(self.polygon_list)
    else:
        i = random.randint(0, len(cand) - 1)
    cand = copy.deepcopy(cand)

    if len(cand[i][0]) > 3:
        cand[i] = self.__remove_vertex(cand[i][0]), cand[i][1]
    return cand
```

El método anterior utiliza el método *remove_vertex,* que recibe como entrada un polígono. Se selecciona un vértice del polígono de entrada al azar y lo elimina, devolviendo el polígono resultante.

```
def __remove_vertex(self, polygon):
    polygon = copy.deepcopy(polygon)
    polygon.remove(random.choice(polygon))
    return polygon
```

3.1.3 color_neighbor

Este método genera un vecino mediante la modificación aleatoria del color del último polígono (o un polígono al azar si *polygon_list=None*) añadido a la solución candidata.

```
def __color_neighbor(self, cand):
    if self.polygon_list:
        i = random.choice(self.polygon_list)
    else:
        i = random.randint(0, len(cand) - 1)
    cand = copy.deepcopy(cand)

    cand[i] = cand[i][0], self.__perturb_color(cand[i][1])
    return cand
```

El método anterior utiliza el método *perturb_color* que dado un color de entrada, selecciona una de las componentes (RGBA) que definen el color y la modifica ligeramente (offset).

```
def __perturb_color(self, color):
    offset = 10
    i = random.randint(0, 3)
    color = copy.deepcopy(color)
    color[i] = max(min(color[i] + random.randint(-min(offset, color[i]), min(offset, 255 - color[i])), 255), 0)
    return color
```

3.1.4 add_neighbor

Este método genera un vecino añadiendo un nuevo vértice (o lado) al último polígono (o un polígono al azar si *polygon_list=None*) añadido a la solución candidata

```
def __add_neighbor(self, cand):
    if self.polygon_list:
        i = random.choice(self.polygon_list)
    else:
        i = random.randint(0, len(cand) - 1)
    cand = copy.deepcopy(cand)

    cand[i] = self.__add_vertex(cand[i][0]), cand[i][1]
    return cand
```

El método anterior llama a *add_vertex*, que recibe como entrada un polígono y tras comprobar que dicho polígono no supera el número máximo de lados (*max_edges*) selecciona aleatoriamente la posición en la que añadirá el nuevo vértice. El vértice que ha sido seleccionado aleatoriamente y el siguiente quedarán desconectados y se unirán al nuevo vértice para formar el nuevo polígono.

```
def __add_vertex(self, polygon):
    if(len(polygon) < self.max_edges):
        polygon = copy.deepcopy(polygon)
        polygon.insert(random.randint(0, len(polygon) - 1), self.get_random_point())
    return polygon
```

3.2 Obtener punto, color y polígono aleatoriamente

El método *get_random_color* obtiene un color de forma aleatoria

```
def get_random_color(self):
    return [random.randint(0, 255), random.randint(0, 255), random.randint(0, 255), random.randint(0, 255)]
```

El método *get_random_polygon* genera un polígono de más de 3 lados al azar.

```
def get_random_polygon(self):
    num_edges = random.randint(3, self.max_edges)
    return [self.get_random_point() for _ in range(num_edges)]
```

El método *get_random_point* obtiene un punto al azar que no supere los límites de la imagen objetivo.

```
def get_random_point(self):
    return random.randint(0, self.w), random.randint(0, self.h)
```

3.3 VNS / VND

Se ha modificado el bucle que busca la solución inicial para aplicar búsqueda por vecindades VND o VNS.

Por motivos de tiempo y medios disponibles para la computación, no se ha aplicado ambas búsquedas por vecindades conjuntamente. Es por esto que se ha introducido un parámetro al problema llamado *vns_vnd* que puede tomar los siguientes valores:

- *None*: no se aplica ni VNS ni VND.
- *vns*: se aplica VNS en la búsqueda de la solución inicial.
- *vnd*: se aplica vnd en la búsqueda de la solución inicial.

A continuación la parte del código que realiza la búsqueda por vecindades:

