Práctica de algoritmos evolutivos

Problemas de optimización combinatoria
QAP

UNIVERSIDAD
DE GRANADA

Inteligencia Computacional

2018/2019

Sergio Samaniego Martínez

Master en Ingeniería Informática

0. ÍNDICE

Contenido

1. INTRODUCCIÓN

El problema de la asignación cuadrática o QAP [Quadratic Assignment Problem] es un problema fundamental de optimización combinatoria con numerosas aplicaciones. El problema se puede describir de la siguiente forma:

Supongamos que queremos decidir dónde construir n instalaciones (p.ej. fábricas) y tenemos n posibles localizaciones en las que podemos construir dichas instalaciones. Conocemos las distancias que hay entre cada par de instalaciones y también el flujo de materiales que ha de existir entre las distintas instalaciones (p.ej. la cantidad de suministros que deben transportarse de una fábrica a otra). El problema consiste en decidir dónde construir cada instalación de forma que se minimice el coste de transporte de materiales.

Formalmente, si llamamos d(i,j) a la distancia de la localización i a la localización j y w(i,j) al peso asociado al flujo de materiales que ha de transportarse de la instalación i a la instalación j, hemos de encontrar la asignación de instalaciones a localizaciones que minimice la función de coste:

$$\sum_{i,j} w(i,j)d(p(i),p(j))$$

donde p() define una permutación sobre el conjunto de instalaciones.

Igual que en el problema del viajante de comercio, que se puede considerar un caso particular del QAP, una solución para el problema es una permutación del conjunto de instalaciones que indica dónde se debe construir cada una.

2. IMPLEMENTACIÓN

2.1 Algoritmo estándar

La primera versión que estudiaremos será la de un algoritmo genético estándar.

Para comenzar lo primero que haremos será probar distintas combinaciones de los parámetros de entrada y finalmente nos quedaremos con los siguientes parámetros:

- Número máximo de evaluaciones: 50.000
- Tamaño de la población: 50
- Probabilidad del cruce: 0,5
- Probabilidad de mutación: 0,5

Conocidos estos parámetros, pasaremos a calcular el número esperado de cruces y de mutaciones, seleccionando un número aleatorio de la siguiente forma:

```
int n_esperado_cruces = Math.round(prob_cruce * (tamanio_poblacion/2));
int n_esperado_mutaciones = Math.round(prob_mutacion * tamanio_poblacion * n  );
```

Obtenidos estos valores, comenzaremos generando la población inicial.

Esta población inical será una población seleccionada de forma aleatoria. Es decir, tendremos un array de vectores, donde cada vector será una solución aleatoria que tendrá los números de 0 al tamaño de la población, ordenados de forma aleatoria.

Esto podemos observarlo en la siguiente imagen:

```
/**
 * @param p_actual
 * @return población inicial
 */
private ArrayList<ArrayList<Integer>> generaPoblacionInicial(ArrayList<ArrayList<Integer>> p_actual) {
    ArrayList<ArrayList<Integer>> salida;
    for(int i=0; i < p_actual.size(); i++) {
        P.add(generaSolucionAleatoria());
    }

    salida = P;

    return salida;
}

/**
 * @return Array con una solución aleatoria
 */
private ArrayList<Integer> generaSolucionAleatoria() {
    ArrayList<Integer> resultado = new ArrayList();

    for(int i=0; i<n; i++) {
        resultado.add(i);

        Collections.shuffle(resultado);
    }

    return resultado;
}
```

Obtenida la población inicial, la evaluaremos para ver cuál de ellas es la mejor solución y el coste asociado a la misma.

```
 /**
  * @param p_actual
  * @param indice
  * @param contador
  * @return Obtenemos la solución con menor coste
  */
 private ArrayList<Integer> mejorSolucion(ArrayList<ArrayList<Integer>> p_actual, int indice, int contador) {
     ArrayList<Integer> resultado = new ArrayList();
     resultado = new ArrayList<>(p_actual.get(0));
     int mejor_costo = evaluaSolucion(resultado), costo_actual=0;
     this.contador++;

     for(int i=1; i< p_actual.size(); i++) {
         costo_actual = evaluaSolucion(p_actual.get(i));
         this.contador++;

         if(costo_actual < mejor_costo) {
             resultado = new ArrayList<>(p_actual.get(i));
             mejor_costo = costo_actual;
             indice = i;
         }
     }


     return resultado;

 }
```

Lo único que haremos será ir evaluando cada una de las soluciones aleatorias de la población inicial y calculando el coste de la misma, quedándonos con la que menor coste tenga.

