VNS APLICADO A TSP

Héctor E. Núñez*, Ricardo D. Quiroga† y Nelson Efrain A. Cruz††

*Universidad Nacional de Salta
Salta, Argentina
e-mail: gimlihen@hotmail.com

†Universidad Nacional de Salta
Salta, Argentina
e-mail: l2radamanthys@gmail.com

††Universidad Nacional de Salta
Salta, Argentina
e-mail: neac03@gmail.com

1. ABSTRACT

El problema del viajante es uno de los más complejos, más utilizados y más estudiados de los problemas de optimización combinatorial. Es una pequeña e irónica paradoja en la que existe un algoritmo sencillo para hallar la solución óptima pero que es, en la practica imposible de aplicar por su altísimo costo de tiempo computacional. En este trabajo atacamos al problema de TSP aplicando la metaheuristica de VNS básico con unas pequeñas modificaciónes para mejorar los resultados obtenidos mediante una mejor elección de soluciónes iniciales. Es importante buscar alternativas de resolución, pues al ser un problema NP completo, el tiempo y esfuerzo computacional requerido para resolver este tipo de problemas aumenta exponencialmente con respecto al tamaño del problema, por lo tanto es importante idear algoritmos que puedan encontrar soluciónes lo suficientemente rápido y que sean tan buenas como para la toma de decisiones.

2. INTRODUCCION

El problema del viajante (TSP por sus siglas en ingles) es de un planteamiento bastante sencillo: "Dado un conjunto de n ciudades que poseen caminos que permiten ir de una ciudad a otra sin tener que pasar por ninguna intermedia, se desea conocer cual es la ruta mas corta para recorrer todas las ciudades y volver al inicio si es que se conocen las distancias de todos los caminos y además cada ciudad se debe visitar solo una vez".
El interés en el estudio de técnicas para su solución se debe a la enorme cantidad de aplicaciones prácticas en problemas de toma de decisiones donde este aparece como subestructura: por ejemplo desde implementaciones en logística de transporte hasta encontrar la forma de realizar la menor cantidad de desplazamientos para realizar perforaciónes en una plancha. TSP es un tipo de problema que no se puede resolver en tiempo polinomial en función del tamaño del problema, es un problema en el que el tiempo y esfuerzo computacional para resolver este problema aumenta exponencialmente respecto a su tamaño. La mejor opción para intentar resolver este problema es obtener soluciónes aproximadas que puedan encontrarse rápido y lo suficientemente buenas como para ser útiles a la hora de tomar decisiones. Para ello se han planteado distintos algoritmos que no garantizan optimalidad pero si buenas soluciónes, estos son los algoritmos heuristicos y metaheuristicos los que en realidad se puede ver como una heurística que maneja heurísticas.
En este informe aplicamos la metaheuristica Bsqueda por Entornos Variables (Variable Neighbourhood Search, VNS) como enfoque básico para intentar hallar soluciónes, aunque después de implementar VNS básico, nos dimos cuenta que no era suficiente, pues si bien se hallaban soluciónes relativamente buenas, llegar a ellas requeria de bastantes iteraciónes y por lo tanto de

bastante tiempo, fue así que introdujimos una memoria de largo plazo, con el fin de lograr una mejor forma de explorar el espacio de las soluciónes, con esto se logró tenér una mayor velocidad de aproximació al óptimo local pero también nos atrapaba en el mismo, así que tubimos que se utilizo un criterio para flanquear y escapar de este estancamiento.
La idea de una memoria a largo plazo ya se a útilizado en otros métodos como el Tabu Search, así como la idea asignarle una probabilidad (sección 4.2) a las aristas que aparentan ser las mejores candidatas para encontrarse en el óptimo global, aún que no se ha encontrado en ningún trabajo anterior de los obtenidos que se utilice esta última, al menos no de forma explícita.
En la sección siguiente proveemos información un poco más detallada sobre el problema en cuestión y algunos de los enfoques ya utilizados para atacar este problema, luego en la sección 3 pasamos a describir como encaramos finalemente el problema y algunos comentarios sobre la implementación del mismo, a continuación de ello brindamos detalles de las pruebas realizadas con el algoritmo sobre las instancias de problemas publicadas en la conocida libreria de TSPLIB, finalmente se podrań hallar las conclusiones que obtuvimos luego de realizada la experiencia.

