Manual de usuario
SOLVER BIO-INSPIRADOS

José Lara Arce
Pontificia Universidad Católica de Valparaíso, Chile
jose.lara.a01@mail.pucv.cl

April 29, 2025

Índice

1. Dependencias

Para el correcto funcionamiento del solver, es necesario contar con los siguientes requisitos:

- **Versión de Python**: 3.11.9 (64 bits). Una de las librerías hace conflicto en versiones posteriores.
- **IDEs recomendados**: Visual Studio Code, PyCharm.
- **Gestión de dependencias**: Se utiliza el módulo `venv`, incluido en la biblioteca estándar de Python (desde la versión 3.3), para crear entornos virtuales.

Nota: La creación de un entorno virtual es opcional. Si no desea aislar las dependencias del proyecto y prefiere instalar los paquetes de Python directamente en el entorno global del sistema, puede omitir los pasos de creación y activación del entorno virtual y proceder directamente a la sección 1.3. Sin embargo, se recomienda encarecidamente el uso de un entorno virtual para evitar conflictos entre versiones de librerías en diferentes proyectos.

1.1. Crear y Activar el entorno virtual con venv

El módulo `venv` permite crear entornos virtuales aislados para gestionar paquetes sin afectar otras configuraciones del sistema. Para ello:

1. Navegue hasta el directorio de su proyecto en la terminal o línea de comandos. 2. Cree un entorno virtual llamado `env` (puede elegir otro nombre si lo prefiere) con el siguiente comando:

```
python -3.11 -m venv env
```

3. Una vez creado, active el entorno virtual. El comando varía según su sistema operativo y terminal:

- **Windows (cmd.exe)**:

```
env\Scripts\activate.bat
```

- **Windows (PowerShell)**:

```
.\env\Scripts\Activate.ps1
```

(Nota: Es posible que necesite ajustar la política de ejecución de scripts en PowerShell con `Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope Process`).

- **Unix o MacOS (bash/zsh)**:

```
source env/bin/activate
```

Al activarse correctamente, la línea de comandos debería mostrar el nombre del entorno virtual entre paréntesis al inicio, similar a:

```
(env) C:\Users\Usuario\Proyecto>
```

o

```
(env) usuario@computador:~/Proyecto$
```

1.2. Configurar Visual Studio Code

Para asegurarse de que Visual Studio Code use el intérprete de Python correcto dentro del entorno virtual:

1. Abra la **Command Palette** (Paleta de Comandos) con:

```
Ctrl + Shift + P
```

(o `Cmd + Shift + P` en macOS). 2. Escriba `>Python: Select Interpreter` (o `>Python: Seleccionar Intérprete`) y presione Enter. 3. Seleccione la opción que apunta al ejecutable de Python dentro de su entorno virtual. Usualmente contendrá la ruta `./env/bin/python` (Unix/macOS) o `..exe` (Windows). VS Code a menudo detecta y sugiere automáticamente los entornos `venv`.

1.3. Instalar dependencias

Para instalar las dependencias del proyecto, asegúrese de que el entorno virtual (`env`) esté activado (verá `(env)` al inicio de la línea de comandos) y ejecute en la consola:

```
pip install -r requirements.txt
```

2. Introducción al solver

El solver que presentamos en este manual está diseñado para resolver problemas de optimización utilizando técnicas de metaheurísticas. Estas técnicas son métodos de búsqueda de soluciones aproximadas para problemas donde las soluciones exactas son difíciles o imposibles de obtener debido a la complejidad computacional. En particular, el solver implementa las siguientes metaheurísticas hasta la fecha:

- Arithmetic Optimization Algorithm (AOA)
- Enhanced Beluga Whale Optimization Algorithm (EBWOA)
- Elk Herd Optimizer (EHO)
- Eurasian Oystercatcher Optimizer (EOO)
- Flilled Lizard Optimization (FLO)
- Fox-inspired Optimization Algorithm (FOX)
- Genetic Algorithm (GA)
- Gannet Optimization Algorithm (GOA)
- Grey Wolf Optimizer (GWO)
- Fox-inspired Optimization Algorithm (FOX)
- Genetic Algorithm (GA)
- Honey Badger Algorithm (HBA)
- Horned Lizard Optimization Algorithm (HLOA)
- Lyrebird Optimization Algorithm (LOA)
- Narwhal Optimization (NO)
- Puma Optimizer (PO)
- Pendulum Search Algorithm (PSA)
- Particle Swarm Optimization (PSO)
- Quokka Optimization Algorithm (QSO)
- Reptile Search Algorithm (RSA)
- Secretary Bird Optimization Algorithm (SBOA)

- Sine Cosine Algorithm (SCA)
- Sea-Horse Optimizer (SHO)
- Tasmanian Devil Optimization (TDO)
- Whale Optimization Algorithm (WOA)
- Wombat Optimization Algorithm (WOM)

El solver permite al usuario optimizar diferentes funciones objetivo, que se definen en el archivo de configuración.

3. Estructura interna del solver

El solver está organizado en módulos con el objetivo de facilitar su mantenimiento y expansión. A continuación, se describen los componentes clave de la estructura del código:

- **BD**: Módulo encargado de gestionar la base de datos, así como su correspondiente constructor.
- **Discretization**: Contiene los elementos responsables de la binarización de las soluciones.
- **Diversity**: Incluye las métricas utilizadas para evaluar y visualizar la diversidad de las soluciones generadas por el solver.
- **Explicaciones_Manuales**: Reúne archivos PDF con explicaciones detalladas sobre el funcionamiento del solver y los problemas que se abordan a lo largo de la asignatura.
- **Graficos_Benchmark**: Contiene archivos PDF con gráficos que ilustran las funciones matemáticas, permitiendo a los usuarios visualizar la complejidad de las funciones que están optimizando.
- **Metaheuristics**: Incluye los archivos Python (`.py`) correspondientes a las diversas metaheurísticas implementadas en el solver.
- **Problem**: Contiene las instancias de los problemas abordados en la asignatura, como BEN, SCP y USCP.
- **Solver**: Agrupa los códigos que implementan los solvers específicos para cada uno de los problemas mencionados.
- **Util**: Contiene archivos de utilidad empleados por los módulos anteriores para garantizar su correcto funcionamiento.

- **Util/json**: Incluye archivos de configuración necesarios para realizar experimentos, así como archivos relacionados con las rutas y parámetros correspondientes.

3.1. Archivos principales

Los siguientes son los archivos más importantes dentro del proyecto, cada uno con una función específica:

- **analisisBEN.py (analisisSCP, analisisUSCP)**: Archivos Python ejecutables utilizados para realizar análisis sobre ejecuciones previas. Estos archivos son útiles para presentaciones o para verificar que la metaheurística esté funcionando correctamente.
- **crearBD.py**: Crea la base de datos si esta no ha sido creada previamente.
- **levantarCMD.py**: Permite la creación de múltiples instancias de la línea de comandos (CMD) según las necesidades del usuario, facilitando la ejecución paralela del solver.
- **limpiarEntorno.py**: Elimina todos los archivos dentro de la carpeta de resultados, garantizando un entorno limpio para nuevas ejecuciones.
- **main.py**: El programa principal que coordina y ejecuta las funcionalidades del solver.
- **poblarDB.py**: Archivo encargado de poblar la base de datos con los experimentos previamente configurados.
- **reiniciarDB.py**: Permite reiniciar la base de datos, eliminando los experimentos existentes y configurando las instancias por defecto.
- **reiniciarSolver.py**: Reinicia el solver, restaurándolo a su estado inicial, similar a cuando se descarga por primera vez. Este proceso incluye la ejecución de `limpiarEntorno.py` y `reiniciarDB.py`. Será de los archivos con más utilidad para ustedes.

4. Ejemplo de uso y explicación

A continuación, se presenta un ejemplo detallado de cómo utilizar el solver para la optimización de funciones matemáticas mediante el algoritmo SBOA.