```
if i > 1:
    problem.polygon_list=None
    if problem.vns_vnd =='vnd':
        print('VND search...')
        vnd = VND(searcher, problem, ['move', 'color', 'add', 'remove'])
        vnd.search(initial_solution=initial_solution)

        problem.neighborhood = 'all'
        if current_fitness > searcher.best_fitness or initial_solution is None:
            initial_solution = searcher.best
            current_fitness = searcher.best_fitness

    elif problem.vns_vnd =='vns':
        print('VNS search...')
        vns = VNS(searcher, problem, ['move', 'color', 'add', 'remove'])
        vns.search(initial_solution)

        problem.neighborhood = 'all'
        if current_fitness > searcher.best_fitness or initial_solution is None:
            initial_solution = searcher.best
            current_fitness = searcher.best_fitness

    pass

i += 1
print("Solution length: %d" % i)
```

Notar que le hemos indicado *polygon_list=None* para que optimice todos los polígonos, y con *problema.neighborhood=all* indicamos que realice todas las transformaciones posibles para la generación de vecinos (*move*, *color*, *add*, *remove*)

Cuando finaliza la búsqueda por vecindades comparamos con el fitness actual, y si la nueva búsqueda la mejora sustituimos la lista con el candidato obtenido.

3.4 Eliminación de multiprocessing

Ha sido necesario eliminar multiprocessing de la implementación porque en Windows no funciona correctamente el método que consulta el tamaño de la cola y esto provocaba que la imagen de la solución candidata no se pintara.

```
def plot_sol(self, sol):

    if self.draw_process is None:
        self.draw_process = multiprocessing.Process(None, self.__plot_sol, args=(self.draw_queue,))
        self.draw_process.start()

    if self.draw_queue.qsize() < 1:
        self.draw_queue.put(sol)
```

3.5 Dibujar solución

Se ha utilizado la librería *openCV* para dibujar la solución y para calcular la diferencia con la imagen objetivo, tal y como se realiza en el código base del que se ha partido para la implementación de la solución propuesta.

Se ha sustituido el método plot__sol y eliminado el método *__plot__sol. E*stos métodos solo se han utilizado en depuración y una vez que el código estaba estable y probado se ha eliminado la llamada para evitar que pinte en cada mejora. En su lugar se ha implementado un nuevo método, *plot_final_sol* para pintar la solución final una vez el algoritmo de búsqueda ha finalizado, y no en cada mejora.

```
def plot_final_sol(self, sol):
    sol = self.create_image_from_sol(sol, True)
    cv2.imwrite(self.sol_file, sol)
```

De este modo ahorramos tiempo de procesado, dado que tal y como se ha comentado anteriormente, multiprocessing no funciona correctamente en sistema operativo Windows.

La llamada al método se realiza al finalizar la búsqueda (main), pasando como parámetro la mejor solución obtenida.

```
problem.finish()

pk.dump(improving_list, open(improving_file_path + '_%f.pk' % searcher.best_fitness, 'wb'))

end=time.time()

params_dict['elapsed_time']= '{} secs'.format(round(end-start, 2))
params_dict['fitness'] = round(searcher.best_fitness, 4)

problem.params_to_file(params_dict, params_file_path)

problem.plot_final_sol(searcher.best)

print("Finish")
```

3.6 Automatización de pruebas

Con el objetivo de automatizar todo lo posible las prueba, y evitar errores, se ha creado un directorio de pruebas en “test” donde existe una carpeta para cada una de las pruebas realizadas. Esta carpeta contiene:

- Fichero *png* con la imagen de la mejor solución obtenida en ese test
- Fichero *pk* con la mejor solución obtenida
- Fichero *txt* con los valores de los parámetros y medidas de error y rendimiento de dicha prueba.

Con el objetivo de automatizar la generación de los ficheros anteriores se han realizado los siguientes cambios en el código:

3.6.1 Diccionario de parámetros

En el apartado 4.1 se puede consultar el significado de cada uno de estos parámetros.

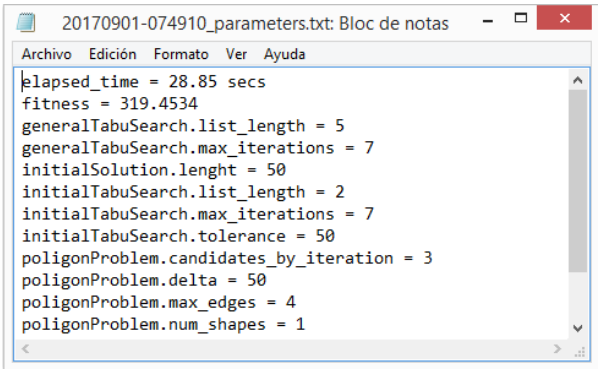
```
# PARAMS CONFIGURATION
params_dict={}

params_dict['test_number'] = '14'
params_dict['generalTabuSearch.list_length'] = 5
params_dict['generalTabuSearch.max_iterations'] = 7
params_dict['initialSolution.lenght'] = 75