3. EL PROBLEMA

TSP puede ser representado gráficamente mediante un grafo completo, en el que los nodo representan los puntos a ser visitados y las aristas (al asignarles pesos) representan el coste de ir de un punto a otro, entonces el camino elegido para realizar el recorrido no seria mas que un ciclo hamiltoniano (en la teoría a el camino se lo suele llamar tour). Visto de este modo se puede formular un modelo matematico de programación lineal entera tal como lo hicieron George Dantzig, Ray Fulkerson y SelmerJohnson [1]:

$$min\, z = \sum_{i=1}^{n}\sum_{j=1}^{n} c_{(i,j)}x_{(i,j)} \tag{1}$$

s.a:

$$\sum_{j=1}^{n} x_{(i,j)} = 1\,, \forall i \in n \tag{2}$$

$$\sum_{i=1}^{n} x_{(i,j)} = 1\,, \forall j \in n \tag{3}$$

$$\sum_{i\in S}\sum_{j\in S} x_{(i,j)} \leq |S| - 1\,, \forall\, S \subset N, S \neq \emptyset \tag{4}$$

$$x_{(i,j)} \in \{0,1\}\,, i,j \in N \tag{5}$$

Donde $x_{(i,j)} = 1$ si se usa la arista (i,j) como parte del camino elegido y $x_{(i,j)} = 0$ en caso contrario. La función a minimizar (1) representa el costo total de el camino elegido y los $c_{(i,j)}$ representan el costo de ir desde el nodo i al nodo j, las funciones (2) y (3) son para que en el camino elegido cada arista se use una sola vez y finalmente la ecuación (4) esta para evitar que se formen sub-caminos, donde $|S|$ es la cantidad de nodos recorridos en el sub-camino S.

3.1. Encontrando soluciones a TSP

Como ya dijimos, muchos enfoques distintos se han realizado para resolver TSP entre ellas podemos encontar los siguientes :

1. Métodos de construcción de caminos [2][3][4] : Como por ejemplo la heuristica del vecino mas cercano o la heuristica voraz (Greedy) .

2. Métodos de mejoramiento : se trata de métodos que partiendo de una solución inicial realizan cambios en ella hasta obtener una mejor solución, podemos nombrar a métodos de búsqueda local descendente como 2-opt [5], 3-opt [6], una variacion de la anterior K-opt (heurísticas basada en Lin-Kernighan) [7][8], simulated annealing [9][10], algoritmos genéticos [13], Tabu search [11][12].

3. algoritmos Branch & Bound sobre el modelo de programación lineal

4. Optimización mediante Colonia de Hormigas, que e una metaheuristica que simula un aspecto de la naturaleza.

Una forma común de medir la calidad (cuanto se acerca a el optimo) de una heurística, es compararla con la cota inferior de Held-Karp (HK) [14], esta cota se obtiene como la solución relajada de la formulación de programación lineal de TSP. Pero obtener la cota de ese modo puede resultar imposible para instancias de TSP muy grandes siendo solo valido para instancias chicas.

4. SOLUCION PROPUESTA

La metaheuristica de Variable neighborhood search (VNS) que traducido seria algo así como búsqueda de entorno variable, dice en su versión básica así[15] :

Pseudocodigo de VNS basico

```
Inicializar
        seleccionar estructura de entornos N_k con k = 1...k_max
        generar una solucion inicial x
        elegir un criterio de parada
Repetir hasta que se alcance criterio de parada
        k = 1
        repetir hasta que k = k_max
                generar una solucion inicial x' tal que x' ∈ N_k(x)
                aplicar busqueda local sobre x' llamar a esta nueva solucion x''
                si x'' < x entonces x = x'' y k = 1
                sino k = k + 1
```

VNS es una metaheurística que está basada en un principio simple: cambiar de estructura de entornos cuando la búsqueda local se estanca en un óptimo local, buscando así poder escapar de este, pues una de los tres echos en los que se basa VNS es que el óptimo local en una estructura de entornos no lo es necesariamente en otra [15]. Pero para obtener buenos resultados con VNS es también muy importante como se definen las estructuras de vecindarios. Para poder definir los vecindarios usualmente se definen medidas de distancia para poder establecer una "lejanía"(o cercanía si se quiere) de las soluciones.

