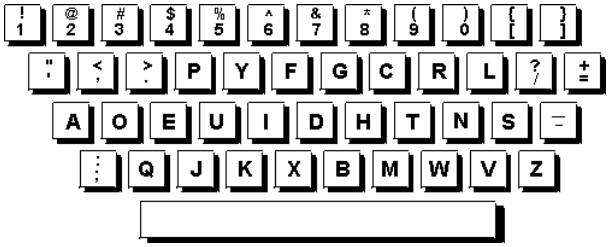
ANÁLISIS Y PROCESADO DE DATOS E INFORMACIÓN - COMPUTACIÓN NUMÉRICA AVANZADA

ANÁLISIS E IMPLEMENTACIÓN DE DISTINTAS META-HEURÍSTICAS PARA EL PROBLEMA DE LA ASIGNACIÓN CUADRÁTICA (QAP)

Junio, 2018



David Lilue Borrero
Universidad Politécnica de Madrid
Departamento de Lenguajes y Sistemas Informáticos e
Ingeniería de Softwares

Índice general

INTRODUCCIÓN

Este trabajo tiene como objetivo general describir, analizar y probar distintas heurísticas para resolver el problema de la asignación cuadrática (QAP). Este problema tiene muchas aplicaciones en la vida real y es uno de los problema fundamentales de optimización combinatoria. Una motivación para escoger este problema es su complejidad, dado que involucra dos variables a la solución; como su nombre lo dice. Sus aplicaciones van desde el posicionamiento de componentes electrónicos en una placa hasta la distribución de letras en un teclado, buscando la eficiencia y minimización de costos.

En específico se han trabajado con algoritmos de búsqueda local como un primer acercamiento al problema, estos tienden a tener un buen desempeño, pueden ajustarse e incorporarse a otras heurísticas; y así tener un mejor rendimiento para el proceso de optimización. En las próximas secciones se describe el problema, decisiones de implementación y casos de prueba para comprobar la eficacia del algoritmo.

Hay una sección dedicada a los algoritmos de búsqueda local que se utilizaron, para cada uno se realizaron distintas pruebas y se ajustaron para tener el mejor rendimiento, buscando comprobar e implementar los conceptos teóricos que describen cada algoritmo. Estos algoritmos posteriormente serán usados para complementar un algoritmo evolutivo, el cual permite la diversidad de soluciones dado por la naturaleza que lo inspiró. Al final se muestran distintos resultados de manera gráfica y así poder concluir las capacidades de cada algoritmo.

DESCRIPCIÓN DEL PROBLEMA

El problema de la asignación cuadrática, *QAP* por sus siglas en inglés, es uno de los distintos problemas de optimización combinatoria que existen, por no decir uno de los más fundamentales y desafiantes de su clase. El mismo se puede explicar con un ejemplo de la vida real y que se usa a lo largo del desarrollo de este trabajo.

Partiendo de un conjunto de n instalaciones y otro con n localidades, para cada par de instalaciones existe una flujo y cada par de localidades una distancias. El objetivo es diseñar un plano de las instalaciones, asignándolas a distintas localidades y minimizando la suma de las distancias por el respectivo flujo. Esto se puede ver en compañías de manufactura, dado que puede implicar mayor costo y ineficiencia en sus operaciones[7].

Se puede enunciar formalmente usando dos matrices de dimensiones $n \times n$, denominadas $D = (d_{ij})$ y $F = (f_{ij})$, la distancia entre dos localidades y el flujo entre dos instalaciones respectivamente. La idea es encontrar la permutación Π de las instalaciones asignadas que minimice la siguiente suma[8].

$$\min_{\pi} \sum_{ij}^{n} d_{ij} f_{\pi(i)\pi(j)}$$

Lenguaje de programción

Para realizar el trabajo se ha usado `Python` como lenguaje de programación, la razón de esta decisión viene dada porque el mismo provee un nivel de abstracción con un sistema orientado a objetos y a su vez permite tener ejecuciones con tiempo relativamente eficientes. Además de reducir el tiempo de implementación en comparación a otros lenguajes que pueden tener mayor desempeño computacional, como son `C` o `C++`, pero resulta difícil de implementar una solución en un tiempo aceptable y con uso eficiente de memoria.