params_dict['initialTabuSearch.list_length'] = 2
params_dict['initialTabuSearch.max_iterations'] = 7
params_dict['initialTabuSearch.tolerance'] = 50
params_dict['poligonProblem.candidates_by_iteration'] = 3
params_dict['poligonProblem.delta'] = 50
params_dict['poligonProblem.max_edges'] = 4
params_dict['poligonProblem.num_shapes'] = 1
params_dict['poligonProblem.vns_vnd'] = 'vnd'
```

3.6.2 Fichero texto con parámetros

Se ha creado un nuevo método *params_to_file* para volcar a un fichero de texto los parámetros de cada una de las pruebas realizadas:

```
def params_to_file(self, params_dict, file_path):
    with open(file_path, 'a') as fparams:
        for param, value in sorted(params_dict.items()):
            line = '{} = {}'.format(param, value)
            print(line, file=fparams)
```

A continuación un ejemplo de fichero de parámetros:



3.6.3 Ruta y nombre variable de los ficheros de salida

Se añade timestamp para diferenciar cada prueba.

```
params_file_path = 'tests/' + params_dict['test_number'] + '/' + datetime.now().strftime('%Y%m%d-%H%M%S') + '_parameters.txt'
solution_file_path = 'tests/' + params_dict['test_number'] + '/' + datetime.now().strftime('%Y%m%d-%H%M%S') + '_solution_file.png'
improving_file_path= 'tests/' + params_dict['test_number'] + '/' + datetime.now().strftime('%Y%m%d-%H%M%S') + '_improving_list'
```

4 Pruebas realizadas

Se han realizado numerosas pruebas variando los parámetros del algoritmo para conseguir la solución óptima.

4.1 Parámetros

Las pruebas realizadas se han hecho variando los siguientes grupos de parámetros.

4.1.1 Parámetros de la búsqueda tabú general

- *generalTabuSearch.list_length*: tamaño de la lista tabú (número de polígonos)
- *generalTabuSearch.max_iterations*: número máximo de iteraciones permitidas sin mejora (criterio de parada)

4.1.2 Parámetros del bucle para la búsqueda de la solución inicial óptima

- *initialSolution.lenght*: longitud (número de polígonos) de la solución inicial optimizada (bucle de búsqueda tabú y búsqueda por vecindades)
- *initialTabuSearch.list_length*: longitud de la solución.
- *initialTabuSearch.max_iterations*: número máximo de iteraciones
- *initialTabuSearch.tolerance:* tolerancia

4.1.3 Parámetros del problema

- *poligonProblem.candidates_by_iteration:* número de candidatos a optimizar en cada iteración.
- *poligonProblem.delta:* parámetro delta de la búsqueda tabú
- *poligonProblem.max_edges:* número máximo de lados del polígono
- *poligonProblem.num_shapes:* número de polígonos de la solución inicial obtenida aleatoriamente
- *poligonProblem.vns_vnd:* Indica si se ha aplicado búsqueda por vecindades (VNS / VND) o no

