Comparando búsqueda A* y algoritmos genéticos para el problema del vendedor viajero

[1]Eduardo Blanco Bielsa, Juan Gómez Tejeda, Fernando Sáenz de Santa María Modroño. Sistemas Inteligentes. Grado en Ingeniería Informática del Software. EII. Universidad de Oviedo. Campus de los Catalanes. Oviedo.

Abstract. En este artículo se presentan dos enfoques para resolver el clásico Problema del Viajante de Comercio (TSP): uno basado en metaheurística y otro en algoritmia genética. El TSP implica encontrar la ruta más corta que visite un conjunto de ciudades y regrese a la ciudad de partida sin repetir ninguna ciudad. Evaluamos el rendimiento de estos enfoques utilizando diversos conjuntos de datos y comparamos los resultados obtenidos entre ellos, además de compararlos con los resultados teóricos conocidos. También analizamos el rendimiento de los enfoques en varias instancias del problema, lo que proporciona una comprensión más profunda de su comportamiento. Nuestros resultados revelan detalles significativos sobre la eficacia de estos métodos para abordar el TSP, contribuyendo así al campo de la optimización combinatoria y ofreciendo perspectivas valiosas para futuras investigaciones en este dominio.

Palabras clave: A*, algoritmos genéticos, TSP, búsqueda heurística, operador de cruce, heurístico, convergencia, estado, coste.

1 Introducción

En el ámbito de la optimización combinatoria, el clásico problema del viajante de comercio o Travelling Salesman Problem (**TSP**) supone un desafío fundamental: encontrar la ruta que visite un conjunto de ciudades y regresar a la ciudad de partida.

Ante la complejidad de este problema, el uso de múltiples técnicas heurísticas ha surgido como herramientas cruciales. En este contexto, exploramos la convergencia desde dos enfoques ampliamente conocidos: la **búsqueda heurística** y los **algoritmos genéticos**.

Este estudio investiga sobre la eficacia de estas dos técnicas en la resolución del **TSP**, analizando su rendimiento, coste y calidad en diversas configuraciones. Nuestro objetivo no es únicamente profundizar en la comprensión de estos métodos, sino también proponer nuevas perspectivas para la resolución eficiente de problemas complejos de optimización combinatoria, con aplicaciones significativas en el mundo real y la ciencia de la computación.

[1] El nombre de los autores ha sido ordenado alfabéticamente por apellido

Por ende, a lo largo de los apartados de este estudio se exponen definiciones clave, algoritmos empleados en la búsqueda de soluciones, diferentes aplicaciones del **TSP**, comparaciones entre las técnicas usadas, resultados obtenidos y conclusiones del estudio, todo ello para una correcta comprensión de este.

2 El problema del viajante de comercio

[1] El problema del viajante de comercio o *Travelling Salesman Problem* (**TSP**) responde a la siguiente pregunta: dado un conjunto de ciudades y las distancias entre cada par de ellas, ¿cuál es la ruta más corta posible que permite visitar cada una de ellas exactamente una vez y volver a la ciudad de partida? Este es un problema NP-Complejo dentro de la optimización combinatoria realmente importante en investigación operativa y ciencias de la computación.

[2] El **TSP** tiene diversas aplicaciones aún en su formulación más simple, tales como: cristalografía de rayos X, circuitos electrónicos, rutas de vehículos y trazado de máscaras en producciones de placas base.

[3] Por ello, debido al interés generado por el problema, se han descubierto varios métodos para intentar resolverlo. Algunos de los métodos de resolución clásicos son: la *aproximación por fuerza bruta*, la *programación dinámica* y el *algoritmo de Christofides'*. Sin embargo, en los últimos años también se ha logrado descubrir nuevos métodos como la *Geometría de polinomios*, *Soluciones fraccionarias* y *técnicas de redondeo*.

3 Los algoritmos de búsqueda

Para este experimento se ha optado por usar los dos algoritmos de búsqueda siguientes:

El **algoritmo *A**** es un algoritmo de búsqueda en grafos en el que se utiliza una función, denominada como *f(n)* siendo *n* el estado concreto, para seleccionar qué estados expandir y cortar, de forma que se prioricen los valores mínimos de esa función. En este caso, dicha función *f(n)* consiste en la suma del coste hacia ese estado *n* (denominado *g(n)*), y una función heurística *h(n)*. En este algoritmo es importante que el heurístico *h(n)*[8] esté bien informado, ya que daría mejores soluciones y evitaría la expansión de nodos innecesarios, reduciendo así el tiempo de resolución del problema.