4.1. Optimización de funciones matemáticas con el algoritmo SBOA

Para comenzar con la optimización, lo primero que deben hacer es seleccionar las funciones que desean optimizar. En este ejemplo, vamos a optimizar las funciones 'F1', 'F8', 'F9' y 'F16'.

4.1.1. Paso 1: Configuración del archivo `experiments_config.json`

Diríjanse al directorio `'Util/json'` y abran el archivo `experiments_config.json`. Dentro de este archivo, encontrarán varias configuraciones que deberán ajustar de acuerdo con los problemas y parámetros específicos. A continuación, se detallan los cambios necesarios para este caso:

- Configuración de funciones BEN

  Para trabajar con funciones matemáticas, deben asegurarse de que la opción `'ben'` esté configurada en `true`, ya que esto indica que se optimizarán funciones BEN (Benchmark):

  ```
  "ben": true,
  "scp": false,
  "uscp": false,
  ```

- Selección de metaheurísticas

  A continuación, deben especificar la metaheurística que desean utilizar. En este caso, seleccionamos `SBOA` (Secretary Bird Optimization Algorithm). Este parámetro puede contener una lista de varias metaheurísticas:

  ```
  "mhs": ["SBOA"],
  ```

- Configuración de acciones de discretización (para problemas binarios)

  La sección `DS_actions` es relevante solo para problemas binarios, como SCP y USCP. Dado que estamos trabajando con BEN, pueden dejar este parámetro tal cual está, ya que no afecta la optimización de funciones matemáticas:

  ```
  "DS_actions": ["S3-ELIT", "V3-ELIT"],
  ```

- Especificación de dimensiones

  En la sección `dimensiones`, se especifican las dimensiones de las instancias de cada función matemática. A continuación, se detallan las dimensiones de las funciones BEN que se optimizarán. Si desean, pueden

modificar la dimensión, pero tengan en cuenta que a mayor dimensión, mayor será el tiempo de cómputo y la calidad de las soluciones podría no ser óptima. En nuestro caso, optimizaremos las funciones 'F1', 'F8', 'F9' y 'F16':

```
"dimensiones": {
    "BEN": {
        "F1": [30], "F2": [30], "F3": [30],
        "F4": [30], "F5": [30], "F6": [30],
        "F7": [30], "F8": [30], "F9": [30],
        "F10": [30], "F11": [30], "F12": [30],
        "F13": [30], "F14": [2], "F15": [4],
        "F16": [2], "F17": [2], "F18": [2],
        "F19": [3], "F20": [6], "F21": [4],
        "F22": [4], "F23": [4]
    }
},
```

- Selección de dimensiones para las funciones específicas

  En esta sección, se debe especificar la dimensión de las funciones que se van a optimizar. En el caso de las funciones seleccionadas, "F1", "F8", "F9ƨ "F16", hemos optado por la dimensión 30 para cada una de ellas, excepto por "F16", que tiene dimensión 2. Esto asegura que el proceso de optimización tenga un espacio de búsqueda adecuado para encontrar la solución óptima.

- Consideraciones sobre la dimensión de las funciones

  Es importante tener en cuenta que aumentar la dimensión de las funciones puede incrementar el tiempo de cómputo, por lo que es recomendable elegir dimensiones que equilibren la eficiencia computacional y la precisión de la optimización.

- Configuración de parámetros de experimentos

  En la sección `experimentos`, se definen los parámetros específicos de los experimentos, tales como el número de iteraciones, la población y el número de experimentos a ejecutar. Para este ejemplo, trabajaremos con 100 iteraciones, una población de 50 y un único experimento:

```
"experimentos": {
    "BEN": {"iteraciones": 100, "poblacion": 50,
    "num_experimentos": 1},

    "SCP": {"iteraciones": 100, "poblacion": 10,
```

```
        "num_experimentos": 31},

        "USCP": {"iteraciones": 100, "poblacion": 10,
        "num_experimentos": 31}
    },
```

- Número de iteraciones

  El parámetro de `iteraciones` indica cuántas veces se ejecutará el algoritmo de optimización. En este caso, hemos establecido 100 iteraciones, lo que debería ser suficiente para encontrar una buena aproximación de la solución óptima.