Una vez hecho esto, tendremos una solución inicial con un coste asociado, y una población inicial. Ahora debemos ir avanzando generaciones y ver si conseguimos mejorar esta solución.

De la población inicial, vamos a pasar a seleccionar los padres para la siguiente generación, los cuales van a ser seleccionados de forma aleatoria entre toda la población existente.

Una vez seleccionados los padres, haremos el cruce entre ellos teniendo en cuenta los padres seleccionados en el paso anterior y el número esperado de cruces que calculamos al inicio.

```
/**
 * @param p_padres
 * @param n_esperado_cruces
 * @return Resultado del cruce de los padres
 */
private ArrayList<ArrayList<Integer>> crucePadres(ArrayList<ArrayList<Integer>> p_padres, int n_esperado_cruces) {

    ArrayList<ArrayList<Integer>> resultado = new ArrayList();
    int k = 0;

    for(int i=0; i<p_padres.size(); i++) {
        resultado.add(new ArrayList());
    }


        for(int i=0; i<n_esperado_cruces; i+=2) {
            resultado.set(k, crucePosicion(p_padres.get(i), p_padres.get(i+1)));
            k++;

            if(evaluaSolucion(p_padres.get(i)) < evaluaSolucion(p_padres.get(i+1))) {
                resultado.set(k, p_padres.get(i));
            }
            else {
                resultado.set(k, p_padres.get(i+1));
            }

            k++;
        }
    for(int i=k; i < p_padres.size(); i++) {
        resultado.set(i, p_padres.get(i));
    }

    return resultado;
}
```

En este paso lo que hacemos es de la lista de padres que tenemos, vamos a ir seleccionándolos por parejas y cruzándolos entre ellos, de forma que, de cada par de padres, obtendremos un individuo que es combinación de ambos y el segundo individuo que será el padre que tenga menor coste en su evaluación.

De esta forma lo que tenemos es la siguiente generación, pero queda un paso importante que realizamos a continuación.

Este paso es el de mutación, donde tendremos que tener en cuenta la población generada y el número esperado de mutaciones calculado al inicio.

La mutación es simple, únicamente cambiaremos los valores de forma aleatoria, de esta manera podremos mejorar o no el coste de la solución.

Obtenida la siguiente generación, volveremos a evaluarla. En este caso calcularemos la peor solución primero y el coste de la misma.

De esta forma tendremos el mejor coste de la generación anterior y el peor de la generación actual.

Entonces, si mejor coste de la generación anterior es menor que el peor coste de la solución actual, añadiremos esta solución de la generación anterior a la generación actual, cambiándolo por la solución con peor coste.

```
peor_solucion = peorSolucion(P_actual, i, contador);
mayor_costo = evaluaSolucion(peor_solucion);

contador++;

if(menor_costo < mayor_costo) {
    P_actual.set(i, mejor_solucion);
}
```

Hecho esto, tendremos una nueva población actual la cual volveremos a calcular de nuevo el mejor coste y la mejor solución.

Este proceso por lo tanto se irá realizando hasta llegar al número máximo de evaluaciones.

2.2 Variante de Baldwin

La variante de Baldwin de nuestro algoritmo genético estándar es una variante que nos tendrá técnicas de optimización local, con heurística Greedy que va a permitir dotar a los individuos de una capacidad de “aprendizaje”.

Esta variante evalúa los posibles fitness de cada individuo, haciendo mutaciones sobre él hasta llegar a un óptimo local, sin embargo, a la siguiente generación pasa el individuo sin modificar, es decir, pasamos al individuo sin las “mejoras” aprendidas.

