DECSAI

Universidad de Granada

Práctica 2: Técnicas de Búsqueda basadas en Poblaciones

Víctor Padilla Cabello 76066048B

vpadi@correo.ugr.es
Grado Ing. Informática Grupo Martes

Curso 2017/2018

Índice

1. Breve descripción del QAP

Estamos ante el problema de asignación cuadrática, el cual nos propone: Dado una serie de locaciones, una matriz representativa de las distancias entre ellas (d) y otra del flujo (f), asignar las unidades a las locaciones para minimizar el flujo y la distancia totales que hay entre ellas, o formulado:

$$min_{\pi \in \prod_N} \left(\sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} * d_{ij} \right)$$

- π es una solución al problema que consiste en una permutación que representa la asignación de la unidad i a la localización (i).
- f_{ij} es el flujo que circula entre la unidad i y la j.
- d_{kl} es la distancia existente entre la localización k y la l.

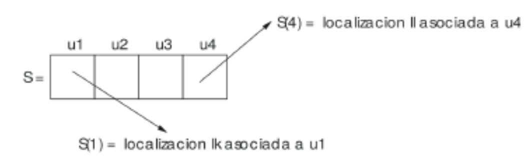
Se le denomina Asignación Cuadrática debido a las sumatorias anidadas, de forma que se convierte en algo parecido a un productorio. El kit del problema se basa en que probar a la fuerza bruta las distintas posibilidades conllevaría demasiado tiempo por la cantidad de permutaciones posibles. Así que intenta buscar una forma que se fije en la solución del problema y su coste final y no en los posibles costes de diferentes permutaciones.

2. Consideraciones comunes a los algoritmos

Aclarar primero que para el desarrollo de esta práctica se ha utilizado un framework sugerido por el seminario 1 de esta asignatura. El nombre de la librería es **ParadisEO**, diseñado especialmente para algoritmos de búsqueda. Tiene implementaciones para algoritmos de búsqueda local, basados en poblaciones, en trayectorias, etc... en el lenguaje $C++$.

2.1. Representacion de soluciones

Para la representación de las soluciones hemos seguido lo indicado por la librería para poder trabajar con ella. Hemos creado una clase *Problema* que herede de la clase *EO* parametrizado para que el objetivo sea minimizar el coste de la solución (eoMinimizingFitness). En la clase creada, hemos añadido un vector de la *STL* que guarde la información de a que unidad se la asigna que localización. Es decir, si a la unidad 1 se la localización 3 entonces en el vector, el elemento 1 contendrá el valor 3.



Además En esta clase hemos tenido que re-definir los métodos esenciales para que la API pueda trabajar sobre ella. Estos son:

- El constructor por defecto y el constructor de copia
- El destructor de la clase
- El operador de asignación (=)
- El operador [ ] para acceder a un elemento de la solución
- El método *create*() para crear una solución
- El método *evaluate*() para devolver el coste de la solución

Además, por mi parte, el operador << para que pueda sacarte por pantalla la tupla solución y su coste.

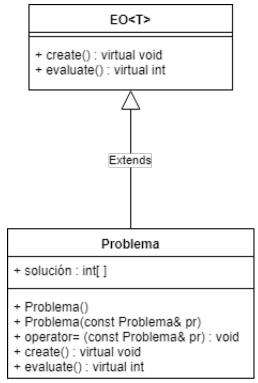


Figura 1: Diagrama ilustrativo de la clase solución

2.2. Función objetivo

La función objetivo del problema sigue manteniendose en este caso. A continuación se muestra el cálculo de esta en pseudocódigo:

Algorithm 1 Calculo de la función objetivo

procedure FUNCIÓNOBJETIVO
    $coste \leftarrow 0$

    for i := 0 **in** Tamaño del Problema **do**
        for j := 0 **in** Tamaño del Problema **do**
            $coste \leftarrow coste + flujo(i)(j) * distancia(\pi(i))(\pi(j))$
        end for
    end for

    return coste
end procedure

2.3. Operadores Genéticos

2.3.1. Generación de población inicial

En el esquema de *ParadisEO*, para inicializar la clase tenemos que crear una clase nueva, en esta caso se llama *ProblemaInit*, que hereda de la clase *eoInit*. Lo único que hay que hacer es sobrecargar el operador *( )* para que este llame al método *create( )* de la instancia de la clase Problema. Este realiza lo siguiente:

Algorithm 2 Creación de la población inicial

procedure CREATE
    for i := 0 **in** Tamaño del Problema **do**
        $Solucion(i) \leftarrow i$
    end for

    for i := 0 **in** Tamaño del Problema **do**
        $IndiceAleatorio \leftarrow GeneraAleatorioEntre(i, TamProblema)$
        $Swap(Solucion(i), Solucion(IndiceAleatorio))$
    end for
end procedure

2.3.2. Mecanismo de selección

El mecanismo de selección está implementado de por si en la biblioteca. Seleccionas entre una abanico de tipos de selección y lo parametrizas esperando que se ajuste totalmente a tu problema.

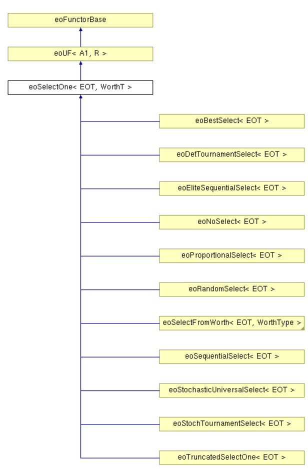


Figura 2: Esquema de clases del mecanismo de Selección

En el caso de la práctica hemos utilizado $eoDetTournamentSelect$, que permite dado un tamaño de torneo elegido por el usuario, realizar un selección determinista (El mejor siempre tiene más probabilidades) y obtiene UN individuo (Como el nombre de la superclase *eoSelectOne* indica). Si le ponemos un tamaño de torneo = 2, entonces se ajusta a lo comentado en la práctica que indica que debe ser binario.

Para seleccionar varios individuos no haría falta pasarle varias veces el mecanismo de selección o modificarlo para ello. Para ello, la librería proporciona una nueva clase que realiza varias selecciones con el motor que se le pasa del tipo $eoSelectOne$.

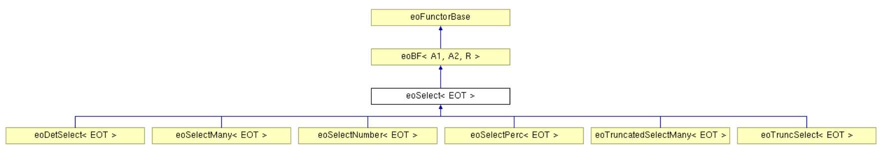


Figura 3: Esquema de clases del mecanismo de Selección de varios individuos

En el caso del AGG escogeremos $eoSelectPerc$ pues es más fácil indicarle un porcentaje del 1,0 que haremos un torneo por cada individuo; y en el caso del AGE haremos uso del $eoSelectNumber$ pasandole 2 como número de individuos.

2.3.3. Operador de Cruce

Para la implementación del operador de cruce, tenemos nuevamente que crear una clase que extienda a la clase de la biblioteca $eoQuadOP$ (de *Quadratic Operator* porque opera con dos elementos) y sobrecargar el operador () para que realice el operador de cruce deseado.En esta práctica contamos con dos operadores de cruce: por *Posicion* y el *Ordered Crossover* (OX). Se detallan en pseudocódigo a continuación:

Algorithm 3 Operador de cruce por Posición

```
procedure CrucePorPosicion(problema1, problema2)
    indiceSinAsignar ← 0
    for i := 0 in Tamaño del Problema do
        if p1:solucion(i) ≠ p2.solucion(i) then
            PosicionSinAsignar(indiceSinAsignar) ← i
            elementosRestantes(indiceSinAsignar) ← p1 : solucion(i)
            indiceSinAsignar ← indiceSinAsignar + 1
        end if
    end for

    for i := 0 in indiceSinAsignar do
        IndiceAleatorio ← GeneraAleatorioEntre(i, TamProblema)
        Swap(elementosRestantes(i), elementosRestantes(IndiceAleatorio)
    end for

    for i := 0 in indiceSinAsignar do
        p1 : solucion(PosicionSinAsignar(i)) ← elementosRestantes(i)
    end for
end procedure
```