El **algoritmo de Búsqueda Genética** [9] se basa en la generación, selección y cruce de poblaciones de individuos por un número de generaciones determinado. Dicha selección se lleva a cabo mediante la función *fitness(n)* (función que evalúa la cercanía de una posible solución a la óptima o deseada) para posteriormente emplear un operador de cruce que genere un nuevo individuo hijo con genes de ambos padres.

4 Aplicación de los algoritmos al problema del viajante de comercio

4.1 Búsqueda heurística con A*

Para aplicar el algoritmo A* al problema, es importante hablar del espacio de búsqueda usado y de los heurísticos que se evaluarán más adelante.

4.1.1 El espacio de búsqueda

El espacio de búsqueda del TSP consiste en todas las combinaciones de caminos válidos, un total de N! estados posibles para un número N de ciudades. Fijando una ciudad como inicio siempre, esto se reduce a (N-1)! estados. En todo caso, la complejidad del problema no se reduce significativamente.

Para describir un estado del espacio, se utiliza una tupla que contiene una lista de las ciudades ya visitadas, y la ciudad actual. Estos estados no representan soluciones al problema, únicamente subproblemas a resolver. Para recuperar la solución, o sea el orden en que se deben visitar las ciudades, el algoritmo guarda en cada nodo, a parte de su estado, el nodo desde el que se expandió.

La tupla descrita no contiene la ciudad inicial, ya que se hace fija en el código y no es necesario para la descripción completa de un estado.

4.1.2 Heurísticos utilizados

El diseño de heurísticos mediante relajación del problema consiste en simplificar el problema de forma metódica para convertirlo en un problema soluble en tiempo polinomial. Para ello, se establecen una serie de restricciones del problema original, y se eliminan unas u otras. Crear heurísticos mediante este método asegura la admisibilidad y la monotonía de estos [4]. Además, permite comparar la dominancia de los heurísticos generados.

Con respecto al TSP, dado un estado en el que se han visitado k >= 1 ciudades y estamos en la ciudad X, se busca un subconjunto de arcos del grafo de coste mínimo, con las siguientes restricciones:

R1. Que tenga N-k+1 arcos.
R2. Que los arcos toquen a la ciudad inicial, a las no visitadas, y a X.
R3. Que el conjunto de arcos sea de grado 1 para la inicial y X, y de grado 2 para las no visitadas.
R4. Que los arcos conecten a la ciudad inicial, a las no visitadas, y a X entre sí.

h1 es el único heurístico estudiado que no corresponde a una relajación del problema, y establece una cota mínima para la solución. Utiliza un grafo residual, compuesto por los arcos que unen a la ciudad inicial, la actual, y las no visitadas entre ellas. De éste,

selecciona el arco mínimo que sale de cada ciudad, y suma sus costes. Al ceñirse a una cota baja del coste del problema, es un heurístico admisible.

h2 es el heurístico más sencillo de los aquí estudiados. Es un heurístico derivado de relajar R2, R3 y R4. Por tanto, lo que busca es el conjunto de N-k+1 arcos de coste mínimo del subgrafo residual. Dado que h2 relaja mucho el problema del TSP, se podría decir que está bastante poco informado.

El heurístico h3 proviene de la relajación de las restricciones R3 y R4. Así, trata de obtener el subconjunto de N-k+1 arcos que toquen a todas las ciudades del grafo residual con menor coste. Esto podría dejar ciudades desconectadas, pero es una aproximación mejor que h2, que ni siquiera tiene en cuenta visitar todas las ciudades.

Obtener el valor de h3 para un estado es un proceso iterativo sobre una matriz. Esta matriz representa todos los arcos del grafo residual. En cada iteración, se obtiene el arco de menor coste, y su valor se añade al total. Además, la fila y columna donde se encontraba este arco se eliminan de la matriz, para indicar que ya se ha tomado una ruta saliendo de una ciudad y llegando a la otra. El proceso continúa hasta que se alcanza el número esperado de arcos, y se devuelve el total acumulado al terminar.

h_MST proviene, al igual que los dos últimos, de relajación del problema. En concreto, este heurístico únicamente relaja R3. Así, lo que calcula es el coste del árbol de expansión mínimo del grafo residual.

