que este problema de optimización en redes, al cual hemos definido particularmente como el PMCC en sistemas de redes de transporte de gas natural (*NP-completo*), pudo resolverse eficientemente al desarrollarse y aplicarse un algoritmo no exponencial (véase Apéndice D), el cual entregó soluciones de muy buena calidad, sobrepasando en mucho a los métodos tradicionales de programación no lineal existentes.

BIBLIOGRAFÍA

[1] R. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, EUA, 1957.

[2] D. P. Bertsekas. *Dynamic Programming: Deterministic and Stochastic Models.* Prentice-Hall, Englewood Cliffs, N. J., EUA, 1987.

[3] A. Brooke, D. Kendrick y A. Meeraus. *GAMS: A User's Guide, Release 2.25.* The Scientific Press, San Francisco, EUA, 1992.

[4] R. G. Carter. Pipeline optimization: Dynamic programming after 30 years. En *Proceedings of the PSIG Meeting*, pp. 1-19, Denver, EUA, Octubre 1998.

[5] D. Cobos Zaleta. *Modelos de Optimización Entera Mixta No Lineal en Sistemas de Transporte de Gas Natural.* Tesis de Maestría, Universidad Autónoma de Nuevo León, San Nicolás de los Garza, México, Noviembre 2003.

[6] D. Cobos-Zaleta y R. Z. Ríos-Mercado. A MINLP model for minimizing fuel consumption on natural gas pipeline networks. En *Proceedings of the XI-Latin-Ibero-American Conference on Operations Research,* artículo A48-01, pp. 1-9. Concepción, Chile, Octubre 2002.

[7] S. A. Cook. The Complexity of Theorem-proving Procedures. *Proceedings 3rd Annual ACM Symposium on Theory of Computing.* Association for Computing Machinery, pp. 151-158, New York, 1971.

[8] N. Creignou, S. Khanna y M. Sudan. *Complexity Classifications of Boolean Constraint Satisfaction Problems.* SIMM Monografías sobre Matemáticas Discretas y Aplicaciones. New Jersey, EUA, 2001.

[9] A. Díaz, F. Glover, H. M. Ghaziri, J. L. González, M. Laguna, P. Moscato y F. T. Tseng. *Optimización Heurística y Redes Neuronales.* Editorial Paraninfo, Madrid, España, 1996.

[10] H. J. Flores-Villarreal y R. Z. Ríos-Mercado. Computational experience with a GRG method for minimizing fuel consumption on cyclic natural gas networks. En N. E. Mastorakis, I. A. Stathopulos, C. Manikopoulos, G. E. Antoniou, V. M.

Mladenov y I. F. Gonos, editores, *Computational Methods in Circuits and Systems Applications*, pp. 90-94. WSEAS Press, Atenas, Grecia, 2003.

[11] H. J. Flores-Villarreal y R. Z. Ríos-Mercado. Efficient operation of natural gas pipeline networks: Computational finding of high quality solutions. En *Proceedings of the International Applied Business Research Conference,* artículo 352, pp.1-7. Acapulco, México, Marzo 2003.

[12] C. A. Floudas. *Nonlinear and Mixed-Integer Optimization Fundamentals and Applications*. Oxford University Press, New York, EUA, 1995.

[13] M. R. Garey y D. S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completeness*. W.H. Freeman, New York, EUA, 1979.

[14] B.J. Gilmour, C.A. Luongo y D.W. Schroeder. Optimization in natural gas transmission networks: A tool to improve operational efficiency. Reporte Técnico, Stoner Associates Inc., presentado en la Tercera Conferencia sobre optimización SIAM, Boston, EUA, Abril 1989.

[15] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, 5(3):533-549, 1986.

[16] F. Glover y M. Laguna. *Tabu Search*. Kluwer, Boston, EUA, 1997.

[17] F. Glover, M. Laguna. OpenTS – Java Tabu Search, sitio web http://faculty.bus.olemiss.edu/fglover/ http://www.worldscinet.com/ijitdm/01/sample/S0219622002000026.html

[18] I. E. Grossmann, J. Viswanathan, A. Vecchietti, R. Raman y E. Kalvelagen. *GAMS/DICOPT: A Discrete Continuos Optimization Peckage*. Carnegie Mellon University and GAMS Development Corporation,Washington, EUA, Noviembre 2001.