Algorithm 4 Operador de cruce OX

```
procedure OX(problema1, problema2)
    crearSubcadenaAleatoria(Izquierda, Derecha)
    for i in Subcadena do
        hijo(i) ← p1 : solucion(i)
    end for

    indiceReal ← 0

    for i := 0 in Tamaño del Problema do
        if indiceReal = izquierda then
            indiceReal ← derecha + 1
        end if

        if p2:solucion(i) no esta en hijo then
            hijo(indiceReal) ← p2 : solucion(i)
            indiceReal ← indiceReal + 1
        end if
    end for
end procedure
```

Aquí se indica para crear solo un hijo. Para crear otro basta con repetir el proceso

2.3.4. Operador de mutación

De nuevo, para implementar el operador de mutación, $ParadisEO$ nos obliga a crear una nueva clase (en mi caso, *ProblemaSwapMutacion* que herede de $eoMonOP$ y sobrecargar el operador ().El operador de mutación se basa en el de intercambio que se hizo para la búsqueda local. La descripción es muy simple:

Algorithm 5 Operador de mutación

```
procedure OPERADORMUTACION(problema)
    i ← GeneraAleatorio(0, TamProblema)
    j ← GeneraAleatorio(0, TamProblema)

    while i = j do
        j ← GeneraAleatorio(0, TamProblema)
    end while

    swap(problema : solucion(i), problema : solucion(j))
end procedure
```

3. Estructura del método de búsqueda

3.1. Estructura base

La base para la estructura de la búsqueda para los algoritmos géneticos es la misma. De la forma en la que he desarrollado mi práctica, $ParadisEO$ se encarga de realizar tal tarea con un clase llamada $eoEasyEA$ (*EasyEvolvingAlgorithm*). Con una llamada al operador () de la instancia de clase, ejecutará el algoritmo de su forma propia. En la documentación se especifica que el algoritmo hace lo siguiente:

Algorithm 6 Algoritmo Génetico

```
procedure AG(poblacion)
    while not Condición de parada do
        Sucesores ← Reproducir(poblacion)
        Evaluar(poblacion, sucesores)
        Reemplazar(poblacion, sucesores)
    end while
end procedure
```

3.2. Componentes específicos

3.2.1. Algoritmos genéticos

Sobre el esquema de evolución ya hemos discutido y cómo funciona a nivel de implementación en el apartado sobre el operador de selección, así que nos centraremos en el de reemplazamiento.

Al igual que en el operador de selección, para el operador de reemplazo se utiliza directamente clases de la librería que representan ya los operadores de reemplazo más comunes.