Esturcturas y representación

La representación del problema y sobretodo de una solución usan `numpy`, una librería para manejar datos numéricos en colecciones como arreglos o matrices, para mantener la información de las distancias entre localidades como una matriz de adyacencia donde cero (0) implica que no hay comunicación entre ellas, sino se especifica la distancia. También se mantiene una matriz con el flujo entre las instalaciones, de manera similar a la representación de las distancias.

Toda esta información es almacenada como atributos de una clase, logrando encapsular el

comportamiento de una solución de forma abstracta. Es de esperar que existen otros atributos como el valor de la función objetivo aplicada a la solución y otro con la permutación de las instalaciones, estos atributos son los propiedades que nos permitirán comparar distintas soluciones e ir minimizando la misma. La clase utilizada es la que se muestra a continuación, sin mostrar distintos métodos que la misma tiene implementados.

```
import numpy as np

class Solution:
    def __init__(self, n, ds, fs):
        self.n = n
        self.permutation = np.array(xrange(1,n+1))
        self.distances = np.array(ds)
        self.flows = np.array(fs)
        self._cost = None
```

Instancias del problema

Para probar el desempeño y proximidad a soluciones óptimas de los diferentes algoritmos se utiliza un conjunto de instancias y soluciones usadas en distintos artículos y autores que se listan en el portal web de QAPLIB[2]. Para este trabajo se usan cuatro instancias con distintas dimensiones y cantidad de flujo entre las instalaciones. Cada una es pasada como entrada a los distintos algoritmos y se intenta ajustar los parámetros para hacer la mayor cantidad de pruebas. Dejando la posibilidad de llegar una conclusión amplia.

Este *dataset* se ha ajustado a un mismo formato para generalizar la lectura de los archivos desde el *script*. Es importante destacar que las matrices escogidas son simétricas y la diagonal principal presenta solamente ceros. Esto se debe a que los casos aplican a la vida real donde las localidades no tiene distancia entre ellas mismas y entre una misma instalación no hay flujo. A continuación se muestra un ejemplo de como sería un archivo de entrada.

```
5

0  1  2  3  1
1  0  1  2  2
2  1  0  1  3
3  2  1  0  4
1  2  3  4  0
```

```
0  5  2  4  1
5  0  3  0  2
2  3  0  0  0
4  0  0  0  5
1  2  0  5  0
```

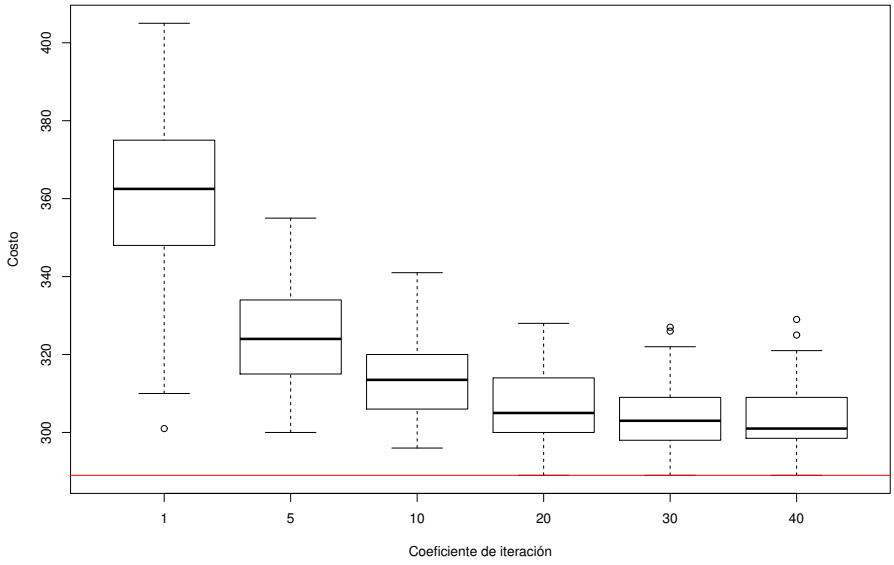
Comenzando con la dimensión de las matrices, ambas cuadradas, siguiendo la matriz de distancias y de segunda la de flujos.