[19] R. Harder. OpenTS - Java Tabu Search, sitio web http://www.iHarder.net

[20] R. Horst, P. M. Pardalos y N. V. Thoai. *Introduction to Global Optimization*. Kluwer, Dordrecht, Holanda, 1995.

[21] J. T. Jefferson. Dynamic programming. *Oil and Gas Journal*, pp. 102-107, Mayo 1961.

[22] S. Kim, R. Z. Ríos-Mercado y E. A. Boyd. A heuristic for minimum cost steady-state gas transmission networks. En *Proceedings of the 25th International*

Conference on Computers & Industrial Engineering, New Orleans, EUA, Marzo 1999.

[23] S. Kim, R. Z. Ríos-Mercado y E. A. Boyd. Heuristics for minimum cost steady-state gas transmission networks. En M. Laguna y J. L. González-Velarde, editores, *Computing Tools for Modeling, Optimization, and Simulation*, capítulo 11, pp. 203–213. Kluwer, Boston, EUA, 2000.

[24] H. S. Lall y P. B. Percell. A dynamic programming based gas pipeline optimizer. En A. Bensoussan y J. L. Lions, editores, *Analysis and Optimization of Systems*, pp. 123-132, Springer-Verlag, Berlín, Alemania, 1990.

[25] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan y D. Shmoys. Sequencing and scheduling: Algorithms and complexity. En S. S. Graves, A. H. G. Rinnooy Kan y P. Zipkin, editores, *Handbook in Operations Research and Management Science, Vol. 4: Logistics of Production and Inventory*, 445-522. North-Holland, New York, 1993.

[26] A. V. Levitin. *Introduction to the Design and Analysis of Algorithms*. Addison-Wesley, N. Y., EUA, 2002.

[27] B. Melián, J. A. Moreno y J. M. Moreno. Metaheuristics: A global view. *Inteligencia Artificial. Revista Iberoamericana de Inteligencia Artificial*, 6(19):7-28, 2003.

[28] A. J. Osiadacz. *Simulation and Analysis of Gas Networks*. Gulf Publishing Company, Houston, EUA, 1987.

[29] A. J. Osiadacz. Dynamic optimization of high pressure gas networks using hierarchical systems theory. En *Proceedings of the 26th PSIG Annual Meeting*, San Diego, EUA, Octubre 1994.

[30] A. J. Osiadacz y M. Górecki. Optimization of pipe sizes for distribution gas network design. En *Proceedings of the 27th PSIG Annual Meeting*, Albuquerque, EUA, Octubre 1995.

[31] A. J. Osiadacz y S. Swierczewski. Optimal control of gas transportation systems. En *Proceedings of the 3rd IEEE Conference on Control Applications*, pp. 795–796, Agosto 1994.

[32] C. H. Papadimitriou. *Computational Complexity.* Addison-Wesley, N. Y., EUA, 1995.

[33] C. H. Papadimitriou y K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Publicaciones Dover, EUA, 1999.

[34] P. B. Percell y M. J. Ryan. Steady-state optimization of gas pipeline network operation. En *Proceedings of the 19th PSIG Annual Meeting*, Tulsa, EUA, Octubre 1987.

[35] D.T. Pham y D. Karboga. *Intelligent Optimisation Techniques, Genetic Algorithms, Tabu Search, Simulated Annealing and Neural Networks.* Springer, New York, EUA, 2000.

[36] M. Pinedo. *Scheduling, Theory, Algorithms, and Systems*. Segunda Edición. Prentice-Hall, Englewood Cliffs, EUA, 2002.

[37] V. J. Rayward-Smith. *A First Course in Computability*. Blackwell, 1986.

[38] M. Resende y J. L. González-Velarde. GRASP: Procedimientos de búsqueda miopes aleatorizados y adaptativos. *Inteligencia Artificial. Revista Iberoamericana de Inteligencia Artificial*, 6(22):42-61, 2003.

[39] M. G. C. Resende y C. C. Ribeiro. Greedy randomized adaptive search procedures. En F. Glover y G. G. Kochenberger, editores, *Handbook of Metaheuristics*, Kluwer, Boston, EUA, 2003.

[40] R. Z. Ríos-Mercado. Natural gas pipeline optimization. En P. M. Pardalos y M. G. C. Resende, editores, *Handbook of Applied Optimization*, capítulo 18.8.3, pp. 813-825. Oxford University Press, New York, EUA, 2002.