- Tamaño de la población

  El parámetro `poblacion` especifica el número de soluciones (individuos) en cada iteración. Hemos configurado una población de 50, lo cual es adecuado para que el algoritmo explore el espacio de soluciones de manera efectiva sin ser demasiado costoso computacionalmente.

- Número de experimentos

  El parámetro `num_experimentos` indica cuántas veces se debe ejecutar el experimento para obtener resultados estadísticamente significativos. En este caso, configuramos 1 experimento, ya que estamos interesados en ver cómo el algoritmo se comporta en una sola ejecución.

- Selección de instancias a optimizar

  Finalmente, en la sección `instancias`, deben seleccionar las instancias que desean optimizar. En este caso, seleccionaremos las funciones 'F1', 'F8', 'F9' y 'F16' del conjunto BEN:

```
    "instancias": {
        "BEN": ["F1", "F8", "F9", "F16"],
        "SCP": ["41", "51", "61"],
        "USCP": ["uclr10", "uclr11"]
    },
```

4.1.2. Paso 2: Poblar la base de datos

Una vez que hayan configurado el archivo `experiments_config.json`, el siguiente paso es poblar la base de datos. Para ello, deben dirigirse al archivo `poblarDB.py` y ejecutarlo. Este script se encarga de preparar las instancias de los problemas y las configuraciones necesarias para el experimento.

4.1.3. Paso 3: Ejecutar el solver

Finalmente, deben ir al archivo principal `main.py` y ejecutarlo. Al hacerlo, podrán observar cómo el solver optimiza las soluciones en tiempo real a través de la consola.

Durante la ejecución, se mostrará el progreso de las optimizaciones, y podrán ver cómo las soluciones evolucionan conforme avanzan las iteraciones.

5. Agregar una nueva metaheurística

El diseño modular del solver facilita la incorporación de nuevas metaheurísticas bio-inspiradas o de otro tipo. Para añadir un nuevo algoritmo, sigue estos pasos:

5.1. Paso 1: Implementación de la lógica del algoritmo

1. **Crear el archivo Python:**
   Navega hasta el directorio `Metaheuristics/Codes/`. Crea un nuevo archivo Python para tu metaheurística (ej. `NAE.py`).

2. **Definir la función de iteración:**
   Dentro de tu nuevo archivo, define la función principal que ejecutará una iteración del algoritmo (ej. `iterarNAE(...)`).

3. **Establecer la firma estándar de la función:**
   Es **crucial** que tu función utilice los nombres de parámetros estandarizados que espera la función `iterate_population`. La firma general podría ser:

```python
import numpy as np
# otras importaciones...

def iterarNAE(maxIter, iter, dim, population, fitness,
    best, fo=None, lb=None, ub=None, lb0=None, ub0=None,
     vel=None, pBest=None, objective_type='MIN')

    """
    Realiza una iteracion del Nuevo Algoritmo de Ejemplo
     (NAE).

    Args:
        maxIter (int): Maximo de iteraciones.
        iter (int): Iteracion actual.
        dim (int): Dimension del problema.
        population (np.ndarray): Poblacion actual.
        fitness (np.ndarray): Fitness de la poblacion
    actual.
```

```
        best (np.ndarray): Mejor solucion global hasta ahora.
        fo (callable, optional): Funcion objetivo. Defaults to None.
        lb0 (float, optional): Limite inferior escalar. Defaults to None.
        ub0 (float, optional): Limite inferior escalar. Defaults to None.
        ...

    Puede tener mas argumentos segun la metaheuristica a implementar.

    Returns:
        np.ndarray or tuple: Nueva poblacion, o tupla (pob, vel/mejoras).
    """
    N = population.shape[0]
    new_population = population.copy()

    # ------------------------------------
    #    LOGICA DE TU METAHEURISTICA NAE
    # ------------------------------------

    # Ejemplo placeholder:
    # for i in range(N):
    #     r = np.random.rand(dim)
    #     new_population[i] = population[i] + r * (best - population[i])

    # --- Valor de Retorno ---
    return new_population
    # O: return new_population, new_vel
    # O: return new_population, posibles_mejoras