BÚSQUEDA LOCAL

Comenzando con un algoritmo simple de búsqueda local, que son ampliamente usados para resolver problemas de optimización. La implementación que se muestra a continuación altera aleatoriamente una solución (intercambiando instalaciones en distintas localidades) tantas veces con indique un coeficiente que va relacionado con la dimensión de la solución y en caso de conseguir una permutación mejor, esta perdura a la próxima iteración. Sino se regresa a la solución que hasta el momento es la que tiene mínimo costo.

```
def search(sln, iterations_coeff=LOCAL_SEARCH_COEFFICIENT):
    for _ in xrange(0, int(iterations_coeff * sln.n)):
        cities = randomOptions(sln.n, k=2)
        aux_cost = sln.cost

        sln.exchangeFacilities(cities[0], cities[1])

        if aux_cost < sln.cost:
            sln.exchangeFacilities(cities[0], cities[1])

    return sln
```

A continuación se presenta una gráfica comparativa del algoritmo al incrementar el coeficiente de iteración, en específico a una instancia de dimensión $n = 12$, haciendo 100 ejecuciones y comenzando con una permutación aleatoria. Usando un coeficiente de iteración mayor es de esperar que el tiempo de ejecución incrementará pero en todos los casos esta no supera una décima segundo. La instancia pertenece a las que probablemente sean de las más utilizadas. Las mismas fueron propuestas por Nugent, Vollmann y Ruml en 1968[6]. Este caso particular se uso a lo largo del trabajo para probar los distintos algoritmos. En el eje y se indica el costo de la solución encontrada aplicando la función objetivo y en el eje x se pueden ver los distintos coeficientes.



Como se puede observar, el desempeño tiende a mejorar al incrementar el número de iteraciones pero este comportamiento es asintótico. Llegando en algunos casos a conseguir la solución óptima, resaltada con la linea roja. Esto nos ayuda a conseguir la mejor configuración de los parámetros para el algoritmo, que más adelante se usa con un algoritmo evolutivo.

El principal inconveniente con este algoritmo es que tiende a quedarse en mínimos locales porque la perturbación es leve por ello se ha implementado el algoritmo de *simulated annealing* que se enuncia en la próxima sección.

Simulated annealing

Este algoritmo fue propuesto por Koopsmans y Beckmann en 1957[5], posteriormente se aplicó a QAP. Está basado en un proceso usado en metalurgia para alterar las propiedad físicas y químicas de distintos metales. Presentando un algoritmo que tiene un factor de energía o temperatura, que indica la probabilidad de aceptar soluciones que pueden no ser mejores a la actual pero permite tener una búsqueda más global en el espacio de soluciones[8].

La estrategia de enfriamiento es uno de los elementos fundamentales para que el algoritmo se comporte como se espera y encuentre una buena solución, de igual manera la temperatura inicial juega un papel importante para que esta energía mueva la solución en el espacio sin tardar excesivamente en enfriarse.

