Heuristicos de Búsqueda

Jon Goenetxea

26 de febrero de 2015

Resumen

En este trabajo se presenta una comparativa entre dos diferentes algoritmos de búsqueda heuristica. El primero, denominado *Hill Climbing*, es un algoritmo basado en búsqueda local, que se ha implementado utilizando múltiples inicializaciones (*multi-start*) para tratar de evitar los mínimos locales. Como segundo algoritmo para la comparación se ha escogido un algoritmo genético, que entra dentro de los algoritmos tipificados como poblacionales. En el apartado experimental, se han analizado las posibles configuraciones para la búsqueda local, y se ha hecho una comparación utilizando un test de hipótesis de Wilcoxon.

1. Problema de asignación cuadrática

El problema de la asignación cuadrática, que se denota por sus siglas en inglés *QAP* (*Quadratic assignment problem*), fue planteado como un modelo matemático para un conjunto de actividades económicas indivisibles. Posteriormente fue demostrado que QAP pertenece a los problemas no polinomiales duros, y es aplicable a un sin número de situaciones.

La primera vez que este problema fue planteado fue en 1957 por Kooppmnans y Beckmann. En 1976, Sahni y Gonzalez demostraron que era un problema NP duro .

Este tipo de problemas consisten en escoger para N localizaciones la distribución de N instalaciones que minimice el coste de instalación. Para la toma de decisiones hay que tener en cuenta dos factores principales: 1) el coste de moverse entre localizaciones (distancias, coste de transporte, penalizaciones, etc.), y 2) el flujo de material entre instalaciones.

Para definir estos datos, se definen dos matrices (la matriz de distancias y la matriz de flujos de material) que modelan el comportamiento del sistema de instalaciones. La matriz de distancias se define como $D = [d_{ij}]$, donde d_{ij} representa el coste de transporte de la localización i a la localización j. La matriz de flujos se define como $F = [f_{kl}]$, que representa el flujo de material entre la instalación k a la instalación l.

Cada configuración (una posible ubicación para todas las instalaciones) se representa en forma de permutación (ver ecuación 1).

$$\sigma = (\sigma_1\sigma_2\sigma_3...\sigma_n) \tag{1}$$

La función de coste a minimizar por lo tanto, tendrá en cuenta el coste de transporte dependiendo del flujo entre las instalaciones y la distancia entre las

localizaciones. Esta función se puede modelar como muestra la ecuación 2.

$$f(\sigma) = \sum_{i=1}^{n} \sum_{j=1}^{n} d_{\sigma(i)\sigma(j)} f_{ij} \tag{2}$$

2. Propuestas de solución basadas en búsqueda local

Como se ha presentado anteriormente, el algoritmo implementado para la búsqueda local es el *Hill Climbing*. Es un algoritmo de búsqueda local sencillo. Se basa en analizar los resultados vecinos de una muestra, en busca de un resultado mejor que el actual. Esto se hace iterativamente, por lo que el caer en un óptimo local es muy probable. Para minimizar el caer en óptimos locales, se toman varias medidas.

La primera medida se trata hacen varias reinicializaciones (conocido como *multi start*) utilizando el mejor de los resultados como respuesta final. Dentro de cada iteración, y con el fin de salir de un mínimo local, se generan varias muestras aleatorias y se compara si alguna de ellas es mejor que la candidata actual.

El pseudo-código general para este algoritmo sería el siguiente:

```
Data: cantidad de 'start'-s, función de coste
Result: valor de error mínimo encontrado
for cada 'start' do
    generar permutación inicial con función aleatoria;
    while no se ha encontrado mínimo local do
        escoger candidato;
        while no encontremos mejor vecino do
            calcular nuevo vecino;
            evaluar vecino;
            if nuevo vecino mejora then
                guardar nuevo vecino como actual;
                no se ha encontrado mínimo local;
            end
        end
    end
end
devolver valor de mejor permutación encontrada
```

Algorithm 1: Pseudocódigo de *Hill Climbing* modificado.