Como este algoritmos es simplemente una variante del algoritmo estándar, el funcionamiento es el mismo pero cambiaremos únicamente un apartado, en el que se evalúa el fitness del individuo.

```
mejor_solucion = mejorSolucionBaldwin(P_actual, indice, contador);
menor_costo = evaluaSolucion(mejor_solucion);
```

Como vemos, tenemos un nuevo método para obtener la mejor solución, que es “mejorSolucionBaldwin”.

```java
/**
 * @param p_actual
 * @param indice2
 * @param contador2
 * @return mejor solución sin incluir mejoras
 */
private ArrayList<Integer> mejorSolucionBaldwin(ArrayList<ArrayList<Integer>> p_actual, int indice2,
        int contador2) {

    ArrayList<Integer> resultado = new ArrayList();
    resultado = new ArrayList<>(p_actual.get(0));
    int mejor_costo = evaluaSolucion(resultado), costo_actual=0;
    int cont = 0;
    int index = 0;
    this.contador++;

    for(int i=1; i< p_actual.size(); i++) {
        costo_actual = evaluaSolucion(p_actual.get(i));
        this.contador++;

        while(costo_actual > mejor_costo && cont < resultado.size()) {
            resultado = new ArrayList<>(p_actual.get(i) );
            Collections.shuffle(resultado);

            costo_actual = evaluaSolucion(resultado);
            cont++;
        }

        if(costo_actual < mejor_costo) {
            mejor_costo = costo_actual;
            index = i;
        }
    }


    return p_actual.get(index);
}
```

Este es el método en el que pasamos a calcular la mejor solución con la variante de Baldwin.

Este método recibirá como parámetros la población actual y unos contadores.

Por lo tanto, lo que haremos será los siguientes pasos:

- En el primer bucle “for” recorreremos todos los individuos de la población.
- Evaluaremos el coste inical del primer individuo
- Sobre el individuo seleccionado iremos haciendo mutaciones hasta conseguir un coste menor que el inicial o bien hasta llegar a un máximo de iteraciones.

Estos pasos se irán realizando para cada individuo de la población de forma que consigamos el índice del individuo que nos permita tener un menor coste.

Y finalmente devolveremos el individuo que nos permitía tener ese coste menor pero sin modificar.

Este algoritmo, nos permite calcular los hijos potencialmente mejores. Digo potencialmente, ya que inicialmente, estos hijos no tienen ese fitness calculado, de forma que al realizarle las mutaciones, puede que mejoren, pero también puede darse el caso de que estos hijos no mejoren el fitness en las siguientes generaciones.

2.3 Variante de Lamarck

Esta variante del algoritmo genético estándar se le llama variante de Lamarck. Esta variante utilizará técnicas de optimización local, como algoritmos Greedy que nos permitan dotar a los individuos de aprendizaje.

Con la variante de Lamarck evaluaremos los posibles fitness de cada individuo hasta alcanzar un óptimo local y devolver a la siguiente generación el individuo con mejor fitness y con pasando de generación en generación el "aprendizaje" obtenido.

El algoritmo, igual que los dos anteriores funciona igual, por lo que leyendo el algoritmo estándar entendemos el funcionamiento. Únicamente cambiamos el método utilizado para obtener la mejor solución de cada población como vemos en la siguiente imagen:

```
mejor_solucion = mejorSolucionLamarck(P_actual, indice, contador);
menor_costo = evaluaSolucion(mejor_solucion);
```

Tendremos por lo tanto un nuevo método llamado "mejorSolucionLamarck" que nos permitirá obtener la mejor solución de la población siguiendo la variante de Lamarck.

