INFORME SOBRE PROYECTO DE METAHEURÍSTICA. DICIEMBRE, 2017

Amelia Rabanillo Echaniz[1], Aryan Curiel Pardo[2]

1 C512, a.rabanillo@estudiantes.matcom.uh.cu

2 C512, a.curiel@estudiantes.matcom.uh.cu

1. RESUMEN

En el siguiente texto se ejemplifica la utilización de las metaheurísticas GRASP, Colonia de hormigas y Búsqueda local 2-opt como alternativas de solución para el TSP. Además, se utiliza evolución diferencial como método de optimización de funciones continuas

2. TSP

En el proyecto se emplea la versión simétrica del problema del viajante, lo cual significa que el costo de viajar de la ciudad 'A' a la ciudad 'B' es el mismo que el de viajar de la ciudad 'B' a la 'A'. En este caso se utiliza como función de costo la distancia euclidiana.

2.1 Random

Con el objetivo de obtener soluciones para comparar se implementó la clase *Random_TSP.* Esta devuelve una solución aleatoria al problema del viajante que se le pasa como parámetro.

2.2 Búsqueda local 2-opt

Como método de búsqueda local se implementó 2-opt, el cual se utiliza luego de obtener la solución *random* con el objetivo de ver su desempeño partiendo de una solución mala. Además, se usa en la fase de búsqueda local de la metaheurística GRASP y opcionalmente se puede utilizar al finalizar la metaheurística colonia de hormigas ya que este algoritmo no garantiza hallar un óptimo local.

Este método consiste en dada una solución al problema del viajante, para cada par de aristas cuyos extremos son 4 ciudades diferentes, se quitan de la solución y se colocan modificándolas de forma tal que la ruta siga siendo una solución válida del TPS (esta modificación se puede hacer de una única forma posible). Lo anterior se encuentra implementado en el archivo *TSP_heuristic.py*, como el método *local_search* de la clase *TSP_heuristic*.

2.3 GRASP

La implementación de GRASP consiste en realizar iterativamente las fases de construcción y búsqueda local una cantidad de veces determinadas por un parámetro, manteniendo la mejor solución y el mejor costo obtenidos con el fin de retornarlos al final del algoritmo.

2.3.1 Parámetros

Luego de probar con varias combinaciones de parámetros nos dio un resultado aceptable el siguiente conjunto de los mismos:

La cardinalidad de la lista de candidatos restringida la establecimos como la parte entera de ¼ de la cantidad de ciudades.

Luego de imprimir la iteración en la cual se encontraba la mejor solución notamos que esta podía hallarse en casi cualquier iteración, sin preferencia por ningún intervalo de éstas. Por tanto, decidimos que lo que limitaba la cantidad de iteraciones era el tiempo y establecimos un número de 50.

2.3.2 Fase de construcción

Esta fase selecciona iterativamente una arista de la RCL y la adiciona a la solución si no crea infactibilidad, o sea, si no crea un ciclo y si no une una ciudad a la que el viajante ya llegó y salió.

Para esta fase se usa una cola con prioridad que permita determinar que aristas se debe agregar a la lista de candidatos restringida.

Para evitar que el *tour* pierda la característica de

camino se mantiene un contador por ciudades de cuantas aristas las contienen, y se mantiene la restricción de que para ninguna ciudad este contador puede ser mayor que 2. Además, cada vez que se agrega una arista se verifica usando DFS que no exista un ciclo.

Esta fase se detiene cuando, dadas n ciudades el camino tiene n-1 aristas donde todos los vértices tienen grado 2, excepto 2 vértices con grado 1; y no existe ningún ciclo entre las aristas existentes.

Dado una secuencia de *n* ciudades se asume que existe la arista que une a la primera ciudad con la última y se tiene en cuenta a la hora de calcular el costo del *tour*.

2.3.3 Búsqueda local

Esta fase recibe la solución devuelta por la fase de construcción y le realiza búsqueda local 2-opt ya explicada anteriormente.

2.4 Colonia de hormigas

Para la implementación esta metaheurística se siguió la idea explicada en [2], donde aplican Colonia de hormigas al problema del viajante.

2.4.1 Parámetros

Para la determinación de los parámetros se consultó [4] y se utilizaron, para obtener los resultados expuestos en este texto, los siguientes valores de los mismos:

-$ants = 10$

-$\alpha = 2$

-$\beta = 3$

-$\rho = 0.6$

-$iterations = 50$

-$do\_local\_search = True$

2.4.2 Búsqueda local

Las soluciones obtenidas mediante colonia de hormigas no constituyen óptimos locales por lo cual pueden ser mejoradas si a partir de ellas se aplica búsqueda local. Para esto se utiliza 2-opt, la cual ya estaba implementada como requerimiento de la fase de búsqueda local de GRASP.

La clase *Ant_colony_TSP*, que se encuentra en el archivo del mismo nombre, recibe un parámetro *do_local_search*, en el que se le puede especificar si hacer o no búsqueda local luego de que el algoritmo colonia de hormigas termina.

Como ejemplo particular la Fig.1 representa una solución de colonia de hormigas sin aplicarle búsqueda local. Este tour tiene un costo de 7462.49318565.

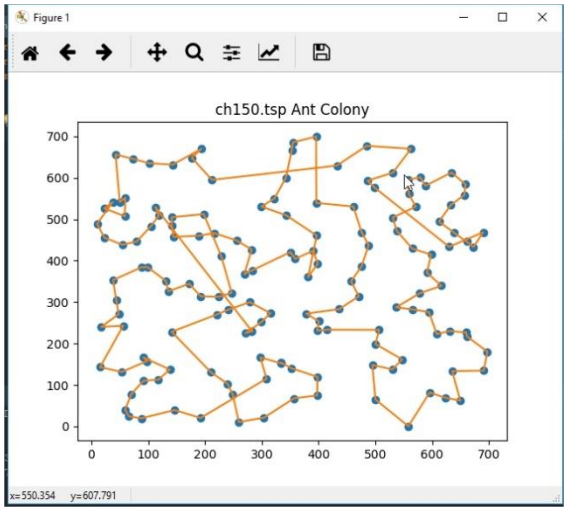


Fig.1

La Fig.2 es el resultado de aplicarle búsqueda local a la solución graficada en la Fig.1. Esta tiene un costo de 7117.98796347