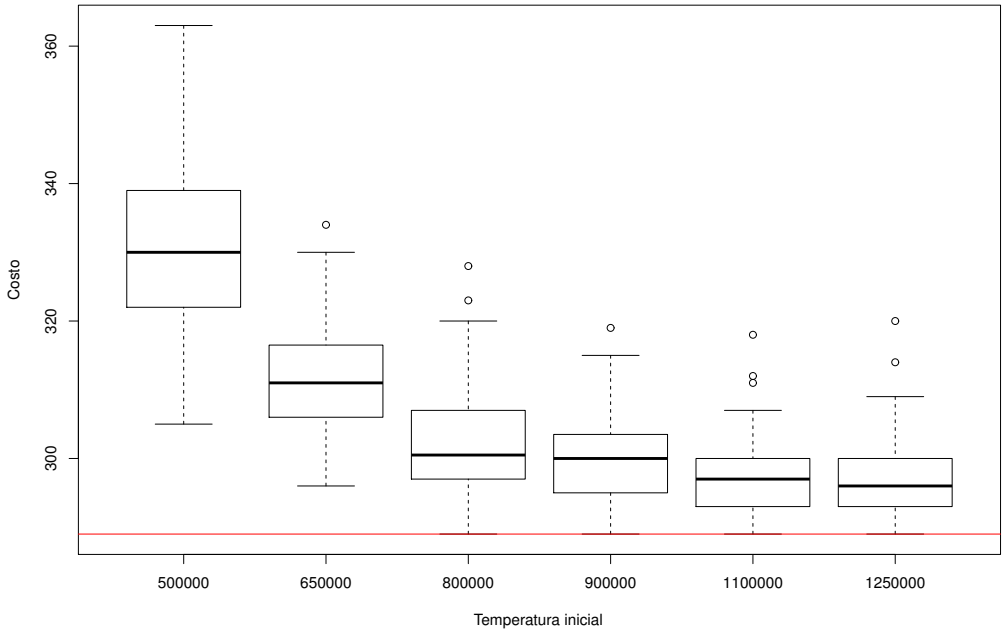
```
def annealing(sln, t_max=ANNEALING_TEMPERATURE_MAX):
    t = t_max
    k = 0.0

    while t > 0.0:
        aux_sln = sln.copy()
        aux_sln.randomize(sln.n if t > sln.n else int(t))

        diff_cost = sln.cost - aux_sln.cost

        if diff_cost > 0 or \
           math.exp(float(diff_cost) / t) > random.uniform(0,1):
            sln.permutation = aux_sln.permutation
            sln.flows = aux_sln.flows
            sln.cost = aux_sln.cost

        del aux_sln
        k += 1.0
        t = math.floor(t_max / (1.0 + MULT_FAC * math.log10(1+k)))
```

La temperatura inicial o máxima desde la que parte el algoritmo es proporcional a la alteración que sufre la solución, por lo que su búsqueda es más global mientras la temperatura sea grande pero eso conlleva a un número mayor de iteraciones. Entonces es complicado encontrar la configuración inicial que se adapte al problema.

La función utilizada como estrategia de enfriamiento está fundamentada en un comportamiento logarítmico[1], esto permite tener valores altos para iteraciones más jóvenes y a lo largo de la ejecución va ir disminuyendo lentamente a 0. La definición formal de la función se puede ver en (1).

$$T_{k+1} = \left\lfloor \frac{T_0}{1 + \alpha \times \log_{10}(1+k)} \right\rfloor, \ \alpha > 1 \tag{1}$$

A continuación se puede ver como se comporta el algoritmo a distintas temperaturas. El tiempo de ejecución es mayor al algoritmo de la sección anterior pero no excede el segundo y medio.