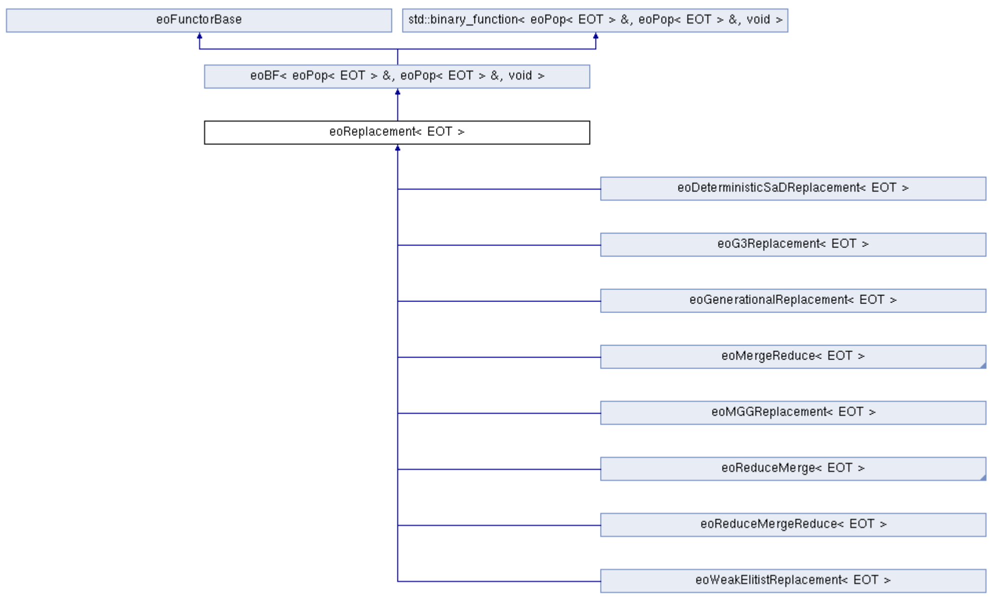


Figura 4: Esquema de clases para el operador de reemplazo

En el caso del AGG he hecho uso de la clase *eoDeterministicSaDReplacement* que nos permite hacer un torneo del tamaño que queramos con la población antigua y la nueva. Es decir, nos permite cuantos de la nueva población sobrevivirán, cuantos de la vieja también lo harán y luego los somete a un torneo para reducir el tamaño de población al de la original (para conservar el tamaño de población a traves de la ejecución del algoritmo). Esto nos da la posibilidad de "matar" a todos los individuos de la población anterior menos el mejor y someterlo a torneo con la población nueva. Si el mejor de la antigua es mejor que el peor de la nueva, lo substituye. El algoritmo en sí es más complejo, para posibilitar mayor flexibilidad, pero para este caso podemos simplificarlo a esto.

Por otro lado, respecto al AGE, se simplifican mucho las cosas al no tener que enfrentar 2 individuos específicos. Para este algoritmo, he utilizado la clase *eoPlusReplacement* que hereda de *eoMergeReduce*. La acción que realiza es la de unir la dos poblaciones, y reducirlas (quitando las peores). Esto permite en nuestro caso, "sumar" los dos hijos generados a los padres, y quitarse los dos peores (porque reduce al tamaño original), preservando así el elitismo y las dos padres en caso de que sean mejor que los hijos.

3.2.2. Algoritmo Meméticos

La integración del algoritmo genético y la búsqueda local en *ParadisEO* es un poco extraña pero directa. Por una lado creamos nuestro algoritmo génetico de manera que hemos especificado en esta documentación, y por otro lado creamos nuestro algoritmo de búsqueda local que aplicaremos sobre cada cromosoma. Para integrar no se hace a nivel de clase o algo similar, si no que se realiza directamente en el programa principal. Se explica en pseudocódigo:

Algorithm 7 Algoritmo Memético

procedure AM
  $AG \leftarrow crearAG()$
  $BL \leftarrow crearBL()$

  $poblacion \leftarrow crearPoblacion(individuoOriginal)$
  $iteracionesTotales \leftarrow 0$

  while iteracionesTotales < 50000 **do**
    $AG(poblacion)$

    for cromosoma **in** poblacion **do**
      $BL(cromosoma)$
      $iteracionesTotales \leftarrow iteracionestotales + iteracionesBL$
    end for

    $iteracionesTotales \leftarrow iteracionesTotales + 10$
  end while
end procedure

4. Procedimiento considerado para desarrollar la práctica

Como se ha indicado varias veces a lo largo de esta documentación, hemos usado el framework *ParadisEO* para desarrollar la práctica. *ParadisEO* es un API formada por varios módulos interconectados entre sí, para el lenguaje $C++$ y pensada para problemas de optimización. Para ello incluye diferentes tipos de algoritmos, desde Búsquedas Locales como el *HillClimbing* o la *TabuSearch* hasta algoritmos para optimización multiobjetivo, pasando por algoritmos géneticos y basados en poblaciones. Hasta incluye un módulo para el cálculo en paralelo de los algoritmos.