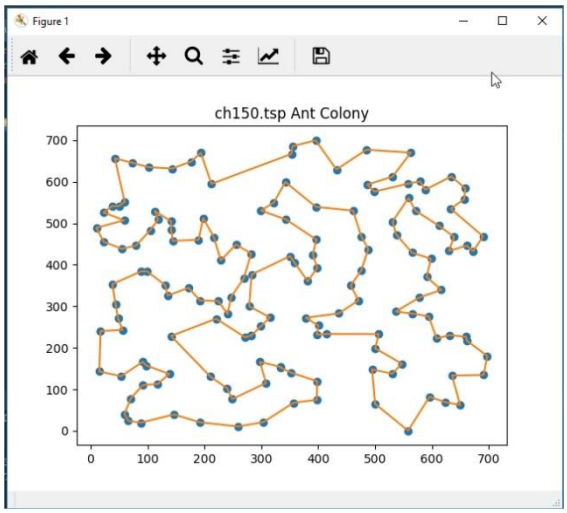


Fig.2

2.5 Resultados TSP

En el proyecto se encuentran casos de prueba descargados de [3], los cuales contaban con la solución óptima, por lo cual se puede realizar una comparación entre el valor retornado por las implementaciones y el óptimo. En la Fig.3 se muestran los datos de la ejecución de los algoritmos implementados para cada uno de los casos de prueba que se encuentran en la carpeta *test_TSP,* Los archivos que contienen los resultados se encuentran en la carpeta *Results*. Los gráficos que corresponden a los *tours* resultado se encuentran en los anexos.

```
***Testing TSP***
    file: test_TSP\a280.tsp
        Random solution cost:       34026.7165902
        Local search 2opt:          2940.9331274
        Ant colony solution cost:   3134.22316625
        GRASP solution cost:        2686.12139307 iter: 39
        Optimum cost:               2587.83190531
==================================================
    file: test_TSP\berlin52.tsp
        Random solution cost:       26502.3386703
        Local search 2opt:          8458.44797931
        Ant colony solution cost:   8006.59169292
        GRASP solution cost:        7661.01327213 iter: 0
        Optimum cost:               7544.3659019
==================================================
    file: test_TSP\ch150.tsp
        Random solution cost:       54161.1902829
        Local search 2opt:          7359.46878703
        Ant colony solution cost:   7154.85632057
        GRASP solution cost:        6702.95434912 iter: 47
        Optimum cost:               6532.28093315
==================================================
    file: test_TSP\pr76.tsp
        Random solution cost:       602376.123554
        Local search 2opt:          113390.249755
        Ant colony solution cost:   115934.364832
        GRASP solution cost:        109530.397289 iter: 36
        Optimum cost:               108159.438274
==================================================
    file: test_TSP\rd100.tsp
        Random solution cost:       58758.3464124
        Local search 2opt:          8514.4592245
        Ant colony solution cost:   9278.07416983
        GRASP solution cost:        8021.88965036 iter: 22
        Optimum cost:               7910.39621022
==================================================
    file: test_TSP\st70.tsp
        Random solution cost:       3557.36055837
        Local search 2opt:          748.980020519
        Ant colony solution cost:   762.148551043
        GRASP solution cost:        688.333188367 iter: 2
        Optimum cost:               678.597452097
==================================================
```

Fig.3

El algoritmo con mejor desempeño fue GRASP. Luego colonia de hormigas y por último búsqueda local. Una desventaja de colonia de hormigas es que es computacionalmente más costoso que GRASP, y que necesita más iteraciones para converger.

Se probó con 150 iteraciones con casos de pruebas con menos de 100 ciudades porque para más ciudades la demora del algoritmo era significativa. Los restantes parámetros se mantienen iguales. Los resultados obtenidos se muestran en la Fig.4

```
***Testing TSP***
    file: test_TSP\berlin52.tsp
        Random solution cost:       31669.2088601
        Local search 2opt:          8315.38059489
        Ant colony solution cost:   7602.11272777
        GRASP solution cost:        7661.01327213 iter: 6
        Optimum cost:               7544.3659019
==================================================
    file: test_TSP\pr76.tsp
        Random solution cost:       575082.185188
        Local search 2opt:          112972.536785
        Ant colony solution cost:   113457.093115
        GRASP solution cost:        109535.639678 iter: 43
        Optimum cost:               108159.438274
==================================================
    file: test_TSP\rd100.tsp
        Random solution cost:       56100.2975079
        Local search 2opt:          8823.65518192
        Ant colony solution cost:   9266.66837958
        GRASP solution cost:        8015.93786807 iter: 8
        Optimum cost:               7910.39621022
==================================================
    file: test_TSP\st70.tsp
        Random solution cost:       3500.9818078
        Local search 2opt:          710.766543424
        Ant colony solution cost:   747.689339326
        GRASP solution cost:        686.853420532 iter: 31
        Optimum cost:               678.597452097
==================================================
```

Fig.4

3. PROBLEMAS CONTINUOS

3.1 Evolución Diferencial

Para la resolución de los problemas continuos se implementó la metaheurística evolución diferencial.

3.1.1 Parámetros

El conjunto de parámetros con el que se obtuvo resultados aceptables fueron:

-elementos en la población: $k = 60$

-factor de escalado: $f = 0.4$

-probabilidad: $cr = 0.9$

-generaciones: $generations\_number = 600$

-valor mínimo por variable: $x\_low = -100$

-valor máximo por variable: $x\_high = 100$

-dimensión del vector resultado: $d = 30$

3.1.2 Operador de recombinación

Este operador se encuentra implementado como el método *recombination_operator* de la clase *Differential_evolution*. Recibe 4 enteros diferentes que corresponden a un padre y a tres elementos de la población que serán mezclados. El operador consiste en generar cada dimensión de la solución mediante la fórmula $u_{ij} = x_{r_3 j} + f(x_{r_1 j} - x_{r_2 j})$. Una de las dimensiones del nuevo vector (escogida de forma aleatoria), será tomada del elemento padre.