Este algoritmo tiene pocos parámetros para configurar, lo que simplifica su configuración. El único parámetro que es necesario definir para su implementación general es la cantidad de inicializaciones a hacer. No obstante, y con el fin de reducir el tiempo de ejecución, se definen varios límites para la ejecución. Estos límites se listan en la tabla 1. Además de esta configuración, es necesario definir el método de calcular los vecindarios, y hay que escoger uno que se adecue al problema.

En este aspecto se han analizado dos sistemas de vecinos diferentes: *Swap* y *2-opt*. El sistema de vecinos *Swap* consiste en generar vecinos cambiando un elemento de la lista con otro. Si intercambiamos un elemento al azar con cada

Parámetro	Valor
Tamaño multistart	15
Número máximo de evaluaciones	150
Número de muestras aleatorias tras mínimo local	3
Número máximo de vecinos analizados	Todos

Cuadro 1: Lista de parámetros de configuración para el algoritmo de búsqueda local.

uno de los demás, generamos un vecindario con n vecinos. El sistema de vecinos *2-opt* consiste en intercambiar dos elementos en la lista de elementos. Si se tienen en cuenta todas las posibles permutaciones, se consigue un vecindario de k elementos, donde k se calcula con la ecuación 3.

$$k = \frac{n * (n-1)}{2} - n \tag{3}$$

En los experimentos, se ha comprobado que el algoritmo *2-Opt* es más efectivo a la hora de acercarse a la solución óptima, pero el tiempo de computo también es significativamente mayor que el del método *Swap*. Por ello, se ha optado por utilizar el método *Swap*, modificando la ejecución para que pare de calcular el vecindario en el momento en el que se encuentre un vecino que mejore el valor actual.

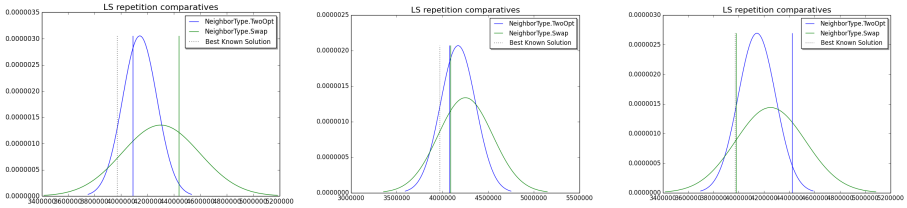

Figura 1: 3 ejecuciones comparativas entre 2-Opt y Swap, con 30 repeticiones cada una y cogiendo los mejores resultados en cada una. La línea vertical representa el valor mínimo para cada algoritmo, y la curva representa la distribución normal teniendo en cuenta la media y la varianza de los mejores resultados de las 30 ejecuciones.

3. Propuestas de solución basadas en algoritmos poblacionales

El algoritmo poblacional implementado es un algoritmo genético. Este algoritmo se basa en las teorías clásicas de la evolución darwiniana como base para generar una serie de iteraciones que generan una población mejor que la generación anterior.

Un algoritmo genético consta de cuatro fases principales: 1) evaluación (evaluation), 2) selección (selection), 3) cruce (crossover) y 4) mutación (mutation). Cada iteración genera una nueva generación después de pasar por todos estos

estados. Se puede poner un criterio de terminación, con el que podemos decidir si la generación actual es ya lo suficientemente buena para el resultado, o si se han utilizado los recursos asignados, por ejemplo. La figura 2 muestra gráficamente la secuencia de estados de una iteración.

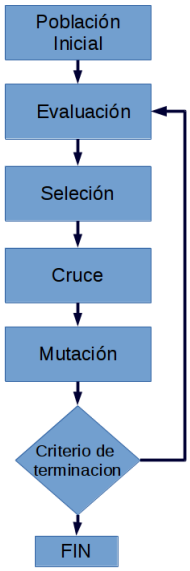

Figura 2: Secuencia de una iteración de un algoritmo genético.

3.1. Fase de Evaluación

En la fase de evaluación se evalúan todos los individuos de la población actual. Esto se hace utilizando la función de coste que se ha expuesto en el apartado 1.

Por lo tanto, cada individuo tendrá asociado el valor de su resultado, dando un indicador de la çalidad"de dicho individuo.