4.1. Distancias, vecindarios y búsqueda local

Teniendo en cuenta que en TSP todas las soluciones posibles no son mas que permutaciones de una cantidad n de nodos. Definimos distancia de las soluciones x_1 a x_2 como:

$$\rho(x_1, x_2) = \textit{número de nodos en diferentes posiciones}$$

Por ejemplo dadas dos soluciones factibles distintas x_1 y x_2 que están descriptas por la siguiente sucesión de $n = 6$ nodos:

$$x_1 = [a, b, c, d, e, f]$$
$$x_2 = [c, a, b, d, e, f]$$
$$\text{es } \rho(x_1, x_2) = 3$$

Ya que tienen 3 elementos en posiciones distintas, es de notar que la distancia mínima que puede haber entre dos soluciones es 2, ya que si un elemento de la solución 1 esta en una posición distinta que en la solución 2 hay forzosamente otro elemento que esta en una posición distinta en solución 1. La distancia máxima es igual a n, que seria el caso en que todos los nodos se hallen en posiciones distintas en ambas soluciones, una definición de distancia similar puede ser encontrada en [16] y aunque en ese caso fue aplicado a el problema de la p-mediana nos sirvió de modelo.
Una ves definida la distancia entre soluciones podemos definir nuestra estructura de vecindarios como sigue:

$$x' \in N_k(x) \Leftrightarrow \rho(x', x) = k$$

Aun falta por esclarecer el modo en que vamos a recorrer los vecindarios y cuantos vecindarios vamos a usar, ya que podríamos usar los $n-1$ vecindarios posibles pero esta acarrearía una mayor cantidad de iteraciones. Nosotros optamos por usar una cantidad fija de entornos, que se puede pasar como parámetro, este parámetro también define de forma implicíta que vecindarios se visitan pues en realidad para k generamos un elemento del vecindario $N_{k'}(x)$ donde $k' = \frac{n}{k}$ o sea que para $k = 1$ generamos un elemento del vecindario $N_n(x)$ siendo n el tamaño del problema. Esto se podría describir como empezar a buscar en el vecindario mas lejano de x para luego empezar a buscar en las cercanías de x ya que a medida que crece k, k' decrece mucho mas rápido, llegando al final por problemas de redondeo a buscar en los mismos vecindarios (en las cercanías de x).
Para la heurística de busqueda local utilizamos movimientos 2-opt hasta encontrar el 2-opt-optimo [14] que si bien no es muy bueno, pues en media llega a valores encima del 5 % de la cota de HK, precisa de poco tiempo para llegar a este óptimo local y es algo que buscábamos pues aplicar un 3-opt, por ejemplo, precisaba de mucho mas tiempo y esto ralentizaba demasiado el algoritmo puesto que se realizan bastantes búsquedas locales.

4.2. Elección estadística del vecino

Una vez probado VNS básico notamos que si bien el algoritmo producía, relativamente, buenas soluciones este era un poco "tonto", pues en pos de la diversificación se generaban de forma aleatoria los elementos de las vecindades correspondientes y dado ello se conseguían demasiadas comparaciones, un poco acercándose a la fuerza bruta, que eran muy probablemente desechadas pues los óptimos locales conseguidos no eran mejores que el óptimo global conocido (no el óptimo global real). Es decir precisábamos una mejor manera de elegir los vecinos, tratando así de reducir las comparaciones realizadas o desde otro punto de vista que los vecinos elegidos sean "buenos vecinos" a los que al aplicarse una búsqueda local generasen buenas soluciones.

Una aproximación a ello fue que a medida que se obtenían nuevos óptimos locales se iba tomando cuenta de las aristas usadas en la solución inicial que origino el óptimo local en una memoria de largo plazo. Entonces cada vez que se usa una arista esto es registrado en la memoria aumentando en uno el contador correspondiente a esa arista. La idea tras esto es que aquellas aristas que aparezcan de forma seguida en las "buenas"soluciones iniciales muy probablemente formen parte de una "buena"solución inicial que generara una buena solución (óptimo local). Veamos que dada una arista podemos referirnos a ella como un par de nodos, que serian los nodos que conecta esa arista, así la arista (a, b) conecta los nodos a con b. Entonces si la arista (a, b) tiene un contador con un valor alto quiere decir que en las soluciones que fuimos encontrado la sub-sucesíon $[a, b]$ o $[b, a]$ apareció muy frecuentemente.
Definimos que si dado que una arista (y, x) tiene el valor de contador mas alto, este tendrá la probabilidad mas alta de **seguir** siendo usada en el vecino que se esta generando y las demás aristas tendran una probabilidad proporcional dependiendo del valor de su contador.
Como armar un vecino de x correspondiente al vecindario k se logra cambiando de posición k nodos en el vector solución x, para poder aplicar la memoria de largo plazo al algoritmo que crea los vecinos de x basta con que cada vez que estemos por elegir un nodo a ser cambiado de posición revisemos cual es la probabilidad de que este se mantenga en su posición (recordando que una arista se puede ver como un par de nodos) y en base a ello decidamos si mover o no este nodo o viéndolo de otro modo si seguir usando o no la correspondiente arista según su probabilidad. Hay que tener en cuenta que mover un nodo de posición implica dejar de usar dos aristas por lo que han que tenerse en cuenta dos probabilidades.
Algo importante que nos dimos cuenta después de probar el algoritmo es que usar de este modo la memoria muy probablemente nos atrapaba en un óptimo local, pero a cambio se aceleraba la llegada a este óptimo y era en general una solución bastante buena, se opto entonces por que inicialmente la memoria funcione de la manera descripta y si después de una cierta cantidad de iteraciones no se hallaba una mejora, se invertiría el uso de la memoria, o sea que la arista con el valor de contador mas alto tendría la mayor probabilidad de **dejar** de usarse en el vecino que se esta generando. En el modo invertido de la memoria lo que se logra es una mayor diversidad pues se tienden a usar las aristas menos usadas, mientras que en el modo normal de la memoria se logra una menor diversificación, de allí que podamos quedar estancados en un óptimo local en el modo normal.