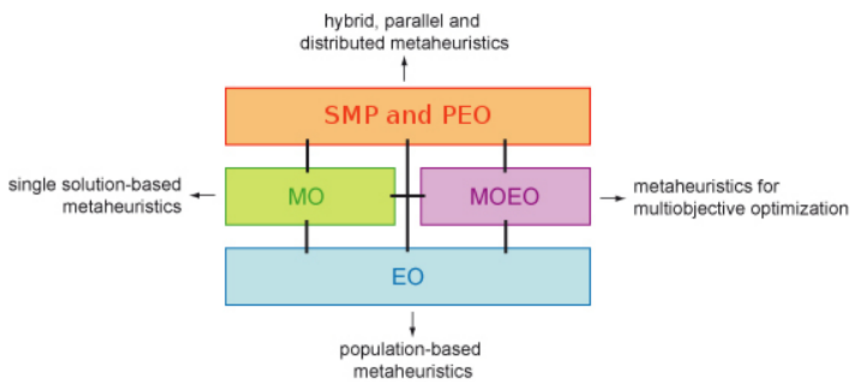


Figura 5: Los diferentes módulos de *ParadisEO*

Por tanto, para desarrollar la práctica, necesitaremos instalar la librería. Para la práctica he hecho uso de las características de multiprocesamiento, así que seria necesario instalar también el paquete *smp*. Se deja enlances que indican como descargar la librería y como instalarla:

Descargar *ParadisEO*

Tutorial de como instalar *ParadisEO*

La estructura de ficheros es la siguiente:

- PaqueteAlgoritmo
  - src
  - paradiseoFinder
  - bin
    - Instancias
  - data
    - resultados
  - build

En cuestión de como ejecutar la práctica desarrollada es sencillo:

1. Entramos en el paquete deseado (Carpeta AGG, AGE o AM)
2. Entramos en la carpeta *build* y ejecutamos *cmake..*
3. Se generará un makefile, ejecutamos *make*
4. ¡Binarios listos!

Para ejecutar el programa basta la siguiente sintaxis (dentro del fichero *bin*):

./nombreDelPrograma ./Instancias/archivoDatos semilla

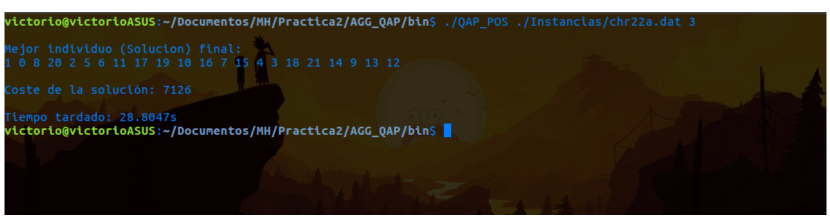


Figura 6: Ejemplo de ejecución

5. Análisis de resultados

Las semillas usadas y los resultados obtenidos son los siguientes:

Cuadro 1: Valores de las semillas

Algoritmo	**Valor Semilla**
AGG POS	4
AGG OX	4
AGE POS	5
AGE OX	5
AM (10,1.0)	7
AM (10,0.1)	7
AM (10,0.1 Mejores)	7

Cuadro 2: Tiempos y Desviación estándar

Algoritmo	**Desv**	**Tiempo**
AGG POS	13,3556	272,8610
AGG OX	12,2186	173,0149
AGE POS	33,2056	33,9220
AGE OX	10,3071	215,2629
AM (10,1.0)	5,0844	241,9511
AM (10,0.1)	5,7730	108,4534
AM (10,0.1 Mejores)	8,7086	89,2970

A partir de estos datos se puede deducir que los algoritmos genéticos funcionan realmente bien con sistemas complejos de datos. Tantos los AGG como los AGE tienen que un rendimiento decente (menos el AGE POS) que como vemos, con una media de menos de 4-5 minutos puede dar soluciones muy aceptables. Pero vamos a profundizar un poco en el tema.

Los algoritmos genéticos generacionales han dado unos resultados estándar. Han consumido una media de tiempo bastante considerable pero a su vez, para los resultados obtenidos, ese margen de tiempo no es comparable a algoritmos clásicos deterministas. Sin embargo, la fiabilidad de estos algoritmos es pseudo-aleatoria, así pues, pueden dar resultados muy buenos o no conseguir profundizar lo suficiente y quedarse estancados en soluciones "solamente" buenas.