```

Listing 1: Ejemplo firma estándar (NAE) Mejorada

Importante:

- No todos los parámetros son necesarios para cada algoritmo.
- Usa los nombres estándar exactos (`iter`, `best`, `fo`, etc.).
- Devuelve la nueva población (y opcionalmente velocidad o mejoras).
- Para SCP/USCP, la función opera en continuo; la binarización/reparación es posterior.

5.2. Paso 2: Registro en el sistema

Registra la nueva función:

1. **Editar `Metaheuristics/imports.py`:**

2. **Añadir importación:**

```
from .Codes.NAE import iterarNAE # Ajusta NAE

```

3. **Añadir a `metaheuristics`:**

```
metaheuristics = {
    # ...
    "NAE": iterarNAE, # Nueva entrada
    # ...
}

```

4. **Añadir a `MH_ARG_MAP`:**

```
MH_ARG_MAP = {
    # ...
    # Lista los strings de params que tu func necesita:
    'NAE': ('maxIter', 'iter', 'dim', 'population', '
    best'),
    # ...
}

```

¡Asegúrate de que los nombres en la tupla coincidan **exactamente** con los parámetros que tu función `iterarNAE` usa!

5.3. Paso 3: Habilitar en experimentos

Para usar la nueva MH:

1. **Editar `util/json/experiments_config.json`:**

2. **Añadir a `"mhs":`**

```
  // ...
  "mhs": ["SBOA", "PSO", "NAE"], // <--- Agrega aqui
  // ...
}

```

5.4. Paso 4: Consideraciones adicionales (opcional)

- **Versiones específicas (BEN vs SCP/USCP):** Si necesitas lógica muy diferente, crea funciones separadas (ej. `iterarNAE_Ben`, `iterarNAE_Scp`) y regístralas con claves distintas (”NAE_BEN”, ”NAE_SCP”). Tengan en

cuenta que idealmente debe solo haber una lógica para todos los problemas.

- **Algoritmos binarios (GA):** Pueden requerir manejo especial si no siguen el flujo estándar continuo + binarización.
- **Pruebas**: Verifica tu nueva MH ejecutándola en problemas de prueba.

Siguiendo estos pasos, podrás integrar nuevas metaheurísticas de manera organizada.

6. Análisis de resultados

Tras ejecutar los experimentos con el solver y almacenar los resultados detallados en la base de datos, el siguiente paso es procesar esta información para obtener gráficos y estadísticas comparativas. El proyecto proporciona scripts específicos para realizar este análisis post-ejecución.

6.1. Estructura de los scripts de análisis

El análisis está organizado en módulos Python separados, uno para cada tipo de problema principal, lo que facilita su mantenimiento y extensión:

- `analisisBEN.py`: Enfocado en los resultados de funciones benchmark.
- `analisisSCP.py`: Enfocado en los resultados del Set Covering Problem.
- `analisisUSCP.py`: Enfocado en los resultados del Unicost Set Covering Problem.

Cada uno de estos scripts (`analisisBEN.py`, etc.) contiene la lógica necesaria para conectarse a la base de datos, extraer los datos de las corridas experimentales (generalmente los archivos CSV almacenados como BLOBs), calcular métricas (fitness final, tiempo, XPL/XPT), generar gráficos específicos y calcular estadísticas resumidas. Para ser ejecutados de forma organizada, cada script define una función principal, comúnmente llamada `analizar_instancias()` o `main()`.

Un script orquestador, llamado `analisis.py`, se utiliza para ejecutar selectivamente los análisis individuales. Este script importa los módulos `analisisBEN`, `analisisSCP` y `analisisUSCP`.

