

Manual TSP-Framework

Índice

1	Conociendo la interfaz	4
1.1	Guardar y cargar soluciones anteriores en modo GUI	6
2	Distribución del código y los paquetes	8
2.1	Paquete General.....	8
2.2	Sub-Paquete de Algoritmos	9
2.3	Sub-Paquete de Herramientas	9
3	Como agregar un nuevo método de búsqueda	10
3.1	Agregar las opciones	10
3.2	Crear el módulo.....	12
3.3	Agregar a modulo principal	16
3.4	Implementar en la Interfaz Gráfica	18
4	Componentes algorítmicos	22
4.1	Componentes algorítmicos manejar la población de Algoritmo Genético	22
4.1.1	Como añadir componentes algorítmicos a Algoritmo Genético	22
4.2	Componentes algorítmicos para manejar el Tour.....	24
4.2.1	Como añadir componentes algorítmicos a Tour	24

Figuras

Figura 1.1: Base Interfaz	4
Figura 1.2: Selección de algoritmo	4
Figura 1.3: Vista Simulated Annealing	5
Figura 1.4: Vista Algoritmo Genético	5
Figura 1.5: Vista Local Search	6
Figura 1.6: Vista Iterated Local Search	6
Figura 1.7: Guardar/Cargar configuración	7
Figura 2.1: Distribución de los paquetes	8
Figura 3.1: enum metaheurísticas	10
Figura 3.2: Definir Argumentos	11
Figura 3.3: Arg. nueva metaheurística	11
Figura 3.4: Procesar Argumentos	12
Figura 3.5: Constructor de clase	13
Figura 3.6: Guardar Trayectoria	13
Figura 3.7: métodos de archivos	14
Figura 3.8: método para visualizar	14
Figura 3.9: Actualizar log	15
Figura 3.10: Mostrar mejor solución	15
Figura 3.11: Imports __init__.py	16
Figura 3.12: importar método en modulo principal	16
Figura 3.13: Crear instancia del algoritmo	17
Figura 3.14: Nombre de la instancia	17
Figura 3.15: Agregar nuevo algoritmo GUI	18
Figura 3.16: Método del algoritmo GUI	19
Figura 3.17: Agregar ejecución GUI	20
Figura 3.18: Cambio de algoritmo GUI	20
Figura 3.19: Gestionar cambio de algoritmo GUI	21
Figura 4.1: Agregar un componente algoritmo genético	22
Figura 4.2: Argumentos nuevo componente algoritmo genetico	23
Figura 4.3: Importar en paquete principal	23
Figura 4.4: Importar en Population.py	23
Figura 4.5: Gestionar nuevo componente de cruzamiento	24
Figura 4.6: Tipos de movimiento TSP	25
Figura 4.7: Gestionar nuevo movimiento TSP	25
Figura 4.8: Movimiento TSP aleatorio	25

1 Conociendo la interfaz

La interfaz cuenta con las siguientes pantallas y funcionalidades:

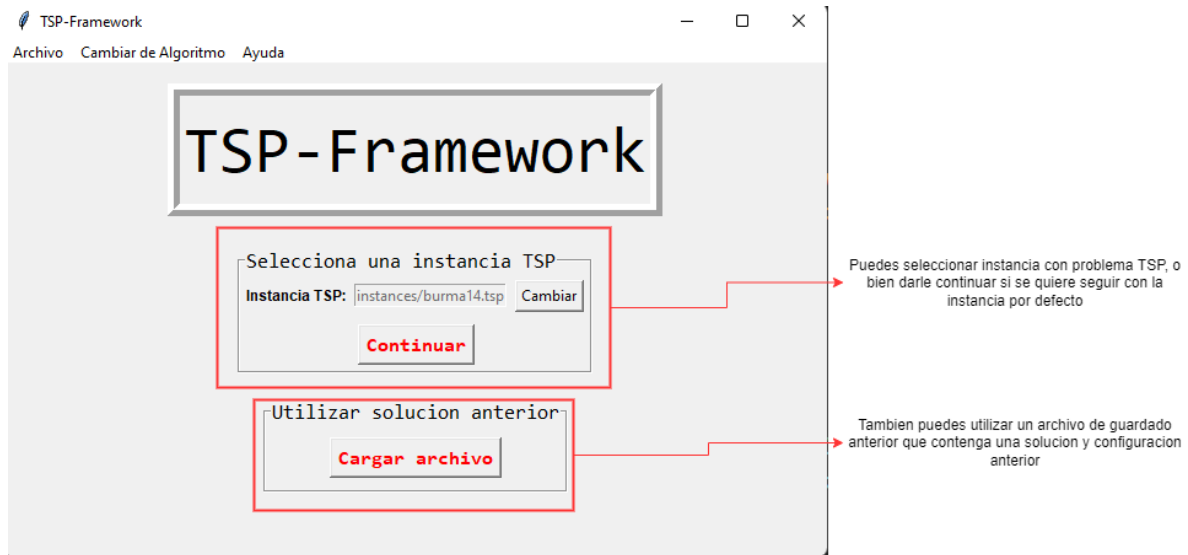


Figura 1.1: Base Interfaz

Si seleccionamos continuar, entonces debemos elegir el método de búsqueda.