[41] R. Z. Ríos-Mercado, S. Wu, L. R. Scott y E. A Boyd. A reduction technique for natural gas transmission network optimization problems. *Annals of Operations Research*, 117(1-4):217-234, 2002.

[42] D.W. Schroeder. Hydraulic analysis in the natural gas industry. En J. J.-W. Chen y A. Mital, editores, *Advances in Industrial Engineering Applications and Practice I*, pp. 960-965, Cincinnati, EUA, 1996.

[43] Y. Villalobos Morales. *Pre-procesamiento de un Problema de Optimización de Redes de Gas Natural.* Tesis de Maestría, Universidad Autónoma de Nuevo

León, San Nicolás de los Garza, México, Octubre 2002.

[44] Y. Villalobos-Morales, D. Cobos-Zaleta, H. J. Flores-Villarreal, C. Borraz-Sánchez y R.Z. Ríos-Mercado. On NLP and MINLP formulations and preprocessing for fuel cost minimization of natural gas transmission networks. En *Proceedings of the 2003 NSF Design, Service and Manufacturing Grantees and Research Conference*. Birmingham, Alabama, EUA, Enero 2003.

[45] Y. Villalobos Morales y R. Z. Ríos Mercado. Pre-procesamiento efectivo de un problema de minimización de combustible en sistemas de transporte de gas natural. Reporte Técnico PISIS-2002-01, FIME, UANL, San Nicolás de los Garza, NL, Noviembre 2002.

[46] P. J. Wong y R. E. Larson. Optimization of natural gas pipeline systems via dynamic programming. *IEEE Transactions on Automatic Control,* AC-13(5):475-481, 1968.

[47] S. Wu. *Steady-State Simulation and Fuel Cost Minimization of Gas Pipeline Networks*. Tesis doctoral, University of Houston, Houston, EUA, Agosto 1998.

[48] S. Wu, E. A. Boyd y L. R. Scott. Minimizing fuel consumption at gas compressor stations. En J. J.-W. Chen y A. Mital, editores, *Advances in Industrial Engineering Applications and Practice I*, pp. 972–977, Cincinnati, EUA, 1996.

[49] S. Wu, R. Z. Ríos-Mercado, E. A. Boyd y L. R. Scott. Model relaxations for the fuel cost minimization of steady-state gas pipeline networks. *Mathematical and Computer Modeling,* 31(2-3): 197-220, 2000.

[50] H. I. Zimmer. Calculating optimum pipeline operations. Reporte Técnico, Compañía de Gas Natural El Paso, 1975. Presentado en la Conferencia de Transmisión AGA, 1975.

LISTA DE TABLAS

APÉNDICE

LISTA DE FIGURAS

APÉNDICE

APÉNDICE

APÉNDICE A: Evaluación del Parámetro *Iter_max* de NONDP_TS

En las Figuras A1 y A2 se muestran las gráficas de desempeño del procedimiento NONDP_TS sobre la topología net-c-6c2-C1, con 100 y 500 iteraciones máximas, respectivamente.