6.2. Configuración del análisis

El proceso de análisis se configura mediante archivos JSON ubicados en `Util/json/`:

- `dir.json`: Especifica las rutas de los directorios donde se almacenarán todas las salidas del análisis, como gráficos (`DIR_GRAFICOS`), resúmenes (`DIR_RESUMEN`), boxplots (`DIR_BOXPLOT`), etc. Es fundamental que las rutas definidas aquí sean válidas y accesibles.

- `analysis.json`: Este archivo actúa como un panel de control para la ejecución del análisis. Permite seleccionar qué partes del análisis se realizarán. Contiene claves booleanas como `"ben"`, `"scp"`, `üscp"`, que indican si el análisis para ese tipo de problema debe ejecutarse (`true`) o no (`false`).

  Actualmente, el script orquestador (`main_analisis.py`) utiliza principalmente estas claves booleanas para controlar el flujo. Otras claves como `"mhs"`, `"DS_actions"` o `ïnstancias"` pueden estar presentes en el archivo para un futuro control más granular del análisis (ej. analizar solo ciertas MHs o instancias), pero la lógica para usar estos filtros detallados reside dentro de los scripts de análisis individuales (`analisisBEN.py`, etc.) y aún dependen de `experiments_config.json` para las listas completas. En un futuro se trabajará en más detalle.

- `experiments_config.json`: Aunque `analysis.json` controla *qué* scripts se ejecutan, los scripts de análisis individuales siguen consultando `experiments_config.json` para obtener listas de referencia, como la lista completa de metaheurísticas (`"mhs"`) a incluir en las tablas comparativas o gráficos.

6.3. Ejecución del análisis

Para realizar un análisis:
1. **Configura `analysis.json`:** Decide qué tipos de problema quieres analizar y ajusta los valores booleanos (`true`/`false`) correspondientes en este archivo. 2. Verifica que los resultados de los experimentos a analizar estén en la base de datos. 3. Desde la terminal, en el directorio raíz del proyecto, ejecuta el script orquestador:

```
python analisis.py
```

El script leerá `analysis.json` y llamará únicamente a las funciones de análisis correspondientes a los tipos de problema marcados como `true`.

6.4. Salidas generadas

Los scripts de análisis producen un conjunto de resultados organizados en subcarpetas por tipo de problema dentro de los directorios definidos en `dir.json`:

- **Archivos CSV de resumen** (ej. en `./Resultados/Resumen/SCP/`):
  - Resúmenes estadísticos de fitness (mejor, promedio, std dev) por MH.
  - Resúmenes estadísticos de tiempo total por MH.
  - Resúmenes de promedios de XPL % y XPT % por MH.
  - Archivo con el fitness final de cada corrida por MH (usado para boxplots).

  Nota: Los nombres de archivo suelen incluir tipo de problema, instancia y binarización (si aplica).
- **Gráficos** (ej. en `./Resultados/Graficos/BEN/`, `./Resultados/Boxplot/SCP/`, etc.):
  - **Por corrida:** Convergencia de fitness, XPL/XPT, Tiempo por iteración.
  - **Comparativos:** Boxplots y Violinplots de la distribución del fitness final entre corridas para cada MH.

6.5. Interpretación de resultados

Las salidas generadas permiten evaluar y comparar las metaheurísticas:

- **Rendimiento:** Comparar el mejor fitness y el promedio obtenido por cada MH (ver resúmenes y boxplots).
- **Robustez:** Analizar la variabilidad de los resultados entre diferentes corridas (ver la dispersión en boxplots y violinplots).
- **Convergencia:** Observar qué tan rápido y hacia qué valor convergen los algoritmos (gráficos de convergencia).
- **Comportamiento:** Estudiar el balance exploración/explotación (gráficos XPL/XPT).
- **Eficiencia:** Comparar el tiempo de ejecución (resúmenes de tiempo, gráficos de tiempo).

Este análisis ayuda a entender las fortalezas y debilidades de cada metaheurística para cada problema.