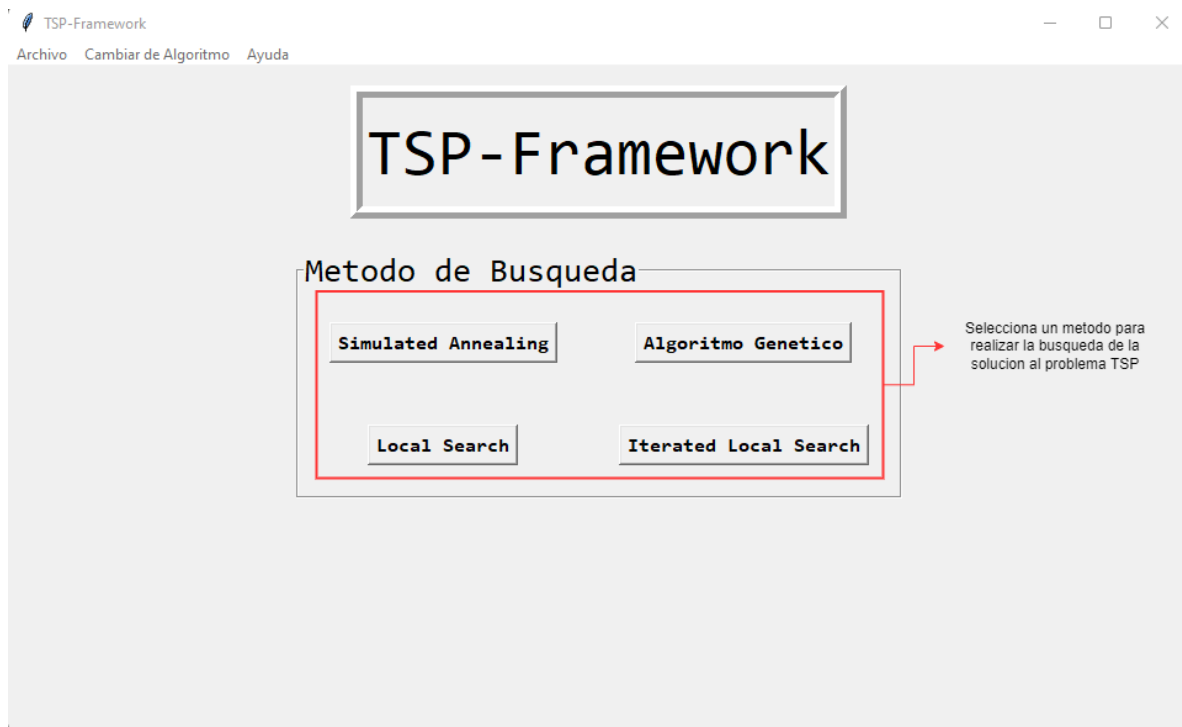


Figura 1.2: Selección de algoritmo

Luego vemos las vistas de los distintos métodos de búsqueda:

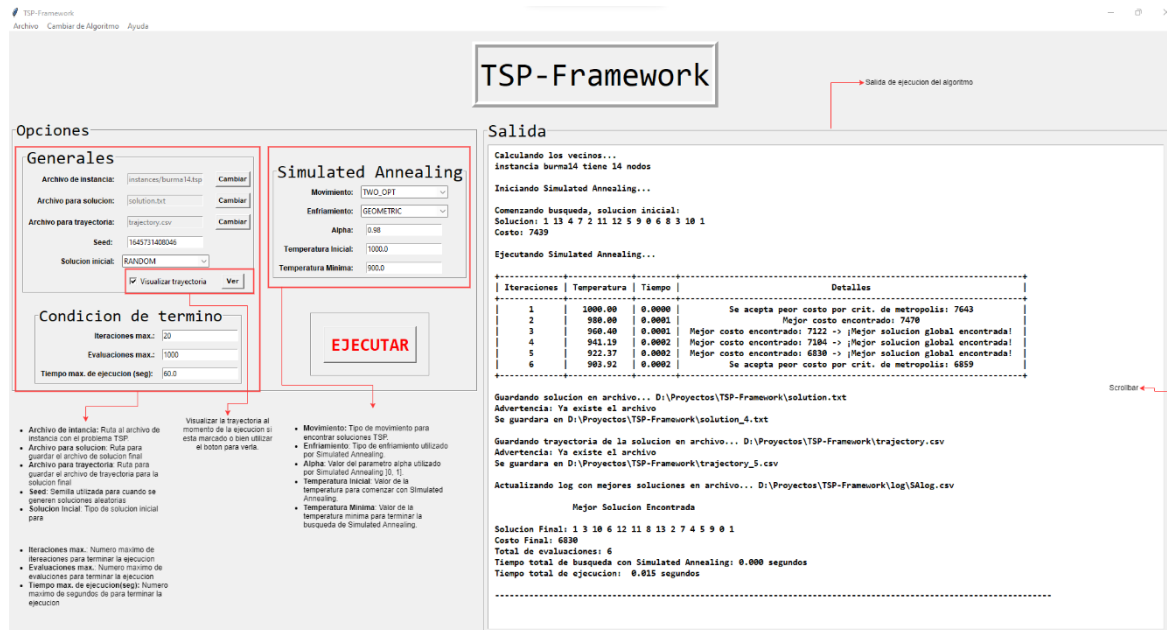


Figura 1.3: Vista Simulated Annealing

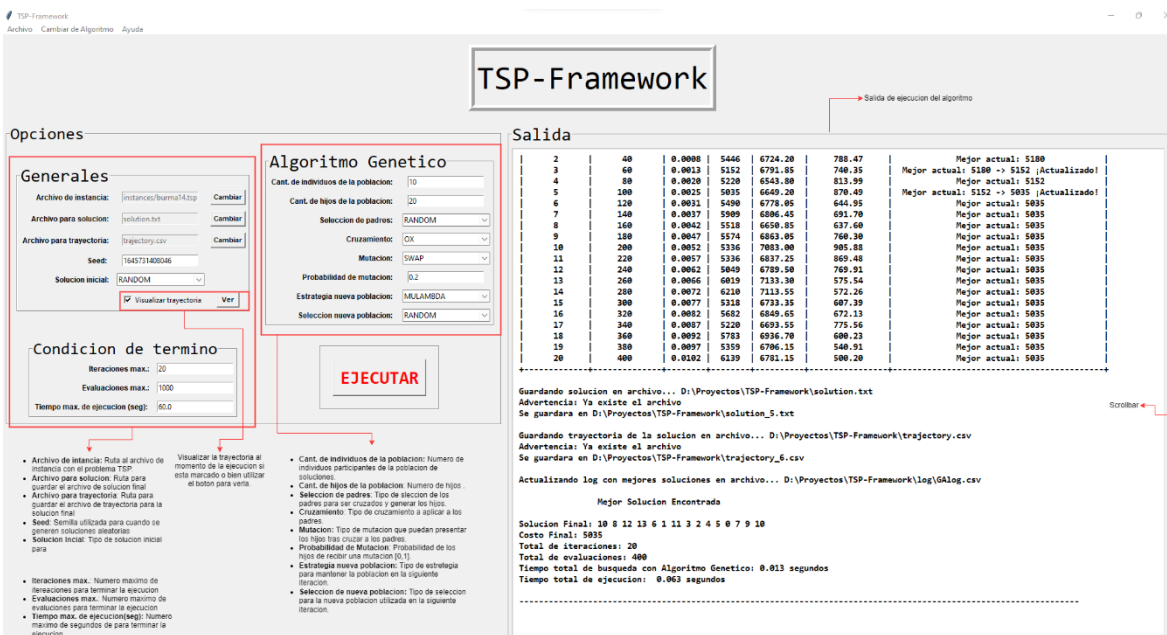


Figura 1.4: Vista Algoritmo Genético

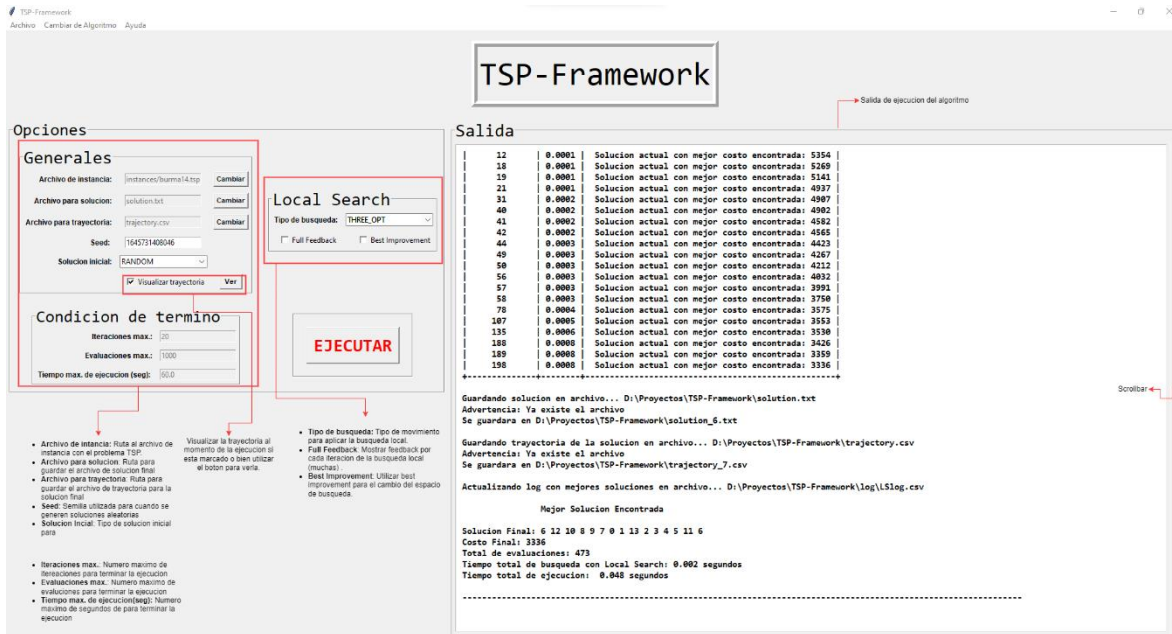


Figura 1.5: Vista Local Search

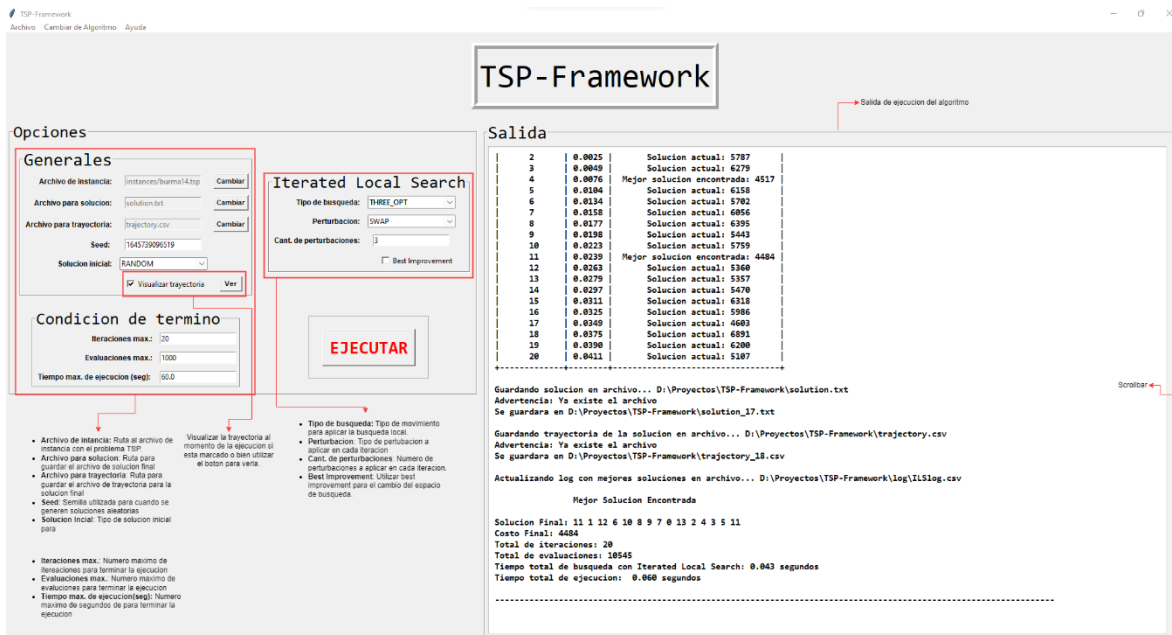


Figura 1.6: Vista Iterated Local Search

1.1 Guardar y cargar soluciones anteriores en modo GUI

Una vez ejecutado algún método de búsqueda obteniendo una solución al problema, es posible guardar la configuración completa de esa ejecución, incluyendo su salida y trayectoria dentro de un

archivo de extensión “.tspf”. Podemos cargar esta clase de archivos desde este mismo menú o al momento de ejecutar en modo GUI en la pantalla principal.

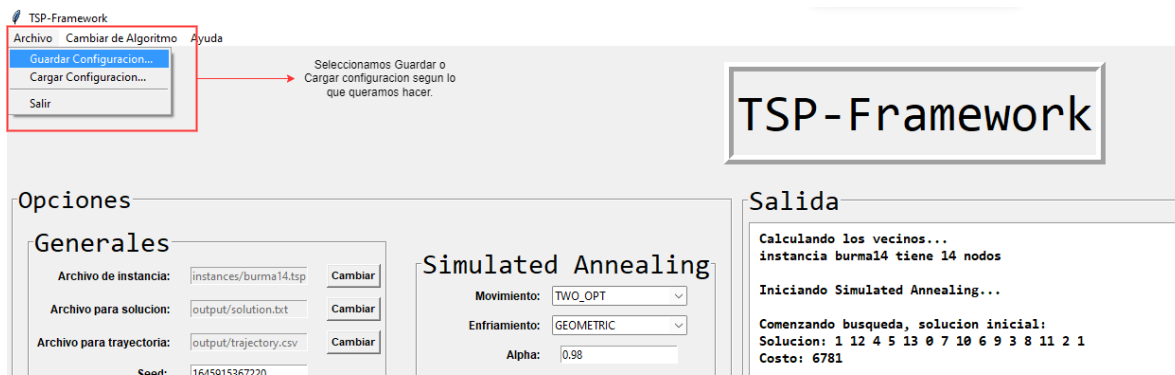


Figura 1.7: Guardar/Cargar configuración

2 Distribución del código y los paquetes

La distribución interna del código fuente está compuesto por diferentes paquetes como se muestra en la siguiente imagen:

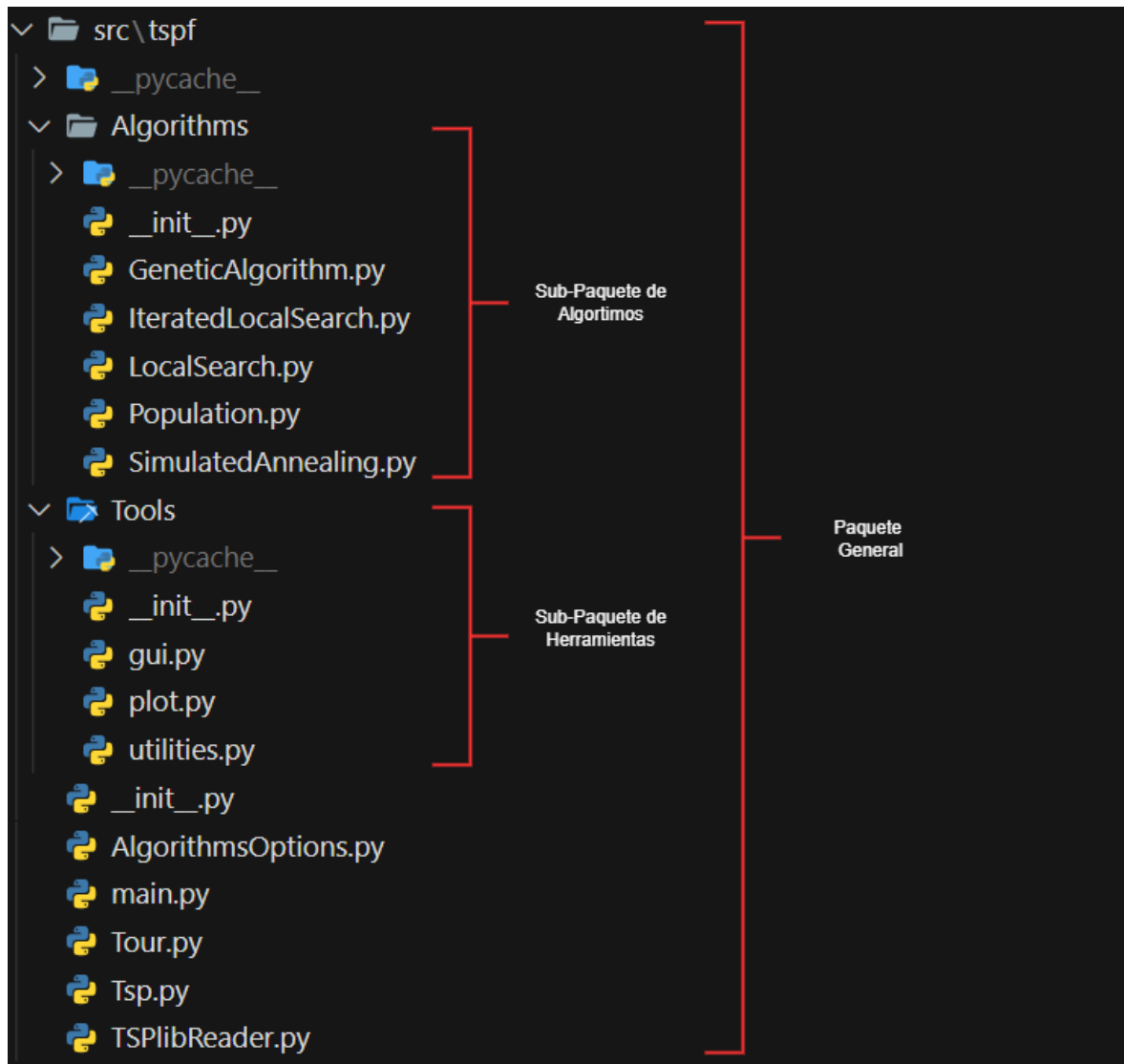


Figura 2.1: Distribución de los paquetes

2.1 Paquete General

Este paquete contiene los módulos generales para gestionar el problema TSP y sus métodos de búsqueda:

- **__init__.py**: Indicador del paquete que contiene todos los imports compartidos por este.
- **AlgorithmsOptions.py**: Módulo que contiene la clase que administra todas las opciones del framework, así como también la de los algoritmos de métodos de búsqueda.

- **main.py:** Modulo que contiene la función que crea las instancias de las clases y ejecuta los métodos de búsqueda.
- **Tour.py:** Modulo que contiene la clase que representa un recorrido del problema.
- **Tsp.py:** Modulo que contiene la clase que representa el problema TSP.
- **TSPlibReader.py:** Modulo que contiene la clase que ejecuta la lectura de una instancia de problema TSP y genera las matrices de distancias.

2.2 Sub-Paquete de Algoritmos

Este paquete contiene todos los algoritmos de métodos de búsqueda implementados y sus complementos:

- **__init__.py:** Indicador del paquete que contiene todos los imports compartidos por este.
- **GeneticAlgorithm.py:** Modulo con la clase que implementa el método de búsqueda de Algoritmo Genético.
- **IteratedLocalSearch.py:** Modulo con la clase que implementa el método de búsqueda local iterativo.
- **LocalSearch.py:** Modulo con la clase que implementa el método de búsqueda local.
- **Population.py:** Modulo con la clase que implementa todos los métodos para manipular las poblaciones generadas por algoritmo genético.
- **SimulatedAnnealing.py:** Modulo con la clase que implementa el método de búsqueda de Simulated Annealing.

2.3 Sub-Paquete de Herramientas

Este paquete contiene todas las herramientas y utilidades utilizadas por el framework.

- **__init__.py:** Indicador del paquete que contiene todos los imports compartidos por este.
- **utilities.py:** Modulo distintas utilidades utilizadas por los demás módulos.
- **plot.py:** Modulo encargado de generar y mostrar los distintos gráficos utilizados como los generados en la trayectoria de las soluciones.
- **gui.py:** Modulo encargado de generar y mostrar la interfaz gráfica de usuario.