5. PRUEBAS

Las instancias de problemas y las mejores soluciones conocidas que se utilizaron para probar el algoritmo fueron tomadas de la librería TSPLIB. Como mas arriba se menciona al algoritmo solo se le definen 4 parámetros:

1. El problema en si (nombre y ruta del archivo que contiene el problema)
2. El numero de vecindades que se usaran K_{max}.
3. La cantidad de iteraciones globales *Its*.
4. La probabilidad asociada a el contador con valor mas alto en la matriz de memoria P_{max}

Durante las pruebas nos dimos cuenta que era recomendable que el valor de P_{max} fuera bastante elevado (mayor a 0.9) para que la memoria tuviera un real impacto, el valor que se uso en las

pruebas se mantuvo constante en 0.98.

En total son 12 los problemas abordados, el mas pequeño que se muestra es de 130 vértices y el mayor de 666 vértices, elegimos estos problemas por que los demás o son muy pequeos y se llega a el optimo bastante rápido, o bien son demasiado grandes y para lograr una solución con menos de 3 % de error se tendría que usar una cantidad adecuada de vecindades (un numero grande) por lo que el tiempo que tardaría en resolverse serian considerablemente alto.
La siguiente tabla muestra los resultados promedios obtenidos durante 10 corridas de cada uno de los problemas que evaluamos, aparte se muestra también como influyen en el resultado el aumento de las vecindades usadas y la cantidad de iteraciones del algoritmo. El valor de P_{max} se mantuvo constante en 0,98.

Problema	K_{max}	Its.	Optimo	Opt Local	% de Error	Tiempo
ch130	50	50	6110	6174.9	1.06	0:04:00
d198	25	50	15780	16043.2	1.67	0:12:00
pr226	50	40	80369	81218	1.05	0:20:00
pr264	66	40	49135	49604.2	0.95	0:22:00
rd400	100	40	15281	15664.4	2.51	1:25:00
gr431	25	50	171414	178637	4.21	0:35:00
gr431	100	50	171414	174974	2.08	2:00:00
pr439	100	40	107217	109529	2.16	1:20:00
d493	25	50	35002	36418	4.04	0:34:00
d493	110	40	35002	35422	1.19	2:00:00
u574	20	50	36905	39337	6.59	0:35:00
u574	30	50	36905	38856	5.29	0:53:00
u574	100	40	36905	37688	2.12	3:00:00
rat575	20	50	6773	7226	6.69	0:36:00
rat575	25	50	6773	7195	6.23	0:42:00
p654	100	50	34643	35221	1.67	4:00:00
gr666	25	50	294358	308649	4.86	1:35:00
gr666	160	40	294358	302752	2.85	8:30:00

Resultados promedio en 10 corridas del algoritmo

Nota: La columna tiempo es un valor promedio en horas que tarda el algoritmo, solo por cuestiones de comparación y de paso fundamentar nuestra razon de usar un algoritmo 2-Opt en ves de un 3-Opt, por ejemplo para el problema 'p654' el tiempo que tarda un el algoritmo 2-Opt (sobre una soluci'on inicial) es de 1 minuto 30 segundos, mientras que el algoritmo 3-Opt tarda mas de 18 minutos y aunque la solución que devuelve el algoritmo 3-Opt es en media mucho mejor que la proporcionada por el 2-Opt, preferimos usar un método determinista que si bien no es tan bueno como un 3-Opt o un K-Opt es muchísimo mas rápido.