Existen algoritmos que resuelven este problema en tiempo polinomial. En nuestro caso, hemos optado por el algoritmo de Kruskal [5]. Para ello, se obtuvo una clase que implementa el algoritmo de [6]. Este heurístico está muy bien informado, siendo mejor tanto que h2 como que h3. Sin embargo, el algoritmo de Kruskal no es simple, con una complejidad de O(nlog(n)), lo que podría hacerlo impráctico en el mundo real para grafos demasiado grandes.

4.2 Búsqueda mediante algoritmos genéticos

El espacio de búsqueda para algoritmos genéticos se representa distinto que para búsqueda A*. Podemos hacer la misma simplificación de fijar una ciudad como la inicial, y así el gen se compone de una lista del resto de las ciudades en cualquier orden, que describe el camino a tomar.

El cruce de dos individuos debe mantener que el gen sea una permutación del orden de las ciudades. Por ello, hay dos opciones principales: cruce uniforme y cruce en dos puntos. El primero consiste en copiar de forma aleatoria algunos genes de un padre, y "rellenar" con los genes del otro en orden. El cruce en dos puntos selecciona una sección de tamaño aleatorio de uno de los padres para copiar al hijo, y "rellena" con los genes del otro padre en orden, saltando las ciudades ya añadidas por el primer padre. Este segundo resulta intuitivamente mejor, ya que es capaz de conservar secciones de camino buenas de los padres, si es que las hay. La mutación de un individuo se realiza intercambiando dos ciudades adyacentes.

La función de fitness debe premiar los caminos menos costosos. Para ello, se calcula el coste del camino descrito por el gen y se introduce en una función positiva constantemente decreciente. Se podría usar simplemente la función inversa (1/x), pero el problema de esta es que premia demasiado poco a los caminos buenos con respecto a los malos, en el rango del coste que estamos hablando. Hemos encontrado que lo mejor es una función con la forma $a^{b/x}$, siendo x el coste del camino, a un parámetro mayor que 1 para indicar cuánto más se quiere premiar a los caminos poco costosos, y b un parámetro para indicar un coste aproximado esperado de la solución. Esto premia adecuadamente a los individuos con menor coste y mejora las soluciones encontradas significativamente.

5 Estudio experimental

5.1 Diseño del sistema experimental

Para hacer el estudio experimental, lo descrito anteriormente debe ser ejecutado. En concreto, se usará un ordenador con un procesador Intel i7-12800 HX 2 GHz. Para el estudio del A*, se ejecutará cada heurístico un total de 20 veces para poder medir adecuadamente su rendimiento. Además, se ejecutará sobre 4 instancias del problema distintas, de distintos tamaños, dos de ellas obtenidas del TSPLIB [7] (gr17 y gr21) y dos de ellas creadas por nosotros de tamaño N=12 y N=19.

Para el estudio de los algoritmos genéticos, debido a su naturaleza aleatoria, se ejecutarán 30 veces cada combinación. A parte de estas cuatro instancias mencionadas, se probará con con los dos operadores de cruce descritos anteriormente, cruce uniforme y cruce en dos puntos.

La mutación de un individuo se realiza un 10% de las veces. Los parámetros a y b de la función de fitness se han inicializado como a=3 y b=10000.

5.2 Resultados experimentales

5.2.1 Búsqueda A*

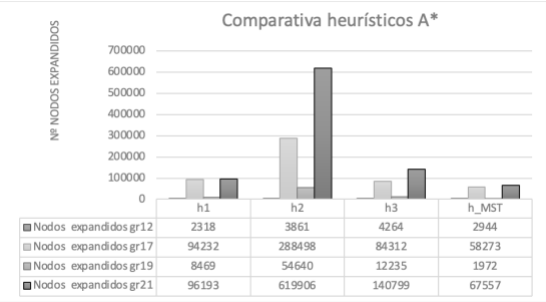


Fig 1: Gráfica de comparativa de nodos expandidos para distintos heurísticos.