La implementación siguió el siguiente *template* tomado de [5]

Input: Parent i, three randomly selected individuals $r_1, r_2, r_3, i \neq r_1 \neq r_2 \neq r_3$.
$j_{rand} = int(rand_i[0, 1].D) + 1$;
For $(j = 1, j \leq D, j++)$ **Do**
  If $(rand_j[0, 1]) < CR)$ or $(j = j_{rand})$ **Then**
    $u_{ij} = v_{ij} = x_{r3j} + F.(x_{r1j} - x_{r2j})$;
  Else
    $u_{ij} = x_{ij}$;
Output: Offspring u_i.

En este operador se tiene en cuenta el rango en que debe estar cada variable. Si al aplicar la fórmula el valor obtenido no se encuentra en el rango definido en los parámetros de la clase, se arregla siguiendo las siguientes condiciones, tomadas de [6]

$$u_{ij}(t+1) = \begin{cases} \dfrac{x_{ij}(t) + x_j^{lo}}{2} & si\ u_{ij}(t+1) < x_j^{lo} \\ \dfrac{x_{ij}(t) + x_j^{hi}}{2} & si\ u_{ij}(t+1) > x_j^{hi} \\ u_{ij}(t+1) & en\ otro\ caso \end{cases}$$

3.1.3 Algoritmo

La población es inicializada de forma aleatoria donde cada componente de cada vector es generado utilizando una distribución uniforme entre el intervalo en que se debe encontrar cada variable

Por cada iteración del algoritmo, se itera por cada elemento de la población que será el padre, se escogen otros tres elementos de forma aleatoria y se aplica el operador de recombinación. Si el resultado obtenido es mejor que el padre, este es sustituido.

Luego de crear todas las generaciones correspondientes se devuelve el elemento de la población con menor evaluación de la función objetivo

3.1.4 Resultados

El algoritmo fue probado con las siguientes 3 funciones.

$$f(x) = \sum_{i=1}^{n} x_i^2 \qquad \text{(I)}$$

$$f(x) = \prod_{i=1}^{n} sen(x_i^2) + \sum_{i=1}^{n} (x_i^2 - 10\cos(2\pi x_i)) \qquad \text{(II)}$$

$$f(x) = \prod_{i=1}^{n} \sum_{i=1}^{n} [x_i^2 - 10\cos(2\pi x_i)] + \sum_{i=1}^{n} [cos(2\pi x_i)] \quad \text{(III)}$$

Aproximar el óptimo de las funciones (II) y (III) es objetivo del proyecto. La función (I) se tuvo en cuenta debido a que es muy simple y se conoce su óptimo que es el vector nulo del espacio.

3.1.4.1 Función I

Los parámetros se mantuvieron invariantes excepto la cantidad de iteraciones que fue fijada en 100 y la dimensión de la solución a la que se le dio valor 10.

La solución y el costo devueltos por el algoritmo se aprecian en la Fig.4. La Fig.5 muestra en el eje *x* las iteraciones y en el eje *y* el costo de la mejor solución obtenida en esa iteración.

```
Best solution:
0.0325235836322
-0.0178003153427
0.00559600523487
-0.0167593682039
0.0215786584642
-0.0282288690969
0.00370929201309
0.00244233436939
-0.0139492074473
0.00718984283585
Cost:
0.00321533204019
```

Fig.4

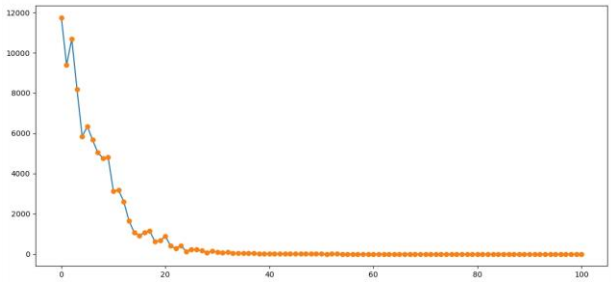


Fig.5

3.1.4.2 Función II

Para la aproximación al óptimo de esta función los parámetros se mantienen como se establecieron en 3.1.1. Obteniendo un resultado mostrado en la Fig.6 y con un historial a lo largo de las generaciones resumido en la Fig.7

```
Best solution:
-0.0845920796908    0.975855955148      -1.0265252312
-1.02526895385      0.455476657825      0.063414402642
0.184440658106      0.0613794462038     -0.0213254982822
-1.56072968908      -0.803091803684     -0.0116453637076
0.022075671027      -0.229593853646     2.2299893774
-0.894410662997     0.796763973847      -0.00694036555869
0.866337955104      0.0774966843755     0.0555798515723
0.130359742714      -0.130210498521     1.88121661327
-1.11065564769      -0.820266260453     -1.08633276216
-2.06890915173      -1.26594231614      -0.0877355841348
Cost:
-155.058167354
```

Fig.6

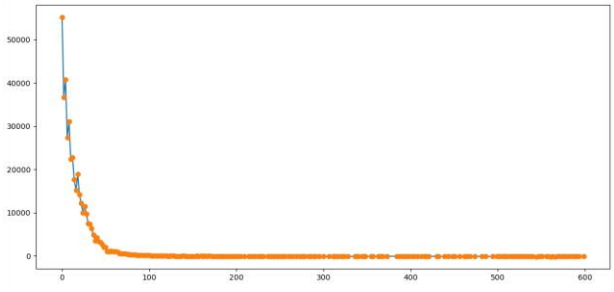


Fig.7

3.1.4.3 Función III

Para esta función los parámetros se mantienen invariantes y se obtiene el vector solución que se muestra en la Fig.8 y el historial que se muestra en la Fig.9

```
Best solution:
87.1791301165       93.4067592286       -99.645095308
-96.9919126526      89.7797967923       97.0018357051
85.3952376288       88.3171366179       98.1499487194
97.6704642174       -93.630317225       92.0598027458
96.1548693541       0.923385347762      97.8872856916
-95.0076721962      89.5097730774       -82.3853884386
76.035616945        95.5814147801       -93.8215995376
-83.9066752837      -97.4911049447      98.8639907437
73.8652826872       -94.0082366761      -96.5113465561
-99.7353670205      -98.9020237064      93.2746070433
Cost:
-1.50645935455e+159
```