3 Como agregar un nuevo método de búsqueda

Para agregar un nuevo método de búsqueda se deben seguir los siguientes pasos:

3.1 Agregar las opciones

Para agregar las opciones del nuevo método de búsqueda nos dirigimos a el módulo que contiene las contiene: **AlgorithmsOptions.py**, dentro de este agregamos las clases enum que utilizará el método:

1. Primero agregamos la identificación del método de búsqueda como tipo de metaheurística y los otros tipos de clases enum que estimemos necesarios para facilitar las opciones:

```
class MHType(Enum):  
    """Tipos de Metaheristicas disponibles  
    SA: Simulated Annealing  
    GA: Genetic Algorithm  
    LS: Local Search  
    ILS: Iterated Local Search  
    """  
    SA = 'SA'  
    GA = 'GA'  
    LS = 'LS'  
    ILS = 'ILS'
```

Figura 3.1: enum metaheurísticas

2. Luego dentro e la clase principal: **AlgorithmsOptions**, agregamos los atributos de clase con las opciones del algoritmo.
3. Luego en el método **readOptions** (aprox. Línea 241) agregamos los argumentos que utilizará este método:

Definir los argumentos que utilizará en la terminal

```
# Definir argumentos de Algoritmo Genetico
parser.add_argument("-p", "--psize", help="Cantidad de individuos")
parser.add_argument("-o", "--osize", help="Cantidad de hijos a generar")
parser.add_argument("-ps", "--pselection", help="Operador de seleccion")
parser.add_argument("-cr", "--crossover", help="Operador de crossover")
parser.add_argument("-mu", "--mutation", help="Operador de mutacion")
parser.add_argument("-mp", "--mprobability", help="Probabilidad de mutacion")
parser.add_argument("-gs", "--gselection", help="Operador de seleccion")
parser.add_argument("-g", "--gstrategy", help="Estrategia de seleccion")

# Definir argumentos de Local Search e Iterated Local Search
parser.add_argument("-b", "--best", help="Ejecuta Local Search en el mejor individuo")
parser.add_argument("-per", "--perturbation", help="Tipo de perturbacion")
parser.add_argument("-np", "--nperturbations", help="Cantidad de perturbaciones")

# Procesar argumentos
```

Figura 3.2: Definir Argumentos

Procesar nuevo tipo de metaheurística en el método `argsGeneral` (aprox. Línea 364):

```
# Seleccion de Metaheuristica
if (args.metaheuristic or 'metaheuristic' in kwargs):
    val = args.metaheuristic.upper() if args.metaheuristic else kwargs['metaheuristic'].upper()
    if (val == 'SA'):
        self.metaheuristic = MHType.SA
    elif (val == 'GA'):
        self.metaheuristic = MHType.GA
    elif (val == 'LS'):
        self.metaheuristic = MHType.LS
    elif (val == 'ILS'):
        self.metaheuristic = MHType.ILS
    else: print(f"{bcolors.FAIL}Error: Metaheuristica no reconocida (-mh | --metaheuristic) {bcolors.ENDC}")
```

Figura 3.3: Arg. nueva metaheurística

Agregar nuevo método que procese los argumentos y valide que no hay errores:

```

# Procesar argumentos
args = parser.parse_args()

# Procesar argumentos generales
self.argsGeneral(args, kwargs)

if self.metaheuristic == MHType.SA:
    # Procesar argumentos de Simulated Annealing
    self.argsSA(args, kwargs)
    # Validar logica de opciones
    if self.errorsSA():
        exit()
elif self.metaheuristic == MHType.GA:
    # Procesar argumentos de Algoritmo Genetico
    self.argsGA(args, kwargs)
    # Validar logica de opciones
    if self.errorsGA():
        exit()
elif self.metaheuristic == MHType.LS or self.metaheuristic == MHType.ILS:
    # Procesar argumentos de Local Search e Iterated Local Search
    self.argsLS(args, kwargs)

```

Figura 3.4: Procesar Argumentos

4. Agregar las opciones para mostrar las opciones en el método **printOptions** (aprox. Línea 606)

3.2 Crear el módulo

Para agregar el nuevo método de búsqueda nos dirigimos a el paquete que contiene los algoritmos, y dentro de este:

1. Creamos el nuevo módulo que contenga la **clase** en el **paquete de algoritmos**. Esta clase debe tener un constructor que reciba una instancia de un objeto de opciones y otra instancia de un objeto con el problema TSP como se ve en este ejemplo:

```
def __init__(self, options: AlgorithmsOptions = None, problem: Tsp = None) -> None:

    # Atributos de instancia
    self.problem: Tsp # Problema TSP

    self.cooling: CoolingType # Esquema de enfriamiento

    self.move_type: TSPMove # Tipo de movimiento

    self.alpha = 0.0 # Parametro alfa para el esquema de enfriamiento geometrico

    self.best_tour: Tour # Mejor tour

    self.evaluations = 1 # numero de evaluaciones

    self.total_time = 0.0 # tiempo de ejecucion de Simulated Annealing

    self.options: AlgorithmsOptions # Opciones

    self.trajectory = [] # lista con la trayectoria de la solucion
```

Figura 3.5: Constructor de clase

Como se ve en la figura, el método debe contar con sus opciones como **atributo de instancia**, además de la mejor solución, de tipo Tour que guardará la mejor solución y se utilizará como punto de comparación para obtener mejores soluciones a través de las iteraciones del algoritmo. También, para concretar la trayectoria se debe tener una lista que guarde la **trayectoria**. Esta trayectoria, definida en el módulo **utilities.py** del paquete de herramientas (Tools), se utilizará para generar la visualización de esta y debería guardarse por cada vez que el método de búsqueda implementado encuentre una mejor solución de la que ya tenemos. Para utilizarla debemos ir añadiendo a la lista instancias del objeto **Trajectory** como se ve a continuación:

```
# Guardar Trayectoria
self.trajectory.append( Trajectory(
    tour=tour.current.copy(),
    cost=tour.cost,
    iterations=self.evaluations,
    evaluations=self.evaluations) )
```

Figura 3.6: Guardar Trayectoria

2. Implementar estos métodos para guardar los archivos de solución y trayectoria (se copiar desde otro algoritmo). Ver modulo **utilities.py** en el paquete de herramientas (Tools) si desea modificar.

```

def printSolFile(self, outputSol: str) -> None:
    """ Guarda la solucion en archivo de texto"""
    utilities.printSolToFile(outputSol, self.best_tour.current)

def printTraFile(self, outputTra: str) -> None:
    """ Guarda la trayectoria de la solucion en archivo de texto"""
    utilities.printTraToFile(outputTra, self.trajectory)