En primer lugar, mostramos la comparativa en términos de nodos expandidos del A* con los heurísticos h1, h2, h3 y h_MST:

Creemos firmemente que los heurísticos *h1* y *h2* cumplen la propiedad de **admisibilidad**, ya que soportan los resultados teóricos al encontrar en todas nuestras pruebas la solución óptima para cada instancia. Sin embargo, los resultados obtenidos con los heurísticos *h3* y *h_MST* son suficientes para determinar que su implementación genera un heurístico no admisible.

Tabla 1: Estados re-expandidos para cada instancia y heurístico.

Instancia	gr12	gr17	gr19	gr21
h1	1	2	0	0
h2	0	0	0	0
h3	357	13467	2081	19309
h_MST	0	0	0	0

A su vez, la implementación de h3 es no monótona. Según se ve en la tabla 1, el algoritmo A* re-expande estados al usar este heurístico, lo que es una muestra de no monotonía. La no monotonía de h1 se demuestra de forma similar. Sobre la monotonía de h2 y hMST, ésta no puede confirmarse dado su comportamiento en las instancias observadas, más allá de decir que, por el estudio teórico, sí deberían ser monótonos, y que nada observado lo desmiente.

5.2.2 Algoritmos genéticos

Como se puede apreciar en la figura 2, el operador de cruce order crossover (ox) encuentra los menores costes mínimos de ambos operadores (línea amarilla), mientras que el uniform crossover (ux) encuentra los menores costes promedios de los dos operadores (línea azul).

Con respecto a la convergencia, se puede apreciar que para tamaños más pequeños del problema se produce una convergencia mucho más rápida, como en el caso del problema de tamaño 12, que converge casi al principio de las generaciones, el de tamaño 17 a las 50 generaciones, el de tamaño 19 a las 125 y el de tamaño 21 justo converge a las 200 generaciones, al final de la ejecución. Sin embargo, el operador de cruce order crossover (ox) converge más rápidamente que el uniform crossover (ux) en todos los casos, como se puede apreciar en las gráficas anteriores.

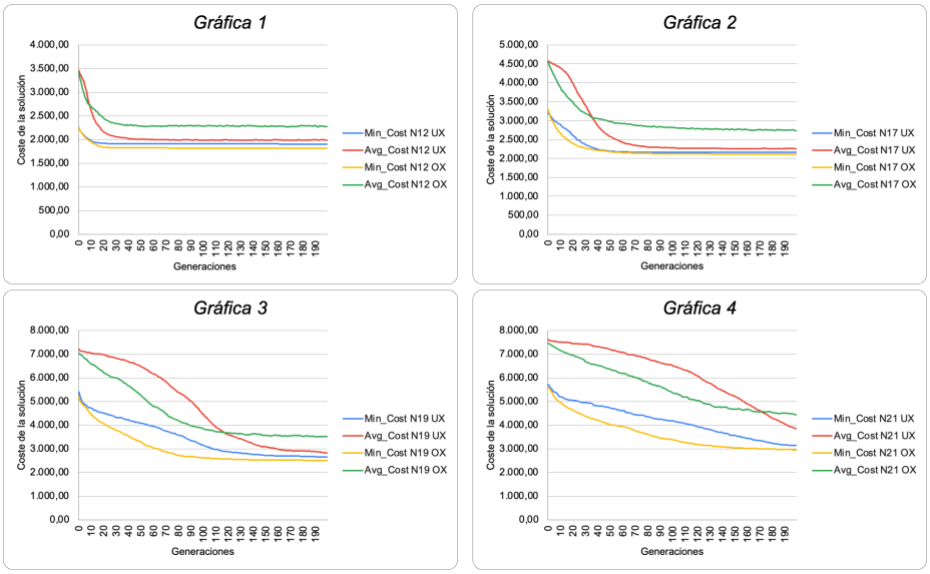
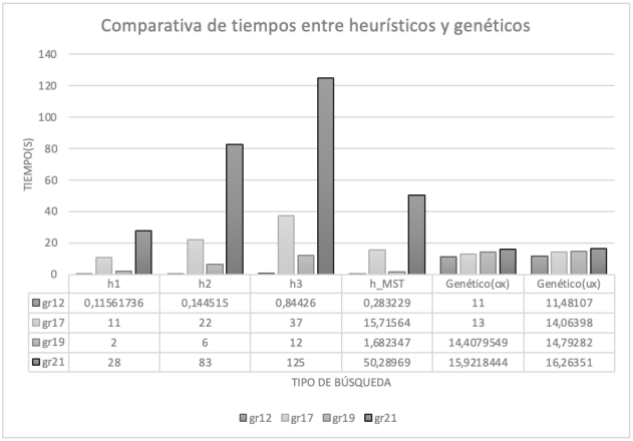