4.2 Tabla comparativa

A continuación la tabla resumen de alguna de las pruebas realizadas, ordenadas de mayor a menor fitness.

Se comenzó realizando pruebas simples con bajo tiempo de convergencia para explorar los parámetros que más influencia tienen en nuestra función objetivo o fitness, y se fue aumentando la complejidad de las pruebas.

El procedimiento que se ha seguido ha sido fijar todos los parámetros, y en cada prueba variar solamente uno de ellos para comparar con el resto de soluciones y comprobar si tiene o no influencia en la mejora de nuestra función objetivo.

Parámetro	test 01	test 02	test 03	test 04	test 05	test 06	test 07	test 08	test 09	test 10	test 11	test 12	test 13	test 14
elapsed_time (secs)	4,89	28,85	93,53	228,63	166,15	1557,75	3291,54	1665,63	1093,57	27988,23	1442,44	36170,85	6578,67	17588,98
fitness	600,50	319,45	248,84	189,01	50,83	34,85	31,79	35,87	32,35	33,80	31,47	25,38	23,71	8,57
generalTabuSearch.list_length	5	5	5	5	5	5	5	5	5	25	5	5	5	100
generalTabuSearch.max_iterations	7	7	7	7	7	7	7	7	25	7	7	7	7	100
initialSolution.lenght	25	50	50	100	25	25	50	25	25	25	25	25	25	100
initialTabuSearch.list_length	2	2	2	2	2	2	2	2	2	2	2	2	2	2
initialTabuSearch.max_iterations	7	7	7	7	7	7	7	7	7	7	7	25	7	100
initialTabuSearch.tolerance	50	50	50	50	50	50	50	50	50	50	100	50	50	50
poligonProblem.candidates_by_iteration	3	3	3	3	3	3	3	3	3	3	3	3	6	100
poligonProblem.delta	50	50	50	50	50	50	50	50	50	50	50	50	50	50
poligonProblem.max_edges	4	4	10	4	4	4	10	10	4	4	4	4	4	7
poligonProblem.num_shapes	1	1	1	1	1	1	1	1	1	1	1	1	1	1
poligonProblem.vns_vnd	None	None	None	None	vns	vnd	vns	vnd	vnd	vnd	vnd	vnd	vnd	None

Figura 1 Tabla resumen de pruebas realizadas

El parámetro *elapsed_time* mide el tiempo que ha tardado en completar la búsqueda.

Las conclusiones tras observar los resultados obtenidos en las pruebas:

- El parámetro que más influencia tiene en los resultados es la longitud de la solución inicial (initialSolution.lenght)
- Aplicar búsqueda por vecindades mejora los resultados, pero el tiempo de procesado aumenta considerablemente.
- Se obtienen mejores resultados con búsqueda por vecindades VND que VNS.
- El número de candidatos por iteración mejora el fitness si lo aumentamos, pero si la solución inicial es considerable, este parámetro no tiene mucha influencia.
- No parece tener demasiada influencia el número máximo de lados del polígono (max_edges), aunque se observa que el fitness mejora si aumentamos.
- Si aumentamos el número máximo de iteraciones también mejoramos los resultados pero aumenta el tiempo de procesado en gran medida.