```

Figura 3.7: métodos de archivos

3. Implementar método para la visualización de la trayectoria (se copiar desde otro algoritmo). Ver modulo **plot.py** en el paquete de herramientas (Tools) si desea modificar.

```

def visualize(self) -> None:
    """ Visualiza la trayectoria de la solucion """
    plot.Graph.replit = self.options.replit
    plot.Graph.trajectory = self.trajectory

    plot.show(self.options.gui)

```

Figura 3.8: método para visualizar

4. (OPCIONAL) Implementar método **updateLog** para guardar log con las mejores soluciones utilizando este método de búsqueda. Se recomienda consultar uno ya implementado de otro algoritmo para guiarse.

```
def updateLog(self) -> None:
    """ Actualiza el registro de mejores soluciones con todas las caracteris
    # crea la carpeta en caso de que no exista (python 3.5+)
    Path("log/").mkdir(exist_ok=True)
    logFile = "log/SAlog.csv"
    # usar el archivo en modo append
    with open(logFile, "a", newline="\n") as csvfile:

        print(f"{bcolors.OKGREEN}\nActualizando log con mejores soluciones e
        # Headers
        fields = ["solution", "cost", "instance", "date", "alpha", "t0", "tmin",
                  "cooling", "seed", "move", "max_evaluations", "max_time", "init
        writer = csv.DictWriter(csvfile, delimiter=';', fieldnames=fields)
        # Si la posicion de el archivo es cero se escriben los headers
        if not csvfile.tell():
            writer.writeheader()
```

Figura 3.9: Actualizar log

5. Implementar método **print_best_solution** que muestre la mejor solución (eliminar **updateLog** en caso de no implementarlo).

```
def print_best_solution(self) -> None:
    """ Escribir la mejor solucion """
    self.updateLog()
    print()
    print(f"\t\t{bcolors.UNDERLINE}Mejor Solucion Encontrada")
    self.best_tour.printSol(True)
    print(f"{bcolors.BOLD}Total de evaluaciones:{bcolors.BLUE} {self.evaluations}")
    print(f"{bcolors.BOLD}Tiempo total de busqueda con {bcolors.BLUE} {self.time}")
```

Figura 3.10: Mostrar mejor solución

6. Importar en `__init__.py` dentro del paquete, agregar el import siguiendo el patrón:

```

import csv
import math
from os import path
from datetime import datetime
from pathlib import Path
import statistics as stats
from timeit import default_timer as timer
from prettytable import PrettyTable

from src.tspf.Algorithms.Population import Population
from src.tspf.Algorithms.GeneticAlgorithm import GeneticAlgorithm
from src.tspf.Algorithms.SimulatedAnnealing import SimulatedAnnealing
from src.tspf.Algorithms.LocalSearch import LocalSearch
from src.tspf.Algorithms.IteratedLocalSearch import IteratedLocalSearch

```

Figura 3.11: Imports `__init__.py`

3.3 Agregar a modulo principal

Una vez completados los pasos anteriores podemos agregar el método de búsqueda a el módulo principal **main.py** en su función **main** siguiendo los pasos:

1. Importar desde el paquete de algoritmos:

```

from .Algorithms import GeneticAlgorithm, SimulatedAnnealing, LocalSearch, IteratedLocalSearch, timer
from .Tools import bcolors, gui
from . import sys, os, AlgorithmsOptions, MHType, Tsp, Tour

```

Figura 3.12: importar método en modulo principal

2. Dentro de la condicional correspondiente crear una instancia de la clase con el método de búsqueda llamada **solver**:


```

# leer e interpretar el problema TSP leído desde la instancia definida
problem = Tsp(filename=options.instance)

# Ejecutar Metaheurística Simulated Annealing
if (options.metaheuristic == MHType.SA):

    # Solucion inicial
    first_solution = Tour(type_initial_sol=options.initial_solution, problem=problem)
    # Crear solver
    solver = SimulatedAnnealing(options=options, problem=problem)
    # Ejecutar la búsqueda
    solver.search(first_solution)

# Ejecutar Metaheurística Algoritmo Genético
elif (options.metaheuristic == MHType.GA):
    # Crear solver
    solver = GeneticAlgorithm(options=options, problem=problem)
    # Ejecutar la búsqueda
    solver.search()

elif (options.metaheuristic == MHType.LS):
    # Solucion inicial
    first_solution = Tour(type_initial_sol=options.initial_solution, problem=problem)
    # Crear solver
    solver = LocalSearch(options=options, problem=problem)
    # Ejecutar la búsqueda
    solver.search(first_solution)

```

Figura 3.13: Crear instancia del algoritmo

Debe llamarse **solver** para ser compatible con los métodos en común con los demás algoritmos

```

# Guardar la solución y trayectoria en archivo
solver.printSolFile(options.solution)
solver.printTraFile(options.trajectory)
# Escribir la solución por consola
solver.print_best_solution()

end = timer() # tiempo final de ejecución
print(f"{bcolors.BOLD}Tiempo total de ejecución: {bcolo

if options.visualize:
    solver.visualize()

```

Figura 3.14: Nombre de la instancia

3. El método de búsqueda debería estar funcionando, probar con “python tspf.py -mh <abreviación definida>”.

3.4 Implementar en la Interfaz Gráfica

Si desea implementar el nuevo método de búsqueda debe seguir los siguientes pasos:

1. Dirigirse al paquete de herramientas (Tools) e ir al módulo **gui.py**
2. Agregar selección de método de búsqueda en la clase **Gui** y el método **mainScreen** (aprox. Línea 145)

Crear el botón acomodándolo dentro del **grid** (seguir lógica del **grid**) y creando el método que lo gestione.

```
def mainScreen(self) -> None:

    self.frameL.destroy()
    self.root.geometry('960x540')
    self.frame.destroy()
    self.frame = LabelFrame(
        self.root,
        text='Metodo de Búsqueda',
        bg='#f0f0f0',
        font=('consolas', 20)
    )

    self.frame.pack(anchor='center', pady=10)

    f = Frame(self.frame)
    f.grid(row=0, column=0)

    b = Button(f, text='Simulated Annealing', command=self.simulatedAnnealing)
    b.config(font=buttonFont)
    #b.pack(anchor='ne', side='left', padx=50, pady=50)
    b.grid(row=0, column=0, padx=25, pady=25)
```

Figura 3.15: Agregar nuevo algoritmo GUI

3. Crear y gestionar las opciones en el método que administre el algoritmo:

```

""" SIMULATED ANNEALING """

def simulatedAnnealing(self) -> None:
    """ Configura las opciones de simulated annealing """

    self.options.metaheuristic = MHType.SA
    self.frameL.destroy()
    self.frame.destroy()
    self.optionsFrame()

    frameSA = LabelFrame(
        self.frameOptions,
        text='Simulated Annealing',
        bg='#f0f0f0',
        font=("consolas", 22)
    )

    #frameSA.pack(anchor='n', side='right', padx=25, pady=15)
    frameSA.grid(row=0, column=3, padx=15, pady=10)

```

Figura 3.16: Método del algoritmo GUI

Dentro de este seguir el ejemplo del patrón anterior manteniendo la posición del grid, pero con los nombres del nuevo algoritmo y agregar las opciones de este posteriormente.