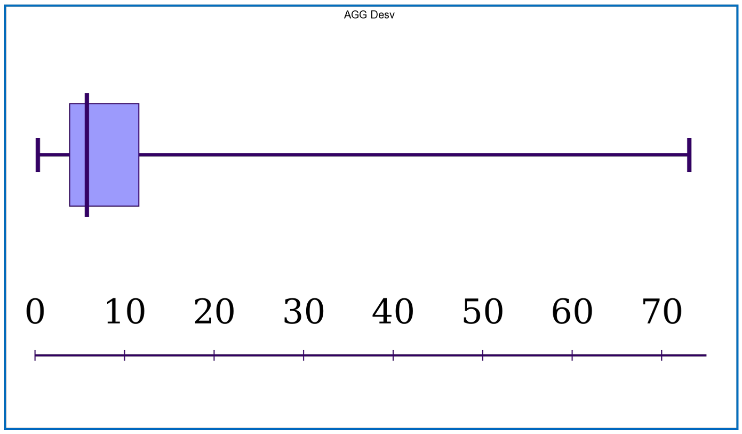


Figura 7: Aqui podemos ver el diagrama de cajas del *AGG POS* sobre la desviación estándar. Podemos ver que la aleatoriedad del algoritmo puede llegar a proporcionarnos desviaciones de hasta 70.

Para ver como realmente puede afectar tal aleatoriedad nos podemos fijar en el caso del AGE POS, donde ha dado resultados muy distantes en la desviación típica. Esto puede darse debido a que a veces, una semilla o una estado inicial, puede ser beneficioso para algunos caso, pero totalmente perjudicial para otros, dando resultados medios bastante pésimos para el tipo de algoritmo, aún buenos en general.

Por otro lado, están los algoritmos Meméticos que solucionan el problema de convergencia que tienen los algoritmos genéticos. En este tipo de algoritmos existe un equilibrio: los algoritmos genéticos introducen diversidad, pues generan soluciones nuevas, aleatorias que pueden llegar a incluirse en nuevas poblaciones; mientras que la búsqueda local añade profundidad, lo que permite llegar a soluciones mejores y converger más deprisa. Además, comparados con los AG, la BL tarda mucho menos porque tiene que generar muchos menos número aleatorios y realizar menos operaciones, por lo que el tiempo de ejecución se reduce significativamente. Aunque no todo son ventajas. El algoritmo memético se dirige a una dirección del espacio de soluciones, es muy difícil que salga de es rumbo. Por tanto, adquiere una de las desventajas de la búsqueda local y es que no consigue a veces salir de óptimos locales.

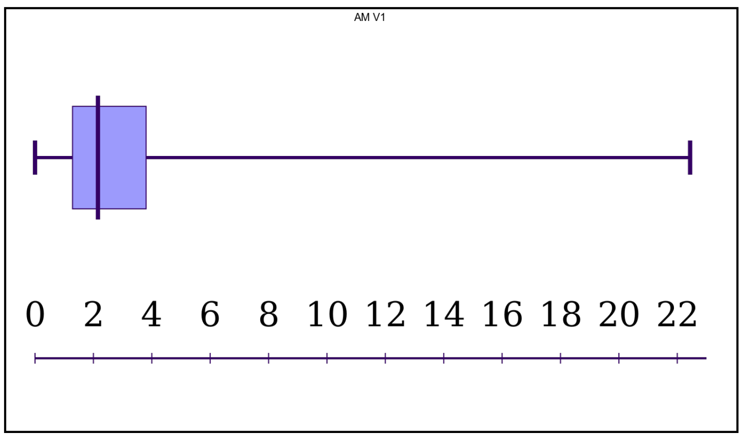


Figura 8: En este caso del AM, podemos observar que los resultados son mucho más uniformes, y que las soluciones se suelen acercar al óptimo de manera más próxima.

Referencias

[1] El-Ghazali Talbi: METAHEURISTICS FROM DESIGN TO IMPLEMENTATION. Hoboken, New Jersey, 2009.

[2] Paradiseo: A software framework for metaheuristics, `http://paradiseo.gforge.inria.fr/index.php?n=Doc.API`