- Aumentar la tolerancia también mejora ligeramente los resultados.

4.3 Pruebas abortadas

Muchas de las pruebas realizadas han sido abortadas porque precisan de gran cantidad de recursos y tiempo computación y no dispongo de un pc dedicado para poder realizar dichas pruebas. Por tanto, toda prueba que supere las 8 horas ha sido abortada.

A continuación algunas de ellas:

generalTabuSearch.list_length	5	2	5
generalTabuSearch.max_iterations	7	7	7
initialSolution.lenght	75	50	25
initialTabuSearch.list_length	2	5	2
initialTabuSearch.max_iterations	7	7	7
initialTabuSearch.tolerance	50	50	50
poligonProblem.candidates_by_iteration	3	3	25
poligonProblem.delta	50	50	50
poligonProblem.max_edges	4	4	4
poligonProblem.num_shapes	1	1	1
poligonProblem.vns_vnd	vnd	vnd	vnd

Figura 2 Tabla de pruebas abortadas

En general, se ha observado que el tiempo para que converja la búsqueda cuando usamos VND y una longitud de solución inicial mayor que 50 polígonos es muy elevado. Pero a la vista de los resultados de las otras pruebas se espera que mejore los resultados obtenidos.

5 Resultados

A continuación los resultados obtenidos en cada prueba ordenados de peor a mejor fitness:

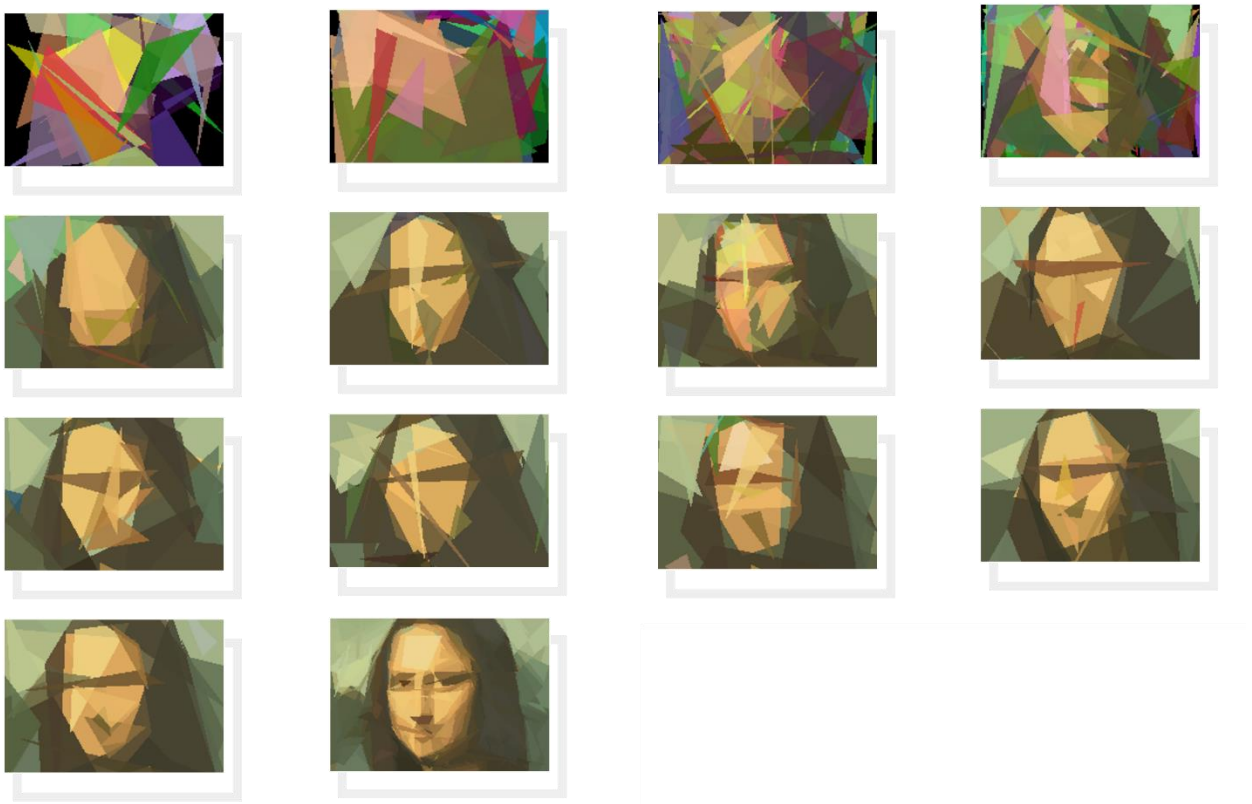


Figura 3 Resultados de las pruebas realizadas

5.1 Mejor resultado obtenido

A continuación un resumen de los parámetros con los que se ha obtenido la mejor solución, para más detalle consultar la tabla resumen de la Figura 1.

- **Fitness = 8,57**
- **Tiempo ~ 5 horas**
- Longitud de la solución objetivo = 100 polígonos
- No usa búsqueda por vecindades
- Número máximo de iteraciones = 100
- Número máximo de candidatos por iteración = 100

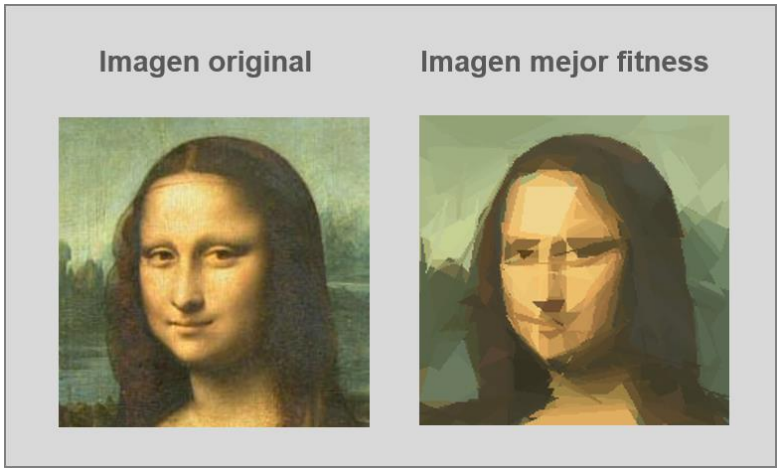


Figura 4 Resultado mejor solución

6 Propuestas de mejora

No se ha implementado en este ejercicio, pero algunas mejoras que podrían plantearse son:

- Optimizar el polígono que introduzca peor resultado, en lugar de optimizar el último que se ha introducido a la lista de candidatos.

```
while initial_solution is None or len(initial_solution) < params_dict['initialSolution.lenght']:
    problem.num_shapes = i
    # una posible mejora podria ser optimizar los polígonos que están introduciendo más error
    # i-1 optimiza el último poligono introducido
    problem.polygon_list = [i-1]

    if not initial_solution is None:
        initial_solution.append([problem.get_random_polygon(), problem.get_random_color()])
```

- Variar el offset de desplazamiento de los vértices en la búsqueda de vecinos mediante movimiento de uno de los vértices (valor por defecto 10)

- Realizar otro tipo de perturbaciones para encontrar vecinos.