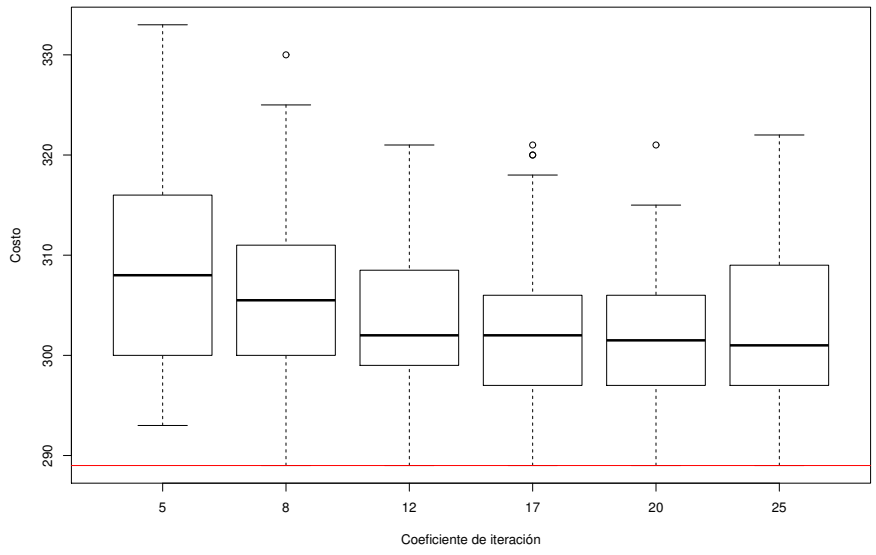
Búsqueda local iterativa

Tratando de mejorar un poco la búsqueda local que se vio antes, se implementa una versión iterativa que sea más ambiciosa, intentando salir de los mínimos locales y así llegar más cerca del óptimo. Manteniendo similitud con el primer algoritmo, este incorpora mayor alteración (factor de mutación) a la solución actual mientras no se encuentra alguna mejor. En caso de encontrar una solución con menor costo, se aplica una recompensa.

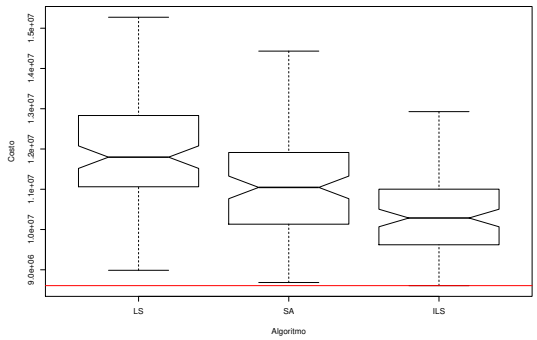
Generalmente este tipo de algoritmos tiene un mejor desempeño pero el tiempo de ejecución es mayor, el objetivo es tener una ganancia considerable sin perder la eficiencia. Incorporar un factor de mutación (relación de la cercanía al final de la ejecución) permite una vista más amplia de la vecindad de una solución, y una ventaja sería tener una exploración que se expande progresivamente. A diferencia de *simulated annealing*, que inicia con amplia exploración y finaliza siendo más recatado.

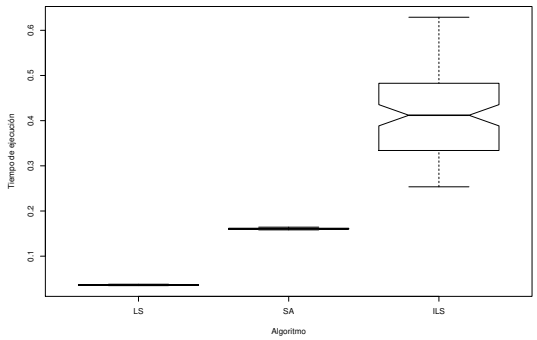
```
def eager_search(sln, iterations_coeff=LOCAL_SEARCH_COEFFICIENT):
    crnt_it = 0.0
    max_it_f = iterations_coeff * sln.n

    while crnt_it < max_it_f:
        crnt_it += 1.0
        mutation_factor = crnt_it / max_it_f
        aux_sln = sln.copy()

        for i in xrange(0, int(math.ceil(mutation_factor * sln.n))):
            aux_sln.randomize()

            if aux_sln.cost < sln.cost:
                sln.permutation = aux_sln.permutation
                sln.flows = aux_sln.flows
                sln.cost = aux_sln.cost
                crnt_it = 0
                break
        del aux_sln
    return sln
```

Como puede verse a continuación, el desempeño es mejor al momento de encontrar la solución óptima pero tener un coeficiente grande implica que las soluciones comienzan a dispersarse más. Dado que este guarda relación con el factor de mutación y eso puede tener ventajas como desventajas.