3.2. Fase de Selección

En la fase de selección se escogen los individuos de la población con mejores resultados. Para ello se pueden utilizar diferentes opciones. Entre ellas se ha escogido la *selección por torneo*.

Esta selección se realiza mediante una comparación entre un subconjunto de individuos de tamaño concreto (definido como parámetro), escogidos al azar dentro de la población.

Este tipo de selección se efectúa de manera muy rápida dado que no es necesario evaluar la totalidad de la población. No obstante, hay que definir un nuevo parámetro por lo que añade complejidad a la configuración.

3.3. Fase de Cruce

En esta fase, se unen y combinan los diferentes individuos en el subconjunto de *mejores* individuos. Es necesario encontrar una función de cruce que genere nuevos individuos aprovechando las cualidades de los originales, y tratando de transmitirlas a las nuevas generaciones. Para esta fase, se han barajado varias opciones.

La primera de ellas, combina dos individuos por mitades. Es decir, coge la primera mitad de un individuo A, y la segunda mitad de un individuo B. Los nuevos individuos (denominados A' y B') tendrán la primera parte de A y la segunda de B respectivamente. Para rellenar la mitad restante, lo que se hace es colocar los valores que faltan por incluir con el mismo orden que aparece en el individuo original. Por lo tanto, A' tendrá la primera mitad de A, y los elementos restante en el mismo orden que aparecen en B, y B' tendrá la segunda mitad de B y los demás valore en el mismo orden que aparecen en A.

Como segunda opción, se ha planteado una función en la que se intercambia un elemento al azar entre los dos individuos. Es decir, se escoge un índice al azar (por ejemplo 2) y se coloca el valor en 2 de A en la posición 2 de B, y viceversa. Esta acción por si sola es probable que duplique valores en el nuevo individuo. Para evitar esto, el antiguo valor en la posición 2 se coloca en el lugar en el que está el valor que se va a insertar. La figura 3 representa gráficamente el proceso de cruce descrito.

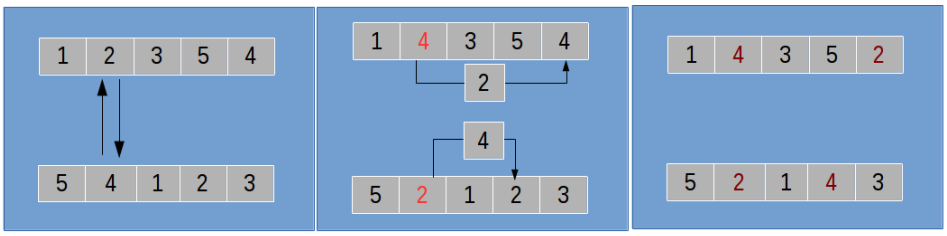

Figura 3: La figura muestra de forma gráfica el proceso de cruce escogido para el algoritmo genético. En el primer paso (izquierda), se seleccionan los elementos con un índice escogido al azar. En el segundo (centro), se intercambia el elemento entre los dos individuos, y se detecta la posición del número que va a ser sustituido. El tercer paso (derecha), se pueden ver los nuevos individuos, y como no tienen repeticiones.

3.4. Fase de Mutación

Para aumentar la variedad de posibilidades, se añade un factor de mutación. Este factor hace que una parte de la población mute (tenga un cambio adicional) en cada iteración. Con esto conseguimos configuraciones que no estuviesen en la generación anterior, y que la fase de cruce no puede generar.

La función de mutación escogida es la de intercambiar 2 elementos del individuo. Para ello, se escogen dos índices al azar, y se intercambian los elementos.

Esta fase añade un elemento a configurar, que es la probabilidad de mutación de un individuo.

Parámetro	Valor
Tamaño población	150
Número de generaciones	15
Factor Mutación	0.05
Tamaño subconjunto torneo	3
Factor de cruce	0.5

Cuadro 2: Lista de parámetros de configuración para el algoritmo genético.

3.5. Configuración final

A diferencia del algoritmo de *Hill Climbing*, este tipo de algoritmos tiene más parámetros que definir. La tabla 2 lista los parámetros escogidos para este algoritmo.