6. CONCLUSIONES

El algoritmo con el cual se realizaron las pruebas no es mas que un híbrido de TSP basico que cuando queda atrapado en un óptimo local utiliza una Memoria de largo plazo (símil Búsqueda Tabú) para intentar escapar de dicho óptimo local.
Durante las pruebas realizadas se detecto que el numero de vecindades a utilizar depende del tamaño n del problema, por ejemplo para problemas de menos de 100 vértices (los datos no aparecen en la tabla) el algoritmo lograba alcanzar el óptimo global con solo 20 vecindades y 50 iteraciones, en cambio en problemas mas grandes con 30 vecindades y 50 iteraciones a lo sumo se lograba estar a un 6 % del óptimo global, aunque usar 100 (ó mas) vecindades en las pruebas nos pareció algo excesivo nos aseguro estar por debajo del 3 % de error en promedio. Mas tarde comprobamos empíricamente que usar $\frac{n}{4}$ vecindades nos aseguraba un error menor del 3 % y tambien que el numero de iteraciones no era un factor decisivo, pues intentamos mejorar resultados, en problemas grandes, aumentando el numero de iteraciones y manteniendo fijo el numero de vecindades pero lo único que conseguimos fue un mayor tiempo de ejecución (salvo raras veces), en cambio aumentar el numero de vecindades si que lograba mejoras. El numero de iteraciones es un factor con el que se puede jugar por que como ya dijimos no es un factor decisivo, pero hay tambíen un valor mínimo que puede tomar, por debajo de este valor por mas que usemos un numero adecuado de vecindades no llegamos a estar dentro del margen de error buscado. En nuestras pruebas procuramos un valor un poco alto para este parámetro asegurándonos de obtener buenos resultados, pero bien podríamos haber buscado valores adecuados (siempre a prueva y error) para lograr minimizar el tiempo de ejecución.

REFERENCIAS

[1] G. Dantzig, R. Fulkerson, S. Johnson, Solution of a Large Scale Traveling Salesman Problem, Paper P-510, The RAND Corporation, Santa Monica, California, [12 April] 1954 [published in journal of the Operations Research Society of America 2 (1954) 393[410].

[2] J.L. Bentley. Experiments on traveling salesman heuristics. Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 1990, 91 - 99.

[3] N. Christofides. Worst-case analysis of a new heuristic for the traveling salesman problem. Tech. Report 388, Carnegie Mellon University, Pittsburgh, USA, 1976.

[4] D.E. Rosenkrantz, R.E. Stearns, P.M. Lewis. An analysis of several heuristics for the traveling salesman problem. SIAM Journal on Computing, 1977, Vol.6, 563581.

[5] G.A. Croes. A method for solving traveling-salesman problems. Operations Research, 1958, Vol.6, 791 - 812.

[6] S. Lin. Computer solutions of the traveling salesman problem. Bell System Tech. Journal, 1965, Vol.44, 2245 - 2269.

[7] S. Lin, B.W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. Opera-tions Research, 1973, Vol.21, 498 516

[8] K. Mak, A. Morton. A modified Lin-Kernighan traveling salesman heuristic. ORSA Journal on Com-puting, 1992, Vol.13, 127 - 132.

[9] S. Kirkpatrick, C.D. Gelatt, Jr., M.P. Vecchi. Optimization by simulated annealing. Science, 1983, Vol.220, 671 - 680.

[10] J. Pepper, B. Golden, E. Wasil. Solving the traveling salesman problem with annealing-based heuristics: a computational study. IEEE Transactions on Systems, Man, and Cybernetics, Part A, 2002, Vol. 32, 72 - 77.

[11] C.-N. Fiechter. A parallel tabu search algorithm for large traveling salesman problems. Discrete Applied Mathematics, 1994, Vol.51, 243 267

[12] J. Knox. Tabu search performance on the symmetric traveling salesman problem. Computers & Operations Research, 1994, Vol.21, 867 - 876.

[13] B. Freisleben, P. Merz. A genetic local search algorithm for solving symmetric and a symmetric traveling salesman problems. Proceedings of the 1996 IEEE International Conference on Evolutionary Computation, Nagoya, Japan, 1996, 616 - 621.

[14] Heuristics for the Traveling Salesman Problem, Christian Nilsson Linkping University, pp1-6

[15] Pierre Hansen, Nenad Misdenovic. Variable Neighborhood Search, a Chapter of "Handbook of Metaheuristics"

[16] Nenad Mladenovi Pierre Hansen, Jack Brimberg Dragan Uro2evi. Variable neighborhood search, City University, London, March 8, 2007.

Héctor E. Núñez

Ricardo D. Quiroga

Nelson Efrain A. Cruz