Este método lo veremos a continuación:

```
private ArrayList<Integer> mejorSolucionLamarck(ArrayList<ArrayList<Integer>> p_actual, int indice2,
        int contador2) {

    ArrayList<Integer> aux = new ArrayList();
    ArrayList<Integer> resultado = new ArrayList();
    resultado = new ArrayList<>(p_actual.get(0));
    int mejor_costo = evaluaSolucion(resultado), costo_actual=0;
    int cont = 0;
    this.contador++;

    for(int i=0; i< p_actual.size(); i++) {
        aux = new ArrayList<>(p_actual.get(i));
        costo_actual = evaluaSolucion(p_actual.get(i));
        this.contador++;

        while(costo_actual > mejor_costo && cont < resultado.size()) {
            aux = new ArrayList<>(p_actual.get(i) );
            Collections.shuffle(aux);

            costo_actual = evaluaSolucion(aux);
            cont++;
        }

        if(costo_actual < mejor_costo) {
            resultado = new ArrayList<>(aux);
            mejor_costo = costo_actual;
            indice = i;
        }
    }


    return resultado;
}
```

En él recibiremos como entrada la población que queremos evaluar, así que para calcular la mejor solución de esta población, seguiremos los siguientes pasos:

- Obtenemos el primer individuo de la población y obtenemos su coste inicial.
- Lo siguiente hacer mutaciones sobre ese individuo
- Haremos mutaciones hasta encontrar una variante del individuo que mejore el fitness de este o bien hasta llegar a un límite de mutaciones.
- Si obtenemos una mutación de este individuo que mejore su fitness la almacenaremos como posible solución.

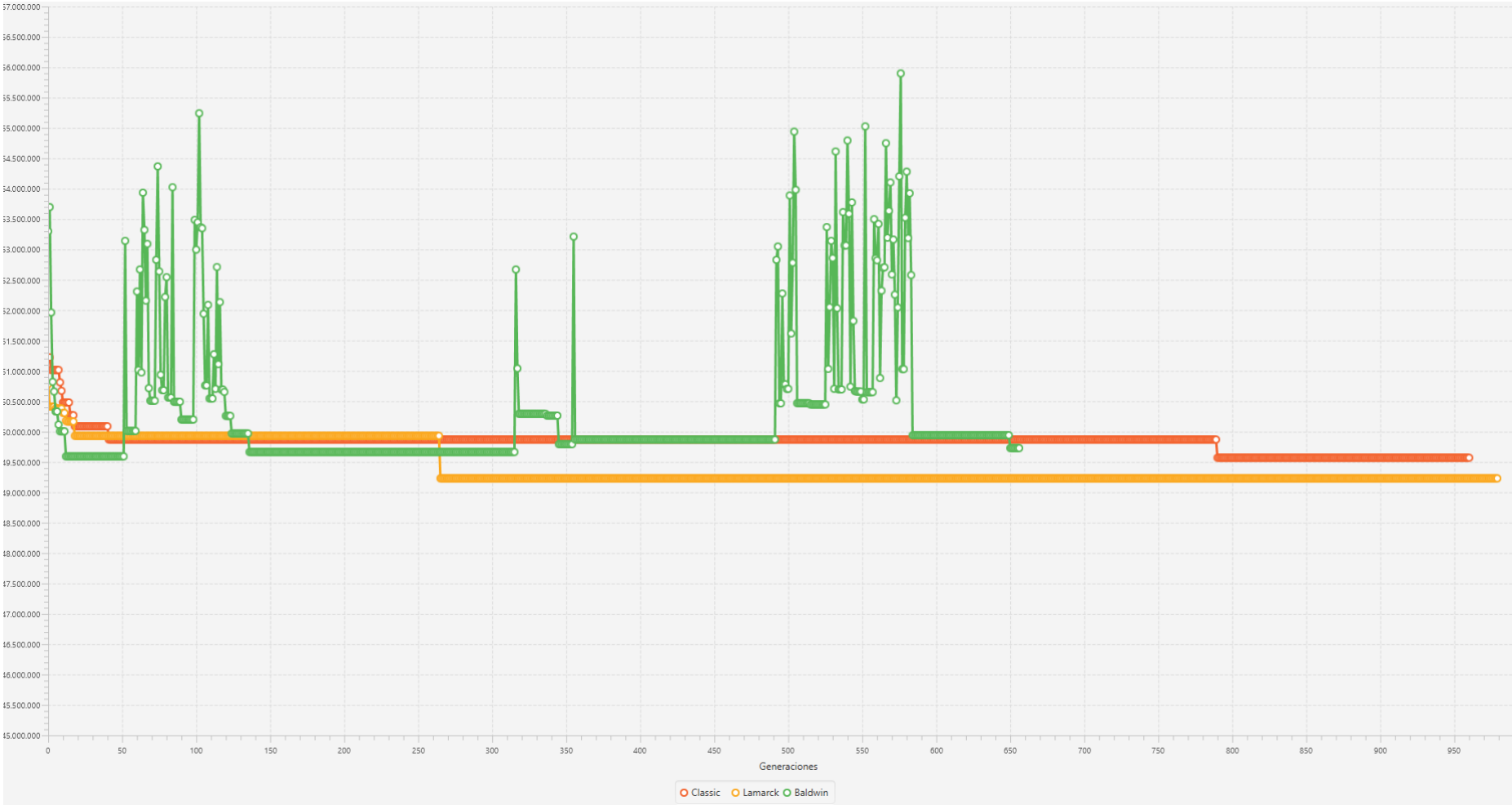
Estos pasos se seguirán para cada uno de los individuos de la población y se devolverá aquel individuo o mutación del mismo que nos devuelva una mejor solución.