Se ha definido el número de generaciones para que sea el mismo que la cantidad de iteraciones que se realizan en el algoritmo de búsqueda local. En este sentido, tanto el algoritmo de búsqueda local realizará n iteraciones, y el algoritmo genético creará a su vez n generaciones para lograr el resultado final.

4. Experimentación

Para los experimentos se han utilizado 10 sets de datos. Cada uno de estos sets de datos contiene las matrices de distancia y coste para un número diferente de fábricas/localizaciones (10, 20, 30, 40, 50, 60, 70, 80, 90, 100). Se ha lanzado un cálculo utilizando las configuraciones descritas en los apartados anteriores.

El código utilizado para estos experimentos se puede encontrar en la dirección: **https://github.com/argosen/HeuristicSearch**. Se compone de varias clases programadas en Python, utilizando las librerías matemáticas disponibles.

Para calcular el valor final, se han lanzado 15 repeticiones de cada test (algoritmo y set de datos). Con estos datos se ha sacado el valor medio y la varianza de los datos para tratar de minimizar la posible variación de los resultados (ya que no son algoritmos deterministas).

El calculo total ha tardado al rededor de 3 horas en una computadora de potencia media. Los resultados obtenidos se resumen en las tablas 3 y 4.

En las tablas se puede ver la media de los resultados, la varianza y el mejor valor entre las 15 ejecuciones. Estos datos se pueden comparar con los valores de referencia de los datasets, listados en la tabla 5. Con estos datos, podemos comparar el error (aproximado) de cada algoritmo. En la figura 4 podemos ver, que en datasets pequeños el error de los algoritmos genéticos es menor, pero enseguida el error aumenta con el aumento de los elementos a manejar.

Otro indicador que se ha tenido en cuenta a la hora de las mediciones es el tiempo de ejecución. En la figura 5 podemos ver una comparativa de estos valores. Se puede ver que el algoritmo de búsqueda local tarda menos con muestras pequeñas, pero el tiempo aumenta considerablemente a medida que las muestras son mayores.

También se ha realizado un test de hipótesis, utilizando los datos de los experimentos. Los datos utilizados se listan en la tabla 6.

Num Fabricas	Media	Varianza	Mejor Valor
10	5399476.0	199189949810.0	4778938.
20	33523525.0	5.99484002705e+12	29813728.
30	158246124.8	2.30578406503e+13	1.48447493e+08
40	255631830.133	4.59699076885e+13	2.42247900e+08
50	419198346.333	8.60109726003e+13	3.94165092e+08
60	703141309.467	1.14463849179e+14	6.85002147e+08
70	1028622211.93	5.02766108189e+14	9.89590035e+08
80	1363689135.67	3.50846047056e+14	1.33785525e+09
90	1772671426.8	1.02377517371e+15	1.69293611e+09
100	2261489384.33	4.5022765545e+14	2.23168536e+09

Cuadro 3: Resultados para algoritmo de Búsqueda Local.

Num Fabricas	Media	Varianza	Mejor Valor
10	4772991.66667	286804247468.0	3971134.
20	34040886.6667	6.06510712846e+12	28956775.
30	170979720.6	4.28131716062e+13	1.57862222e+08
40	273690367.867	3.14248364801e+13	2.61168249e+08
50	460133962.067	1.08218388337e+14	4.39892249e+08
60	761832148.733	1.8909123247e+14	7.39502280e+08
70	1115317385.8	1.71811158722e+14	1.08579442e+09
80	1446955224.87	3.40196580743e+14	1.41524800e+09
90	1894886012.67	2.72508094515e+14	1.86781759e+09
100	2362333033.0	1.01130826717e+15	2.28023394e+09

Cuadro 4: Resultados para resultados de Algoritmos Genéticos.

Num Fabricas	Valor Mínimo conocido
10	3971134
20	17412718
30	95424438
40	150786632
50	254375088
60	442535568
70	661277026
80	852010960
90	1162218930
100	1431932318

Cuadro 5: Los mejores valores conocidos para cada dataset.