Fig.8

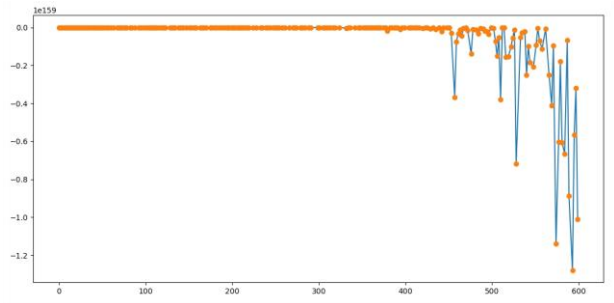


Fig.9

4. CONCLUSIONES

Luego de la implementación de las metaheurísticas podemos concluir que aun sin encontrar el óptimo las soluciones al problema del viajante se pueden considerar bastante buenas.

En cuanto a los problemas continuos, no contábamos con el óptimo de las funciones, por lo que no pudimos comparar. Solo podemos afirmar que al graficar las mejores soluciones por generación el algoritmo parece decrementar paulatinamente el costo de la función objetivo. En cuanto a la función (I) de la que si se tenía el óptimo, las soluciones se acercaban bastante al vector nulo.

5. REFERENCIAS BIBLIOGRÁFICAS

[2] Talbi, E. Metaheuristics, From Design to Implementation, pp. 244

[3] http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/, noviembre/2017

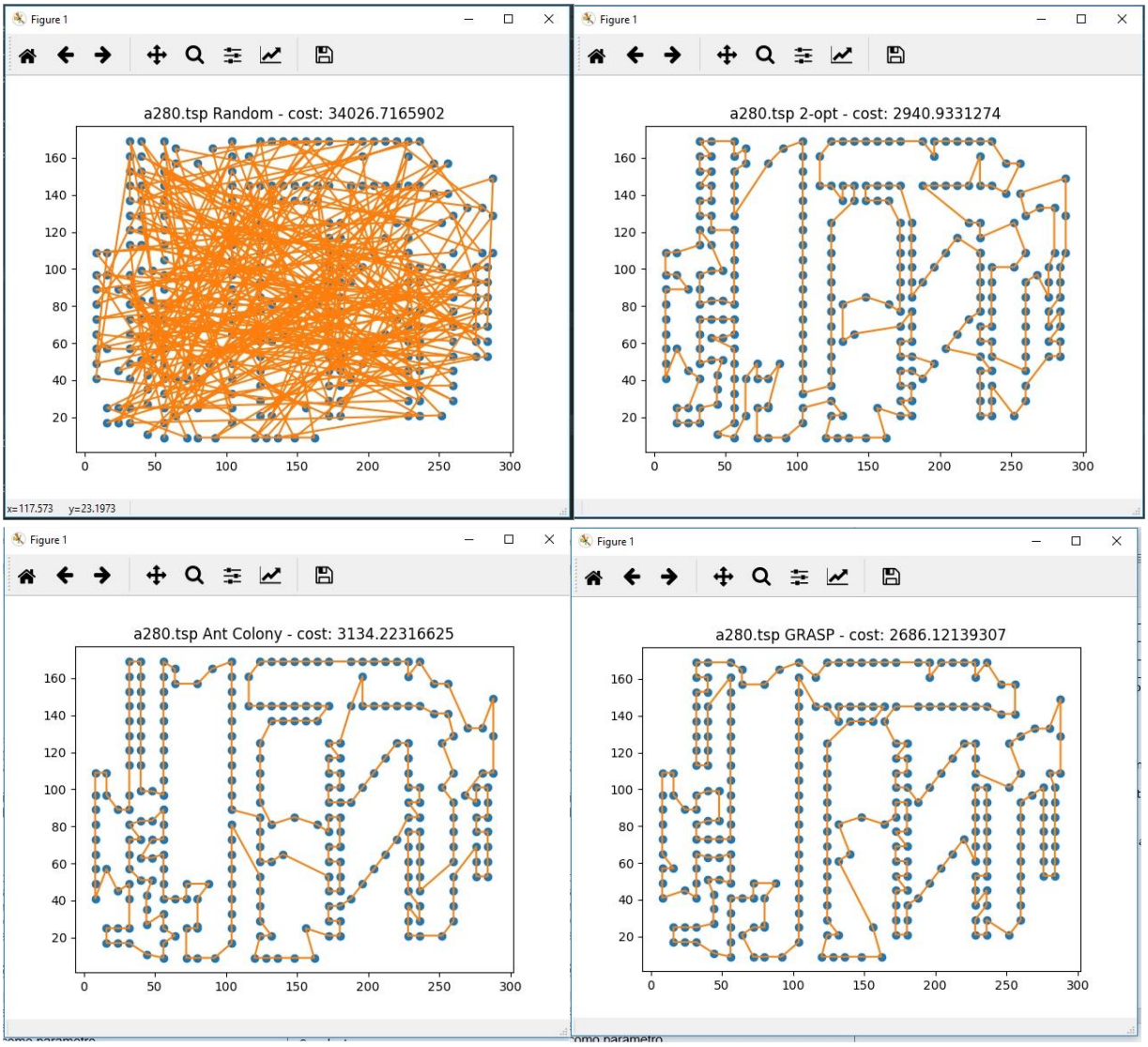
[4] Chen, J. Parameters Evaluation of Ant Colony Algorithm based on TSP.