En este caso, el algoritmo si dota a los individuos de aprendizaje, es decir, los individuos que pasan a las siguientes generaciones incluyen los rasgos aprendidos, por lo tanto, nos permite ir mejorando la población continuamente.

3. RESULTADOS

En este apartado vamos a ver los resultados obtenidos por nuestro algoritmos, la variante estándar, la variante de Baldwin y la variante de Lamarck.

A continuación se aporta el gráfico comparativo de las tres variantes del algoritmo:


Generaciones
Classic
Lamarck
Baldwin

Como se puede observar la variante de Lamarck nos encuentra la solución bastante rápido ya que se estabiliza alrededor de la generación 260 y ya no varía más.

El algoritmo clásico también funciona bastante bien, teniendo una solución bastante buena, pero tarda mucho más en encontrarla, vemos que la solución final la acaba encontrando sobre la generación 800.

Por otro lado, la variante de Baldwin es la que peores resultados nos da, ya que no consigue estabilizarse demasiado y vemos cómo los fitness varían mucho.

4. CONCLUSIONES

Como se puede observar en el gráfico, la mejor variante del algoritmo vemos que es la variante de Lamarck, esto se debe a que nos permite utilizar óptimos locales para encontrar las posibles alternativas de un individuo y mejorarlo. De esta forma, podemos mejorar las poblaciones mucho más rápido y encontrar una solución óptima con un número menor de generaciones.

En cuanto al coste de tiempo, tenemos que remarcar que tanto la solución de Lamarck como la de Baldwin son algo más lentas, debido a que tiene que usar el algoritmo Greedy para calcular los individuos, pero aun así no suponen un coste demasiado elevado:

Tiempo transcurrido algoritmo estándar-- > 25.543 s

Tiempo transcurrido variante Lamarck -- > 87.613 s

Tiempo transcurrido variante Baldwin -- > 67.209 s

Vemos, que la variante de Lamarck multiplica por tres el tiempo invertido y la de Baldwin por dos, respecto al algoritmo estándar, por lo que es necesario contemplar esto para datos mucho mayores.

En cuanto al algoritmo de Baldwin, es un algoritmo que yo no le encuentro demasiado sentido, ya que en el algoritmo Greedy calculamos los posibles fitness que pueda tener este individuo, pero no los aprende. De esta forma, el individuo puede llegar a tener ese fitness o no, por eso es por lo que creo que hay tantas variaciones de generación en generación, ya que el individuo en sí puede tener un fitness elevado pero como potencialmente puede tener uno mejor, se incluye.

En términos generales, el algoritmo estándar nos permite tener unos resultados bastante buenos con un coste muy bajo, mientras que si queremos refinar más necesitaremos hacer uso de la variante de Lamarck.

5. BIBLIOGRAFÍA

[1] https://elvex.ugr.es/decsai/computational-intelligence/lab/app.pdf

[2] https://elvex.ugr.es/decsai/computational-intelligence/slides/G2%20Genetic%20Algorithms.pdf

[3] https://elvex.ugr.es/decsai/computational-intelligence/lab/qap.2017.pdf