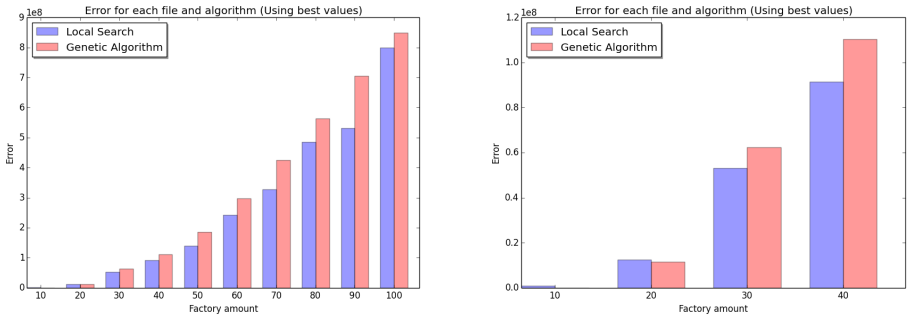

Figura 4: A la izquierda, una representación comparativa de los errores de los algoritmos para todos los data sets. A la derecha, una representación comparativa de los errores de los algoritmos para los primeros 4 data sets.

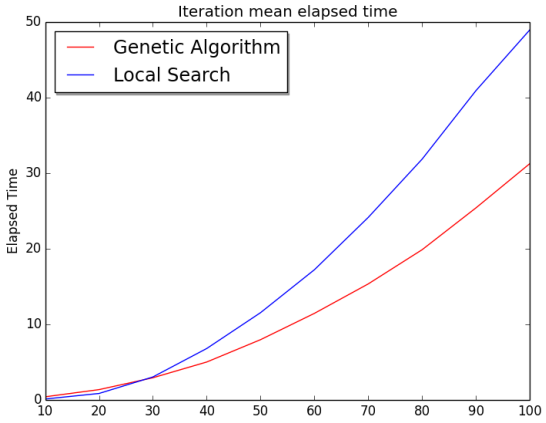

Figura 5: Comparación de los tiempos de computo medios por iteración.

Para el test de hipótesis, se define como H_0 el hecho de que los dos algoritmos son iguales, y se define un 5 % de margen de error. Después de calcular el valor $T = 2{,}0$ utilizando los datos de la tabla, se ha calculado el valor z siguiendo la ecuación 4.

$$z = \frac{T - \frac{1}{4}N(N+1)}{\sqrt{\frac{1}{24}N(N+1)(2N+1)}} \tag{4}$$

El resultado es que $z = 0{,}2038$, por lo que buscando el valor en la tabla de valores ($0{,}5793 => 57{,}93\,\%$), se puede deducir que el 5 % de error se supera con creces, por lo que no se cumple H_0.

LS	GA	diff	Rank
5399476.0	4772991.66667	-626484.333333	2.0
33523525.0	34040886.6667	517361.666667	1.0
158246124.8	170979720.6	12733595.8	3.0
255631830.133	273690367.867	18058537.7333	4.0
419198346.333	460133962.067	40935615.7333	5.0
703141309.467	761832148.733	58690839.2667	6.0
1028622211.93	1115317385.8	86695173.8667	8.0
1363689135.67	1446955224.87	83266089.2	7.0
1772671426.8	1894886012.67	122214585.867	10.0
2261489384.33	2362333033.0	100843648.667	9.0

Cuadro 6: Lista de valores utilizados para el test de Wilcoxon. Las primeras dos columnas listan los datos medios de los resultados. La tercera columna muestra las diferencias entre ellos, y la tercera el ranking de menor a mayor valor absoluto de las diferencias.

5. Conclusiones

Aunque con al configuración propuesta los algoritmos genéticos parecen funcionar mejor con sets de datos pequeños, a medida en que estos aumentan, el error se va haciendo mayor. Aumentando las poblaciones o el número de generaciones los resultados podrían mejorar, pero también aumenta el tiempo de computo, por lo que la optimización de parámetros debe de ser cuidadosa.

Para limitar el tiempo de procesado, se han tenido que acotar tanto la cantidad de vecinos a analizar como la población y cantidad de generaciones, empeorando los resultados de la búsqueda.

Para mejorar los resultados, habría que optimizar los parámetros de los algoritmos por separado.