En los siguientes gráficos se puede ver como se comportan los algoritmos lado a lado con una instancia de dimensiones $n = 19$ propuesta por Elshafei en 1977[3]. A la izquierda, su tanto aproximación a la solución óptima y a la derecha, el tiempo de ejecución.





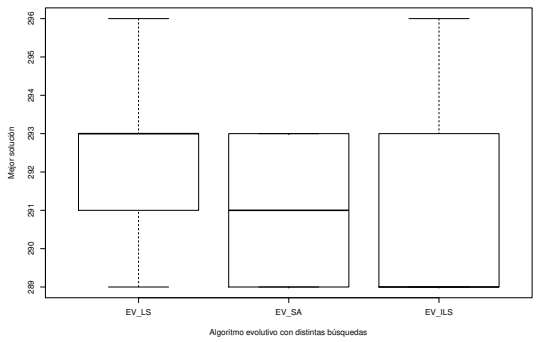
ALGORITMO EVOLUTIVO

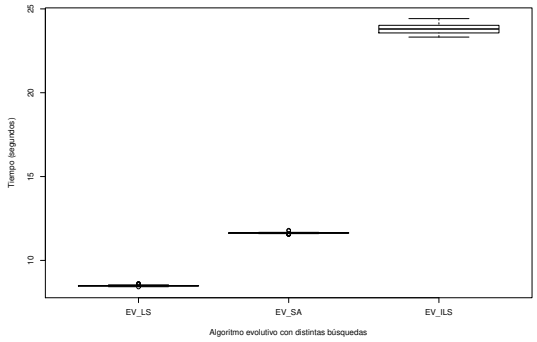
Los algoritmos evolutivos han sido usado en muchas áreas de la optimización combinatoria[7] y en este trabajo se combina con los algoritmos de búsqueda local que se presentaron anteriormente para tener un enfoque más amplio del problema que se está analizando.

Con la ayuda de este tipo de algoritmos se busca la diversidad de soluciones, basándose en el concepto biológico de la evolución. Partiendo de una población inicial, realizando procesos de reproducción y mutación en las distintas generaciones con el objetivo de tener diversidad y escoger a los mejores candidatos para reproducir en cada generación.

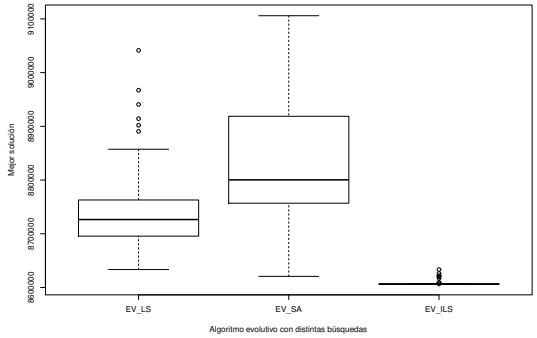
En nuestro caso se ha escogido una población aleatoria inicial de 60 soluciones y se reproducen durante 7 generaciones. Cada par de padres tendrá 3 hijos, a estos se les aplicará una búsqueda local y serán descartados aquellos que tengan las peores cualidades. La idea es ver como se comporta el algoritmo evolutivo con las distintas búsquedas locales con diferentes instancias.

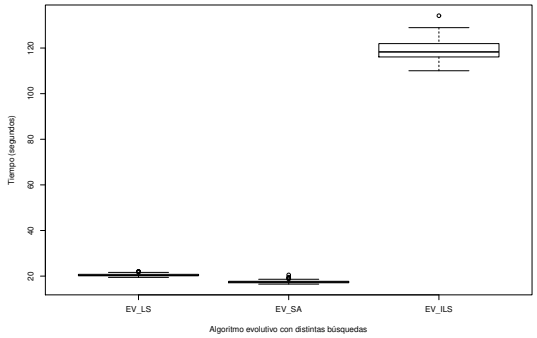
A continuación se puede ver como se comporta el algoritmo genético dependiendo de que búsqueda local se utilice.