[5] Talbi, E. Metaheuristics, From Design to Implementation, pp. 226

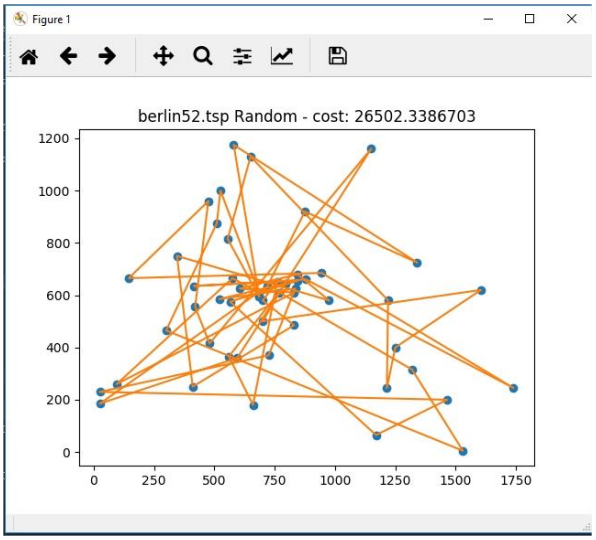
[6] Talbi, E. Metaheuristics, From Design to Implementation, pp. 228

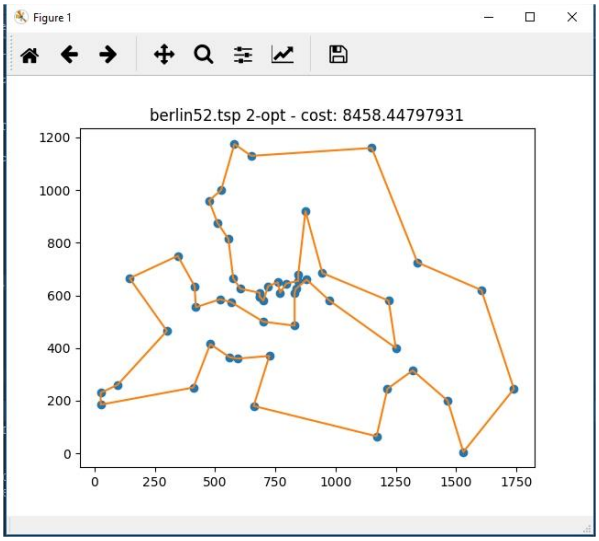
6. ANEXOS

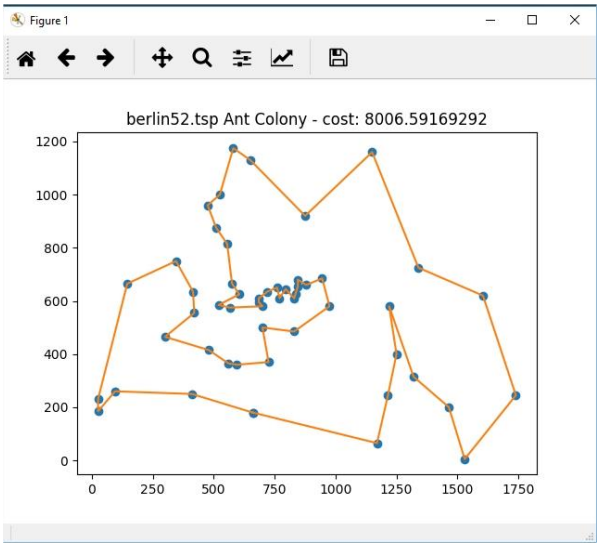
- a280.tsp

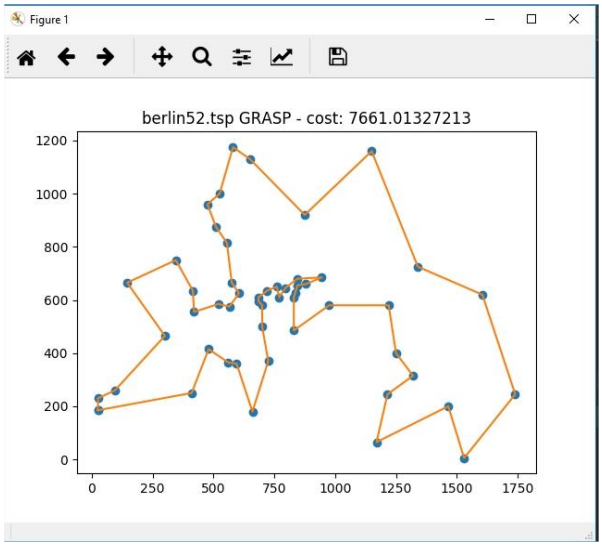


- berlin52.tsp









- ch150.tsp

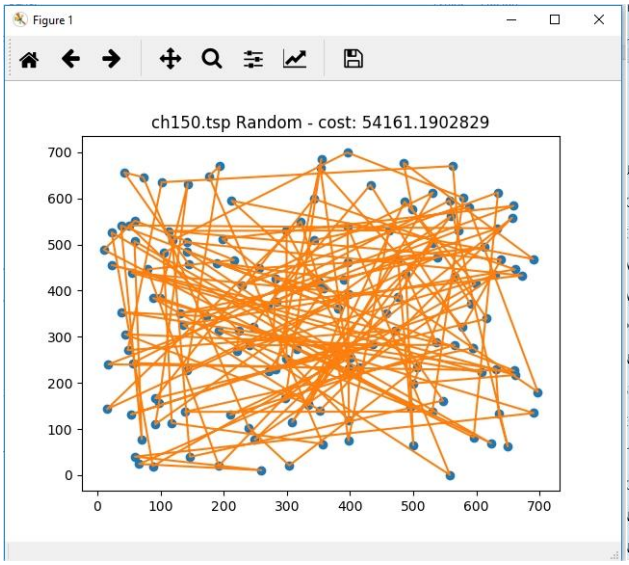


Figure 1
ch150.tsp Random - cost: 54161.1902829

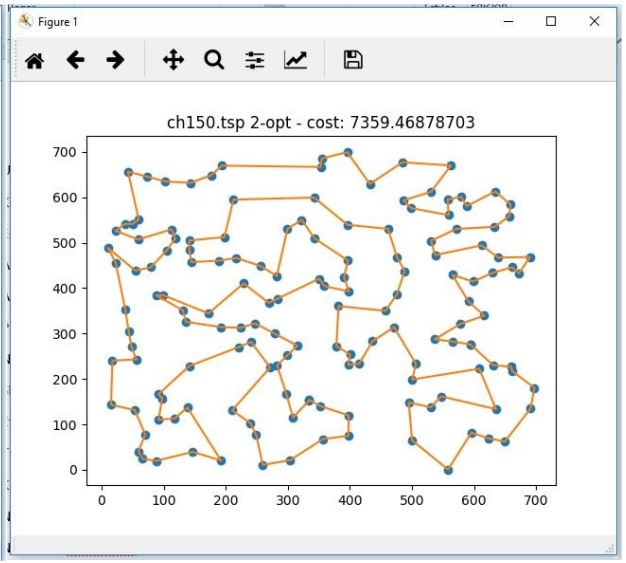


Figure 1
ch150.tsp 2-opt - cost: 7359.46878703