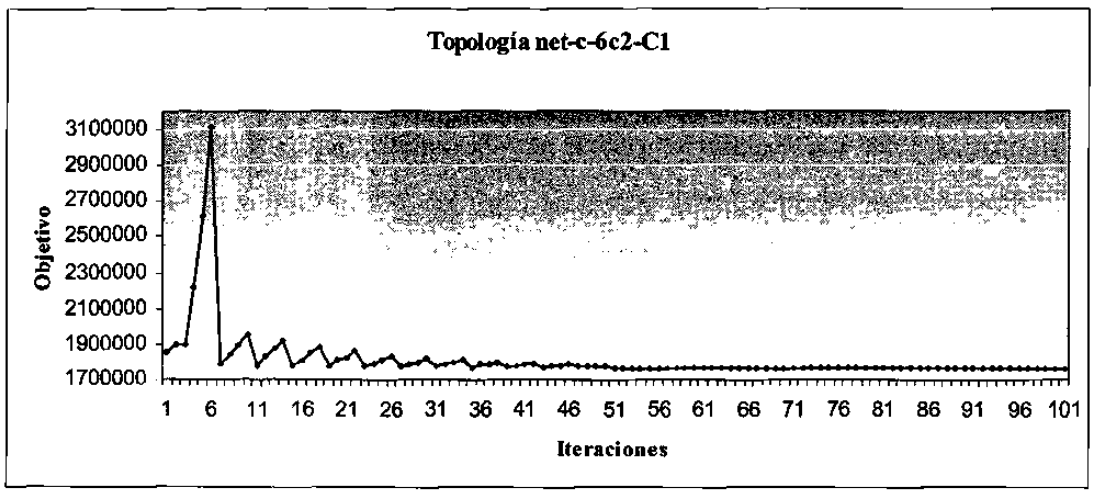

Figura A1. Convergencia del procedimiento NONDP_TS en instancia net-c-6c2-C1 con 100 iteraciones.

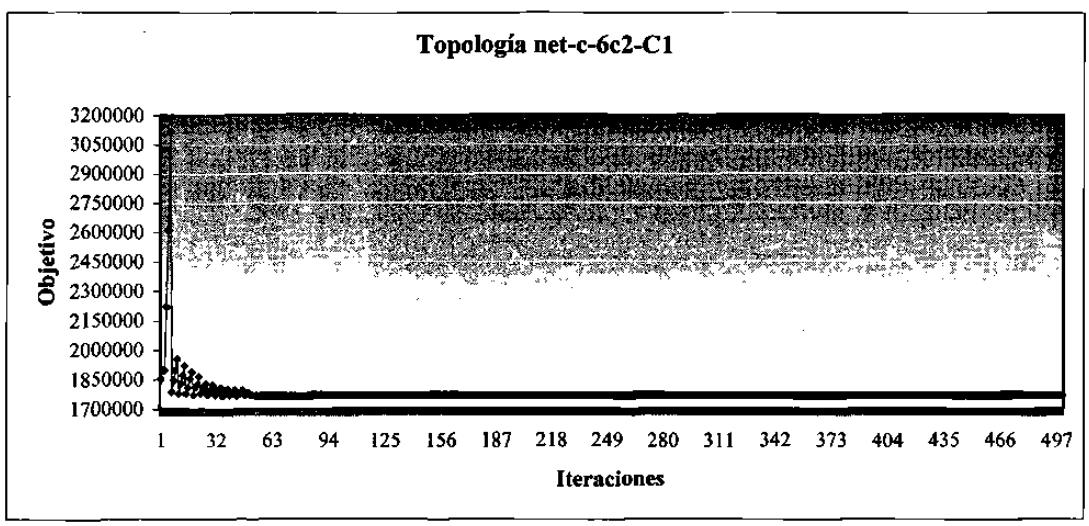

Figura A2. Convergencia del procedimiento NONDP_TS en instancia net-c-6c2-C1 con 500 iteraciones.

Observaciones	100 Iter (Fig. A1)	500 Iter (Fig. A2)
Valor inicial	1852252.54	1852252.54
Mejor valor encontrado	1765178.63	1765178.63
Mejor iteración	51	51
Tiempo de ejecución	3 hrs 06 min 35 seg (11195 seg)	12 hrs 22 min 11 seg (44531 seg)

Tabla A1. Observaciones de las gráficas de desempeño de TS.

En la Tabla A1 se presentan los resultados observados en las gráficas de desempeño de las Figuras A1 y A2 con 100 y 500 iteraciones, respectivamente. En la primera columna se muestran las medidas observadas: valor inicial, mejor valor encontrado, mejor iteración y tiempo de ejecución. En la segunda y tercera columnas se muestran los valores resultantes en la observación con 100 y 500 iteraciones, respectivamente.

Como podemos ver en la Tabla A1, el único valor modificado en los experimentos de evaluación es el tiempo de cómputo invertido, con una gran diferencia en la ejecución con 500 iteraciones. Resultados similares se presentaron en las evaluaciones de toda la amplia gama de la base de datos. Por lo tanto, se concluye que cantidad no significa calidad, tomando de esta manera para las evaluaciones posteriores del procedimiento NONDP_TS un número máximo de 100 iteraciones.