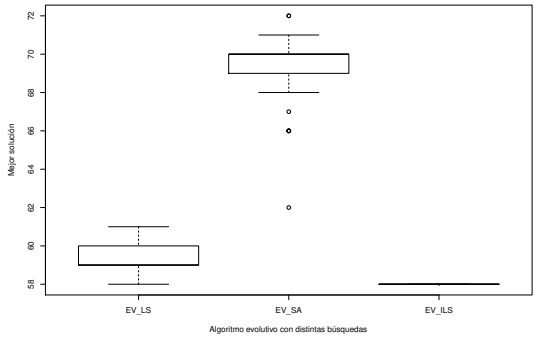


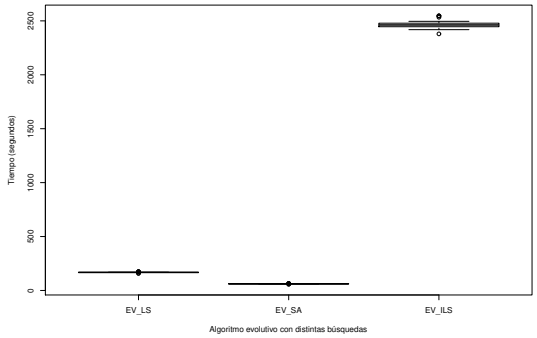
Instancia: nug12[6]





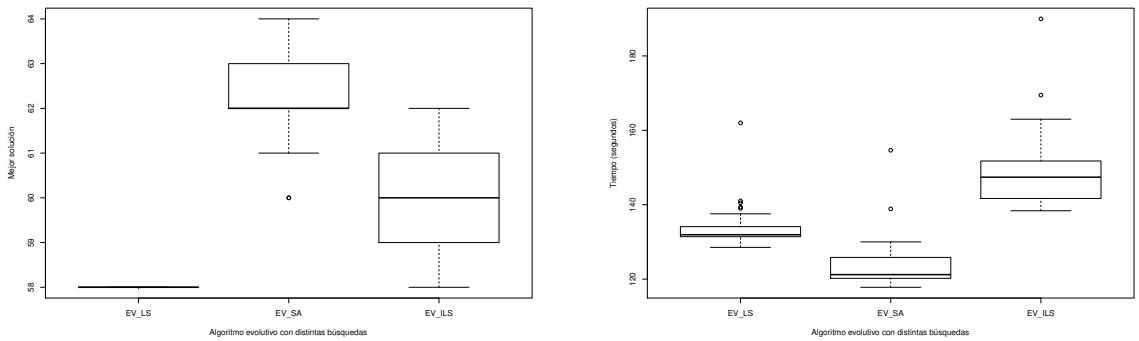
Instancia: els19[3]





Instancia: esc64a[4]

Como puede verse en el último caso, la diferencia en tiempo resulta ser bastante considerable para ILS, llegando a tardar 40 minutos, por lo que se planteó una prueba en donde los algoritmos tuvieran un tiempo de ejecución similar y ver como se desempeñan en cuanto a la mejor solución encontrada. Se redujo la población y los parámetros para cada algoritmo se ajustaron. Los resultados se pueden ver a continuación.



Instancia: esc64a[4] con parámetros ajustados

En general, las ejecuciones tenían una duración de poco más de dos minutos aproximadamente y como se puede notar, la primera búsqueda local, en conjunto con un algoritmo evolutivo, fue la que tuvo el mejor comportamiento. Por su cuenta se estancaba en mínimos locales pero el algoritmo evolutivo logró resolver este problema. La búsqueda local iterativa tiene mayor costo de ejecución y combinarla con un algoritmo evolutivo resulta en un incremento abrupto del tiempo.