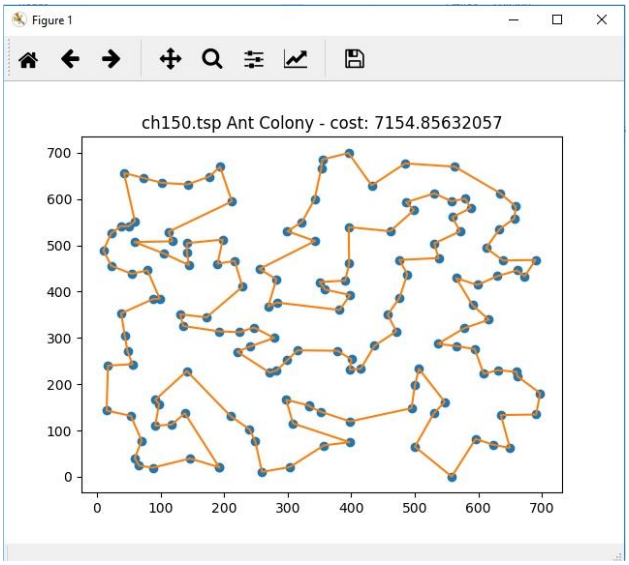


Figure 1
ch150.tsp Ant Colony - cost: 7154.85632057

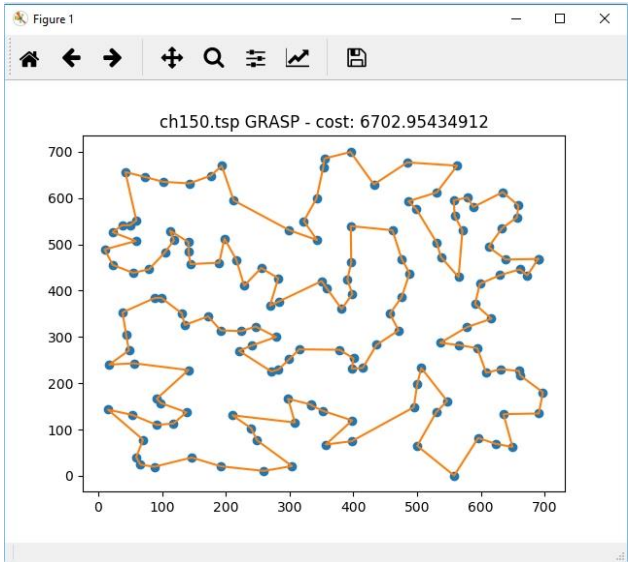


Figure 1
ch150.tsp GRASP - cost: 6702.95434912

- pr76.tsp

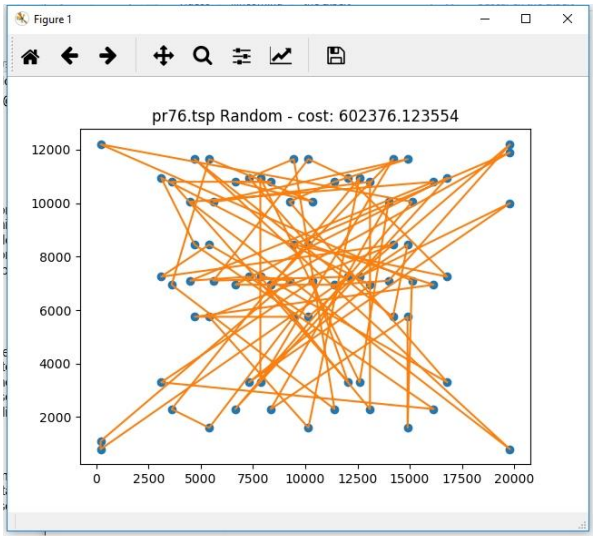


Figure 1
pr76.tsp Random - cost: 602376.123554
12000
10000
8000
6000
4000
2000
0
2500
5000
7500
10000
12500
15000
17500
20000

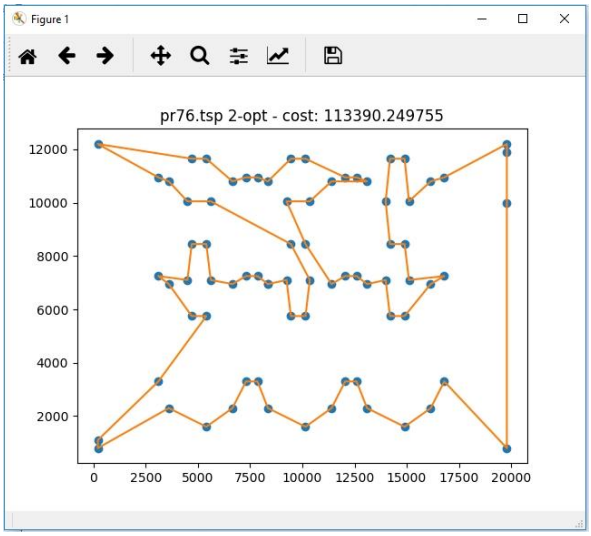


Figure 1
pr76.tsp 2-opt - cost: 113390.249755
12000
10000
8000
6000
4000
2000
0
2500
5000
7500
10000
12500
15000
17500
20000