APÉNDICE B: Evaluación de Δp = {1, 5} en el Procedimiento NONDP_TS

Este experimento consistió en determinar en el procedimiento heurístico NONDP_TS de búsqueda tabú el uso del parámetro de discretización del algoritmo NDP con una de Δp ={1, 5} unidades PSIG; esto con la finalidad de medir el tiempo y calidad de las soluciones óptimas encontradas. Asimismo, en esta evaluación se estableció para el procedimiento heurístico NONDP_TS los siguientes parámetros estáticos:

- *Nei_Size* = 50 — Tamaño del Vecindario V(x)
- *Ttenure* = 10 — Tamaño de *tabu_list*
- *Iter_max* = 100 — Número máximo de iteraciones
- $\Delta p^{Vx} = 10$ — Dispersión de la Vecindad V(x)

En la Tabla B1 se muestran los resultados obtenidos con una Δp=5 unidades de discretización, presentando en la primera columna las instancias probadas, seguidas del tiempo de cómputo invertido, la iteración de la mejor solución encontrada, el valor objetivo de la solución NDP simple y el mejor valor encontrado, así como el porcentaje de mejora con respecto a la solución NDP simple. De manera similar se presentan en la Tabla B2 los resultados obtenidos con una Δp=1 unidades de discretización.

Finalmente, en la Tabla B3 se presentan los resultados obtenidos de la comparación realizada con las dos discretización en cuestión, presentado en la primera columna las instancias probadas, seguida por dos columnas donde se muestran el tiempo de cómputo y el mejor valor objetivo encontrado para cada discretización, y así mostrar en la última columna el porcentaje de mejoramiento de las soluciones obtenidas con una discretización de Δp=1 en comparación con una mayor discretización de Δp=5. Observando cómo el esfuerzo computacional se incrementa considerablemente con una Δp=1, ya que se habla de horas de diferencia con respecto a Δp=5. Por lo tanto, para efecto de las evaluaciones computacionales del procedimiento heurístico NONDP_TS presentado en la Sección 5.3.3, se propone utilizar una discretización de Δp=5 unidades PSIG, con el objetivo de minimizar los recursos computacionales de las mismas.

Topología	Δp = 5 PSIG				
	Tiempo (min:seg)	Mejor Iter	Valor objetivo inicial	Mejor valor encontrado	% Óptimo
net-c-6c2-C1	15:32	12	1,884,060.48	1,786,399.22	5.47
net-c-6c2-C3	35:38	11	1,041,265.76	972,892.87	7.03
net-c-6c2-C4	28:41	35	744,261.08	636,148.21	16.99
net-c-6c2-C5	28:55	87	2,803,372.66	2,717,573.08	3.16
net-c-6c2-C6	26:43	12	3,561,310.69	3,561,310.69	0.00
net-c-6c2-C7	11:44	1	850,674.93	850,674.93	0.00
net-c-6c2-C8	14:46	47	1,468,605.58	1,405,226.98	4.51
net-c-6c2-C9	11:20	1	1,294,408.69	1,294,408.69	0.00
net-c-10c3-C2	22:55	57	5,483,509.96	4,831,979.07	13.48
net-c-10c3-C3	47:10	35	4,081,328.16	3,767,167.75	8.33
net-c-10c3-C4	68:39	62	4,250,905.33	4,094,274.35	3.82
net-c-10c3-C5	63:54	38	7,146,123.19	6,153,153.87	16.13
net-c-10c3-C6	59:24	92	10,996,571.93	10,862,847.37	1.23
net-c-10c3-C8	48:17	96	5,265,444.54	5,195,376.60	1.34
net-c-10c3-C9	22:55	1	3,258,260.47	3,258,260.47	0.00

Tabla B1. Resultados computacionales de las topologías probadas con la heurística NONDP_TS con una Δp = 5.