4. Agregar a la ejecución en el método **search** (aprox. Línea 814) de forma similar a lo hecho para la ejecución en la terminal:

```

# Mostrar Opciones
options.printOptions()

# leer e interpretar el problema TSP leído desde la instancia definida
problem = Tsp(filename=options.instance)

# Ejecutar Simulated Annealing
if (options.metaheuristic == MHType.SA):

    # Solucion inicial
    first_solution = Tour(type_initial_sol=options.initial_solution, problem=problem)
    # Crear solver
    self.solver = SimulatedAnnealing(options=options, problem=problem)
    # Ejecutar la busqueda
    self.solver.search(first_solution)

# Ejecutar Algoritmo Genetico
elif (options.metaheuristic == MHType.GA):
    # Crear solver
    self.solver = GeneticAlgorithm(options=options, problem=problem)
    # Ejecutar la busqueda
    self.solver.search()

# Ejecutar Local Search

```

Figura 3.17: Agregar ejecución GUI

5. Agregar cambio de algoritmo en la clase de **MenuBar** (aprox. Línea 940) siguiendo el patrón:

```

# cambiar metodo
self.editMenu = Menu(self.menuBar, tearoff = False)
self.editMenu.add_command(Label="Simulated Annealing", command=Lambda: self.changeSearch(MHType.SA))
self.editMenu.add_command(Label="Algoritmo Genetico", command=Lambda: self.changeSearch(MHType.GA))
self.editMenu.add_command(Label="Local Search", command=Lambda: self.changeSearch(MHType.LS))
self.editMenu.add_command(Label="Iterated Local Search", command=Lambda: self.changeSearch(MHType.ILS))

self.menuBar.add_cascade(menu=self.editMenu, Label="Cambiar de Algoritmo")

```

Figura 3.18: Cambio de algoritmo GUI

6. Gestionar el cambio de algoritmo siguiendo el patrón en el método **changeSearch**

```

def changeSearch(self, searchType: MHType) -> None:
    """ Cambia de metodo de busqueda en la barra de herramientas """

    if self.gui.frameOptions != None:
        self.gui.frameOptions.destroy()
    if not self.gui.options.replit and self.gui.frameFeedback != None:
        self.gui.frameFeedback.destroy()

    if searchType == MHType.SA:
        self.gui.simulatedAnnealing()
    elif searchType == MHType.GA:
        self.gui.geneticAlgorithm()
    elif searchType == MHType.LS:
        self.gui.localSearch()
    elif searchType == MHType.ILS:
        self.gui.iteratedLocalSearch()

```

Figura 3.19: Gestionar cambio de algoritmo GUI

7. Probar con “python tspf.py --gui”

4 Componentes algorítmicos

Algunos componentes algorítmicos de interés que podría resultar de utilidad saber su ubicación y como agregar alguno nuevo en caso de ser necesario.

4.1 Componentes algorítmicos manejar la población de Algoritmo Genético

A continuación, veremos algunas secciones con los métodos importantes utilizados para manipular las poblaciones de algoritmo genético. Estos métodos están ubicados en el módulo **Population.py** dentro del paquete **Algorithms**.

- **Componentes de selección de padres:** Línea 240 hasta 420
- **Componentes de cruzamiento:** Línea 431 hasta 632
- **Componentes de mutación:** Línea 654 hasta 720
- **Componentes de selección de población:** Línea 733 hasta 898

4.1.1 Como añadir componentes algorítmicos a Algoritmo Genético

Para añadir un nuevo componente a algoritmo genético, tanto como de selección de padres, cruzamiento, mutación o de selección de población, debemos añadirlo en **AlgorithmsOptions.py** del módulo principal.

```
53  """ ALGORITMO GENETICO """
54
55  class SelectionType(Enum):
56      """Tipos de seleccion de individuos
57          RANDOM: Seleccion aleatoria
58          BEST: Seleccion de los mejores (elitismo)
59          ROULETTE: Seleccion proporcional al fitness
60          TOURNAMENT: Seleccion de torneos k=3
61      """
62      RANDOM = 'RANDOM'
63      BEST = 'BEST'
64      ROULETTE = 'ROULETTE'
65      TOURNAMENT = 'TOURNAMENT'
66
67  class CrossoverType(Enum):
68      """Tipos de cruzamiento disponibles
69          PMX: (partially-mapped crossover) hace swap adaptando los tours
70          O1X: (order 1 crossover)
71          OPX: (one point crossover) se realiza cruzamiento en un punto utilizando una lista de referencia
72      """
73      PMX = 'PMX'
74      OX = 'OX'
75      OPX = 'OPX'
76
77  class SelectionStrategy(Enum):
78      """Estrategias de seleccion de individuos de la poblacion
79          MU_LAMBDA: Estrategia (mu, lambda)
80          MUPLUSLAMBDA: Estrategia (mu+lambda)
81      """
82      MULAMBDA = 'MULAMBDA'
```