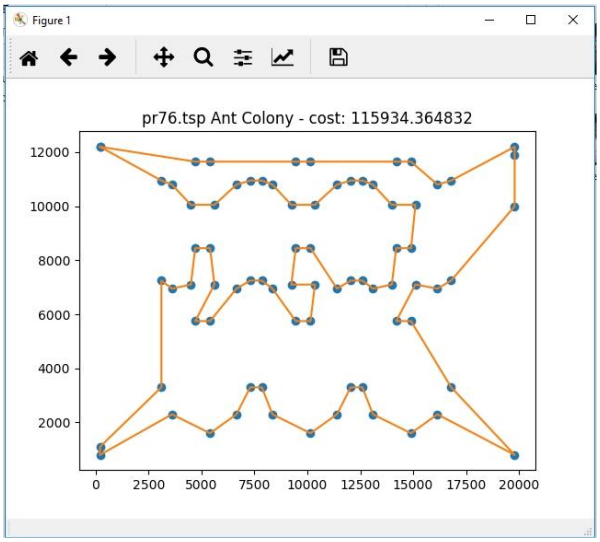


Figure 1
pr76.tsp Ant Colony - cost: 115934.364832
12000
10000
8000
6000
4000
2000
0
2500
5000
7500
10000
12500
15000
17500
20000

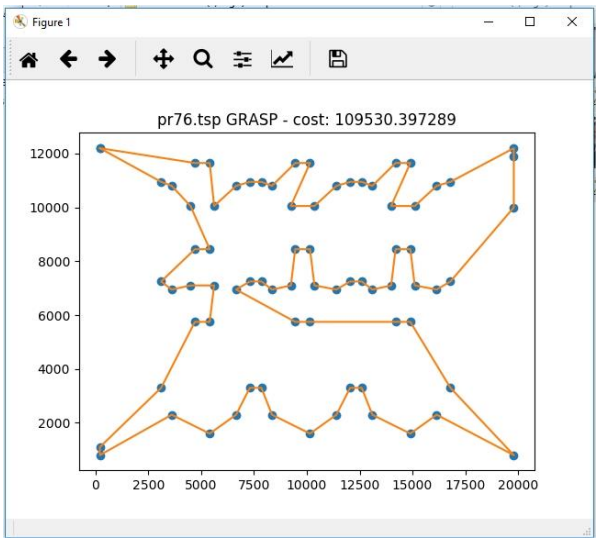


Figure 1
pr76.tsp GRASP - cost: 109530.397289
12000
10000
8000
6000
4000
2000
0
2500
5000
7500
10000
12500
15000
17500
20000