Por si solo, ILS, brinda diversificación y una búsqueda ambiciosa. Usarla en conjunto con un evolutivo parece ser redundante y en cierta forma termina en perdida. Por el caso contrario, la simplicidad de LS resultó tener un desempeño perfecto, retornando siempre la solución óptima. En el caso de SA, resulta difícil de ajustar, su implementación debe hacerse con cuidado y tal vez sea conveniente optar por la simplicidad. De igual manera brinda buenas soluciones.

CONCLUSIONES

Para finalizar este trabajo se desea concluir que sin duda el problema de asignación cuadrática es desafiante, la implementación de distintas heurísticas, combinarlas y probar exhaustivamente los mejores parámetros para que el desempeño sea el más óptimo es complicado. Por si sola, una búsqueda local resulta un buen algoritmo que ofrece soluciones aceptables, y sin duda se pueden ajustar para que su exploración se amplíe. Esa diversificación o visión extendida del espacio de soluciones puede conseguirse de distintas maneras, como se ha podido ver, ya sea por un refinamiento en la búsqueda local o introduciendo un algoritmo evolutivo.

Al final, la implementación de los algoritmos resulta ser lo menos difícil. Por otro lado, ajustar y conseguir las funciones o estrategias ideales es lo más interesante. Lo más probable es que cada heurística se adapte mejor a cierto problema. En general, estos problemas *NP* son controversiales y un acercamiento a la solución usando algoritmos de optimización es una buena respuesta a estos problemas tan complejos. Requiere tiempo y conocimiento que puede existir en un lugares inesperados. Logrando adaptar conceptos que no están directamente relacionados con la ciencia de la computación o matemáticas es sin lugar a dudas artístico y requiere una inmensa creatividad; una visión extendida del mundo.

Bibliografía

[1] E.H.L. Aarts y J. Korst. *Simulated annealing and Boltzmann machines: a stochastic approach to combinatorial optimization and neural computing*. Wiley-Interscience series in discrete mathematics and optimization. Wiley, 1989. ISBN: 9780471921462. URL: https://books.google.es/books?id=K%5C_pQAAAAMAAJ.

[2] R.E. Burkard y col. *QAPLIB - A Quadratic Assignment Problem Library*. Jun. de 2018. URL: http://anjos.mgi.polymtl.ca/qaplib/.

[3] Alwalid N. Elshafei. «Hospital Layout as a Quadratic Assignment Problem». En: *Journal of the Operational Research Society* 28.1 (1977), págs. 167-179. DOI: 10.1057/jors.1977.29. eprint: https://doi.org/10.1057/jors.1977.2.

[4] Bernhard Eschermann y Hans-Joachim Wunderlich. «Optimized synthesis of self-testable finite state machines». En: *FTCS*. 1990.

[5] T.C. Koopmans y M.J. Beckmann. *Assignment Problems and the Location of Economic Activities*. Bobbs-Merrill reprint series in economics. Cowles Foundation for Research in Economics at Yale University, 1957. URL: https://books.google.es/books?id=Kjs24p57FxgC.

[6] Christopher E. Nugent, Thomas E. Vollmann y John Ruml. «An Experimental Comparison of Techniques for the Assignment of Facilities to Locations». En: *Operations Research* 16 (1968), págs. 150-173.

[7] A.S. Ramkumar, S.G. Ponnambalam y N. Jawahar. «An evolutionary search heuristic for solving QAP formulation in facility layout design». En: *IEEE Congress on Evolutionary Computation*. IEEE, 2007, págs. 4005-4011. ISBN: 978-1-4244-1339-3. DOI: 10.1109/CEC.2007.4424993.

[8] Kambiz Shojaee y col. «New Simulated Annealing Algorithm for Quadratic Assignment Problem». En: *The Fourth International Conference on Advanced Engineering Computing and Applications in Sciences, ADVCOMP*. IARIA, 2010, págs. 87-92. ISBN: 978-1-61208-101-4.