Fig 2: Gráficas de comparativas de operadores de cruce para diferentes tamaños del problema.

5.2.3 Tiempos

Como se puede apreciar en la figura 3, en todos los heurísticos existe una tendencia por la cual el tiempo de ejecución aumenta con el tamaño. También se puede apreciar que el heurístico h1 es el más rápido, pese a que el heurístico h_MST expanda menos estados, posiblemente debido a que el cálculo del heurístico es más costoso computacionalmente. Además, también se deduce que no existe una diferencia realmente significativa entre ambos operadores de cruce.



	h1	h2	h3	h_MST	Genético(ox)	Genético(ux)
gr12	0,11561736	0,144515	0,84426	0,283229	11	11,48107
gr17	11	22	37	15,71564	13	14,06398
gr19	2	6	12	1,682347	14,4079549	14,79282
gr21	28	83	125	50,28969	15,9218444	16,26351

Fig 3: Gráfica con datos de comparación de tiempos entre búsqueda heurística y algoritmos genéticos.

Por último, los algoritmos genéticos son mucho más consistentes respecto al tiempo que el algoritmo A*, ya que ambos operadores de cruces tardan prácticamente lo mismo para todos los tamaños del problema, tardando más en el caso de N=12 y N=17, pero menos en el caso de N=19 y mucho menos en el caso de N=21, por ende, se puede deducir que los algoritmos genéticos son más adecuados para tamaños grandes del problema que el A*, ya que encuentran soluciones válidas en un tiempo mucho mejor, aunque son peores para tamaños pequeños del problema, ya que tardan más sin encontrar siempre las soluciones óptimas.

6 Conclusión

Los algoritmos genéticos son significativamente más estables en cuanto a tiempo de ejecución, y escalan mucho más predeciblemente con el tamaño del problema. Eso sí, no aseguran alcanzar una solución óptima. De hecho, como se ha observado, la convergencia en una buena solución tarda más para tamaños más grandes del problema, por lo que se podrían necesitar más y más generaciones al aumentar N.

La búsqueda de A* es determinista, y con un buen heurístico que sea tanto rápido como admisible, como h1, se asegura encontrar la solución óptima; aunque no se tenga seguridad del tiempo de ejecución ya que no es consistente con el tamaño del problema.

A la hora de elegir uno u otro, dependerá de las necesidades de uno: si la solución óptima es necesaria, y no se va a ejecutar a menudo, lo mejor sería utilizar A*. Sin embargo, si basta con una solución buena, y no óptima, y el poder limitar el tiempo de ejecución a un máximo, la búsqueda por algoritmos genéticos es claramente superior.

7 Bibliografía

[1] Wikipedia. Problema del viajante.
[2] InTech, Donald Davendra, Teodora Smiljanic. Travelling Salesman Problem, Theory and Applications. (2010).
[3] Leanne White, RoadWarrior. Solving the Travelling Salesman Problem (2023).
[4] Pearl, J.: Heuristics: Intelligent Search Strategies for Computer Problem Solving, pp. 115-116. Addison-Wesley Publishing Company, Reading, Massachusetts (1984).
[5] Kruskal, J. B.: On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical Society. **7**, pp. 48–50 (1956).
[6] Javatpoint. Implementation of Kruskal's Algorithm in Python.
[7] Reinelt, G.: TSPLIB–A Traveling Salesman Problem Library (1991).
[8]Investigation of the * (Star) Search Algorithms: Characteristics, Methods and Approaches https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=831ff239ba77b2a8eaed473ffbfa22d61b7f5d19
[9] STEPHANIE FORREST :Genetic Algorithms https://dl.acm.org/doi/pdf/10.1145/234313.234350