- rd100.tsp

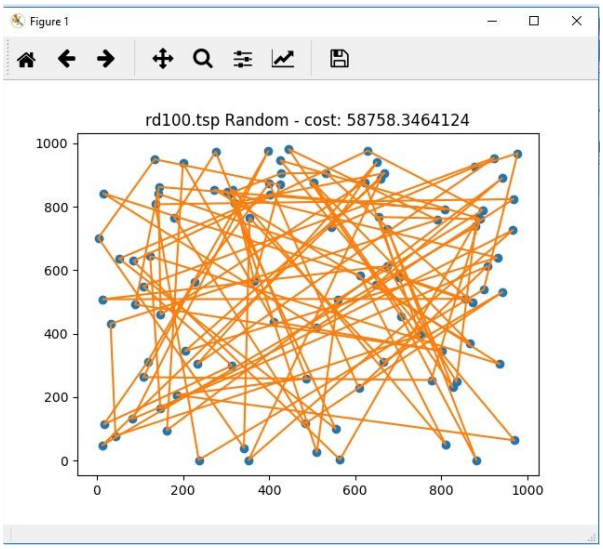


Figure 1
rd100.tsp Random - cost: 58758.3464124
0
200
400
600
800
1000

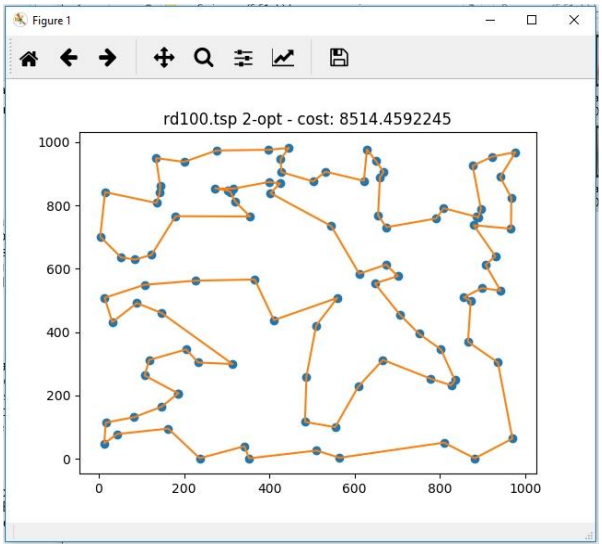


Figure 1
rd100.tsp 2-opt - cost: 8514.4592245
0
200
400
600
800
1000

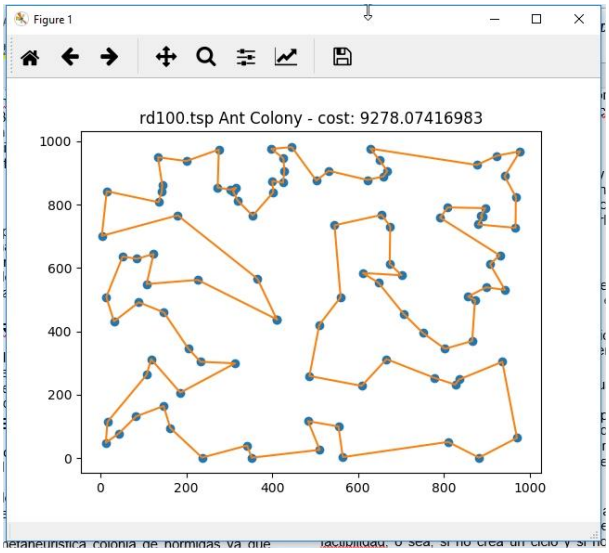


Figure 1
rd100.tsp Ant Colony - cost: 9278.07416983
0
200
400
600
800
1000

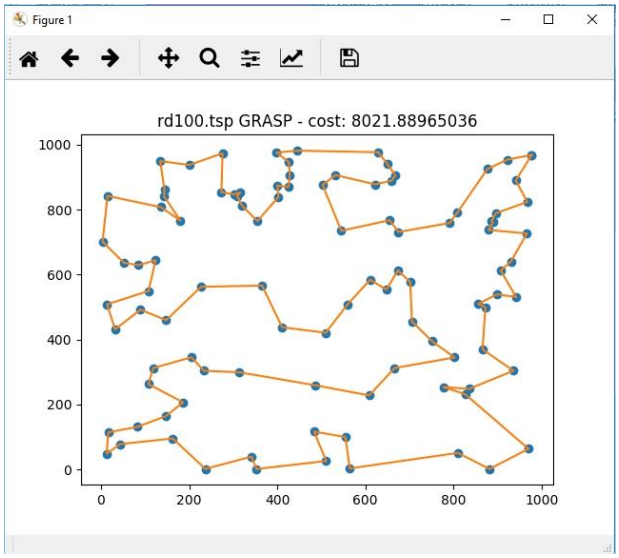


Figure 1
rd100.tsp GRASP - cost: 8021.88965036
0
200
400
600
800
1000

- st70.tsp

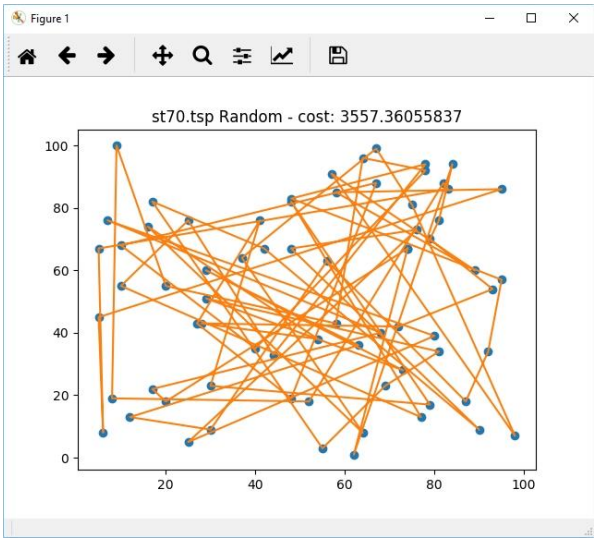


Figure 1
st70.tsp Random - cost: 3557.36055837

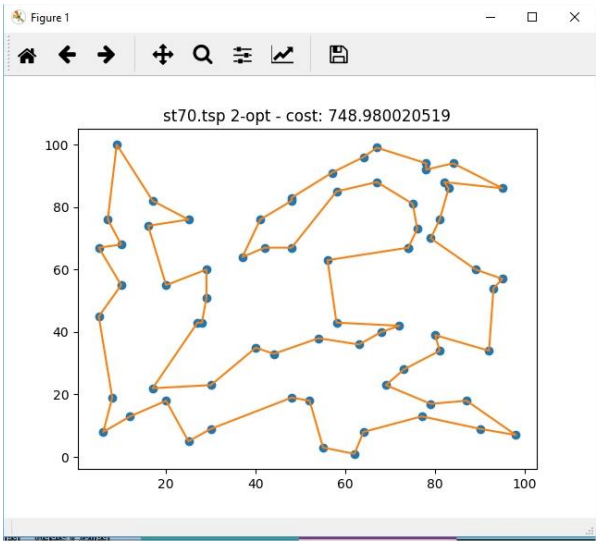


Figure 1
st70.tsp 2-opt - cost: 748.980020519

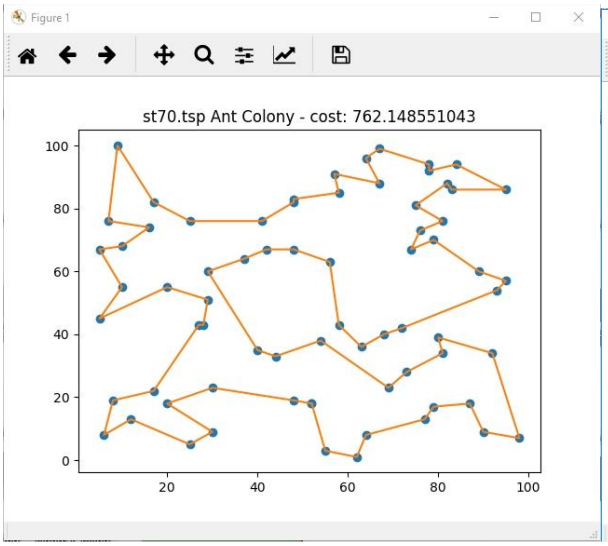


Figure 1
st70.tsp Ant Colony - cost: 762.148551043

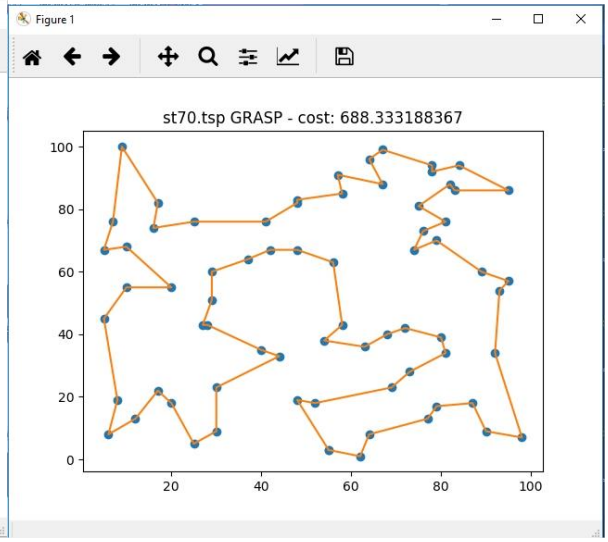


Figure 1
st70.tsp GRASP - cost: 688.333188367