Figura 4.1: Agregar un componente algoritmo genético

Luego, gestionamos su configuración para que puedan ser leídos sus argumentos (aprox. Línea 469) al ejecutarse por la terminal.

```
469 # Seleccion de padres
470 if (args.pselection or 'psselection' in kwargs):
471     val = args.pselection.lower() if args.pselection else kwargs['psselection'].lower()
472     if (val == 'random'):
473         self.pselection_type = SelectionType.RANDOM
474     elif (val == 'best'):
475         self.pselection_type = SelectionType.BEST
476     elif (val == 'roulette'):
477         self.pselection_type = SelectionType.ROULETTE
478     elif (val == 'tournament'):
479         self.pselection_type = SelectionType.TOURNAMENT
480     else: print(f"{bcolors.FAIL}Error: Opcion no reconocida en Seleccion de padres (-ps | --psselection) {b"}
481
482 # Operador de Cruzamiento
483 if (args.crossover or 'crossover' in kwargs):
484     val = args.crossover.lower() if args.crossover else kwargs['crossover'].lower()
485     if (val == 'ox'):
486         self.crossover_type = CrossoverType.OX
487     elif (val == 'opx'):
488         self.crossover_type = CrossoverType.OPX
489     elif (val == 'pmx'):
490         self.crossover_type = CrossoverType.PMX
491     else: print(f"{bcolors.FAIL}Error: Opcion no reconocida en Operador de Cruzamiento (-o | --crossover) {b"}
492
493 # Operador de mutacion
494 if (args.mutation or 'mutation' in kwargs):
495     val = args.mutation.lower() if args.mutation else kwargs['mutation'].lower()
496     if (val == '2opt' or val == '2-opt'):
497         self.mutation_type = TSPMove.TWO_OPT
498     elif (val == '3opt' or val == '3-opt'):
499         self.mutation_type = TSPMove.THREE_OPT
```

Figura 4.2: Argumentos nuevo componente algoritmo genetico

Luego, importamos en `__init__.py` del paquete principal para que sea parte del paquete.

```
6 import argparse
7 import math
8 import os
9 import sys
10 import time
11 from enum import Enum
12 from decimal import Decimal
13
14 from src.tspf.TSPlibReader import TSPlibReader
15 from src.tspf.AlgorithmsOptions import AlgorithmsOptions, InitialSolution, CoolingType, MHType, SelectionStrategy, Se
16 from src.tspf.Tsp import Tsp
17 from src.tspf.Tour import Tour
```

Figura 4.3: Importar en paquete principal

Y nos vamos a **Population.py** del paquete **Algorithms** y lo importamos.

```
6 from ..Tools import utilities, bcolors
7 from . import stats
8 from .. import Tour, Tsp, CrossoverType, InitialSolution, TSPMove, SelectionType
9
```

Figura 4.4: Importar en Population.py

Luego, dependiendo del tipo de componente que estemos añadiendo, nos vamos a la sección del código que necesitemos, para este ejemplo utilizaremos los componentes de cruzamiento. Para esto, nos vamos a la sección de cruzamiento y gestionamos todos los tipos de cruzamiento.

```
439 def crossover(self, parents_id: List, ctype: CrossoverType) -> List:
440     """Aplica el operador cruzamiento
441
442     Parameters
443     -----
444     parents_id : list
445         lista con los índices de los individuos padres seleccionados para cruzamiento
446     ctype : CrossoverType
447         tipo de cruzamiento
448
449     Returns
450     -----
451     list
452         lista con los individuos hijos generados
453
454     """
455     offspring = []
456     parents = []
457     # Obtener individuos padres con los ids
458     parents = self.getIndividuals(parents_id)
459
460     # Aplicar Crossover
461     if (ctype == CrossoverType.PMX):
462         offspring = self.PMXCrossover(parents)
463     elif (ctype == CrossoverType.OX):
464         offspring = self.OXCrossover(parents)
465     elif (ctype == CrossoverType.OPX):
466         offspring = self.OPXCrossover(parents)
467     else:
468         offspring = self.OXCrossover(parents)
469
470     return offspring
```

Figura 4.5: Gestionar nuevo componente de cruzamiento

Posteriormente, añadimos el nuevo componente y debería integrarse correctamente integrado dentro de algoritmo genético si se siguieron las instrucciones correctamente.

4.2 Componentes algorítmicos para manejar el Tour

Los compontes correspondientes para manejar el Tour, el cual representa el recorrido o solución al problema, estos se encuentran en el **Tour.py** dentro del paquete principal.

4.2.1 Como añadir componentes algorítmicos a Tour

Para este ejemplo veremos como seria añadir un nuevo tipo de movimiento TPS, y al igual que anteriormente, debemos partir por **AlgorithmsOptions.py** del paquete principal y agregamos el nuevo tipo.


```

21 class TSPMove(Enum):
22     """Tipos de movimientos disponibles para el TSP
23     TWO_OPT: Operador 2-opt
24     THREE_OPT: Operador 3-opt
25     SWAP: Operador swap
26     """
27     TWO_OPT = 'TWO_OPT'
28     THREE_OPT = 'THREE_OPT'
29     SWAP = 'SWAP'
30

```

Figura 4.6: Tipos de movimiento TSP

Luego gestionamos ese movimiento (aprox. Línea 391) para que pueda ser leído por la terminal.

```

390 # Selecion del movimiento para la metaheurística
391 if (args.move or 'move' in kwargs):
392     val = args.move.lower() if args.move else kwargs['move'].lower()
393     if (val == '2opt' or val == '2-opt'):
394         self.move = TSPMove.TWO_OPT
395     elif (val == '3opt' or val == '3-opt'):
396         self.move = TSPMove.THREE_OPT
397     elif (val == 'swap'):
398         self.move = TSPMove.SWAP
399     else: print(f"{bcolors.FAIL}Error: Tipo de movimiento no reconocido (-mhm | --move) {bcolors.ENDC}")
400

```

Figura 4.7: Gestionar nuevo movimiento TSP

Luego, volvemos a **Tour.py** y agregamos el nuevo movimiento al módulo y a su vez lo incluimos dentro de la selección aleatoria (aprox. línea 310).

```

310 def randomMove(self, move_type: TSPMove) -> None:
311     """ Aplica un movimiento aleatorio recibido por parametro del tipo TSPMove """
312
313     n1 = utilities.random.randint(0, self.problem.getSize()-1)
314     n2 = utilities.random.randint(0, self.problem.getSize()-1)
315     # Determinar que sean numeros diferentes
316     while (n1 == n2):
317         n2 = utilities.random.randint(0, self.problem.getSize()-1)
318
319     if move_type == TSPMove.THREE_OPT:
320         i, j, k = self.getIndThreeOpt()
321         #print(i,j,k)
322
323     # Seleccionar el tipo de movimiento
324     if (move_type == TSPMove.TWO_OPT):
325         self.twoOptSwap(n1, n2)
326     elif (move_type == TSPMove.SWAP):
327         self.swap(n1, n2)
328     elif (move_type == TSPMove.THREE_OPT):
329         self.bestThreeOptSwap(i, j, k)
330     else:
331         self.swap(n1, n2)
332

```

Figura 4.8: Movimiento TSP aleatorio