Topología	Δp = 1 PSIG				
	Tiempo (hrs:min. seg)	Mejor Iter	Valor objetivo Inicial	Mejor valor encontrado	% Óptimo
net-c-6c2-C1	3:12.48	27	1,852,252.53	1,786,399.22	3.68
net-c-6c2-C3	4:04.90	34	1,020,842.29	907,728.31	12.46
net-c-6c2-C4	6:32:09	30	653,675.58	527,513.93	19.3
net-c-6c2-C5	3:53.25	27	2,716,611.07	2,646,948.81	2.63
net-c-6c2-C6	4:18.05	28	3,442,838.76	3,407,748.11	1.03
net-c-6c2-C7	4:42.33	62	788,218.02	726,355.04	8.52
net-c-6c2-C8	4 :31.55	51	1,405,443.48	1,356,798.94	3.59
net-c-6c2-C9	4:28.44	47	1,243,179.03	1,209,864.45	2.75
net-c-10c3-C2	3:58.13	25	4,869,023.74	4,714,008.54	3.28
net-c-10c3-C3	3:55.56	2	3,978,465.67	3,785,841.38	5.09
net-c-10c3-C4	4:44.03	26	1,082,614.93	888,915.07	21.79
net-c-10c3-C5	6:52.15	30	6,624,644.75	5,890,916.41	12.46
net-c-10c3-C6	6:42.13	41	10,766,001.9	10,637,203.60	1.21
net-c-10c3-C8	3:46.55	1	3,316,383.49	3,316383.49	0.00
net-c-10c3-C9	5:17:22	56	3,039,766.16	2,921,299.37	4.06

Tabla B2. Resultados computacionales de las topologías probadas con la heurística NONDP_TS con una Δp = 1.

Topología	Δp = 5 PSIG		Δp = 1 PSIG		% Opt. Δp = 1
	Tiempo hh:mm:ss	Mejor valor encontrado	Tiempo hh:mm:ss	Mejor valor encontrado	
net-c-6c2-C1	00:15:48	1,786,399.22	03:12.48	1,786,399.22	0.0
net-c-6c2-C2	00:15:32	1,786,399.22	03:21.52	1,786,399.22	0.0
net-c-6c2-C3	00:35:38	972,892.87	03:04.90	907,728.31	7.17
net-c-6c2-C4	00:28:41	636,148.21	06:32:09	527,513.93	17.07
net-c-6c2-C5	00:28:55	2,717,573.08	03:53.25	2,646,948.81	2.66
net-c-6c2-C6	00:26:43	3,561,310.69	04:18.05	3,407,748.11	4.5
net-c-6c2-C7	00:11:44	850,674.93	04:42.33	726,355.04	17.11
net-c-6c2-C8	00:14:46	1,405,226.98	04:31.55	1,356,798.94	3.56
net-c-6c2-C9	00:11:20	1,294,408.69	04:28.44	1,209,864.45	6.98
net-c-10c3-C2	00:22:55	4,831,979.07	03:58.13	4,714,008.54	2.5
net-c-10c3-C3	00:47:10	3,785,841.38	03:55.56	3,767,167.75	0.49
net-c-10c3-C5	01:03:54	6,153,153.87	06:52.15	5,890,916.41	4.45
net-c-10c3-C6	00:59:24	10,862,847.37	06:42.13	10,637,203.60	2.12
net-c-10c3-C8	00:48:17	5,195,376.60	03:46.55	3,316383.49	36.16
net-c-10c3-C9	00:22:55	3,258,260.47	05:17:22	2,921,299.37	11.53

Tabla B3. Comparación de los resultados obtenidos por dos tamaños de mallas: Δp = {5, 1}.

APÉNDICE C: Evaluación de la Convergencia del Procedimiento NONDP_TS sobre dos Parámetros de TS: *Ttenure* y *Nei_Size*

Este experimento se realizó con el objetivo de mostrar las gráficas de desempeño del procedimiento NONDP_TS (ver Sección 5.3.3), presentando así una evaluación computacional sobre dos parámetros esenciales de la Búsqueda Tabú, a saber: Tamaño de la *lista tabú* para un ***Ttenure*** = **{5, 8, 10}** y tamaño del vecindario V(x) para un ***Nei_Size*** = **{20, 30}**. Estas gráficas muestran el mejor valor objetivo encontrado en cada iteración.

Las gráficas de desempeño de las evaluaciones realizadas sobre 4 diferentes topologías cíclicas tomadas de la base de datos pueden verse a continuación en el orden que nos muestra la Tabla C1.

Topología	Tamaño de la lista tabú *Ttenure*	Tamaño del Vecindario	
		Nei_Size = 20	*Nei_Size = 30*
net-c-6c2-C4	5	Figura C-1a	Figura C-2a
	8	Figura C-1b	Figura C-2b
	10	Figura C-1c	Figura C-2c
net-c-6c2-C5	5	Figura C-3a	Figura C-4a
	8	Figura C-3b	Figura C-4b
	10	Figura C-3c	Figura C-6c
net-c-6c2-C8	5	Figura C-5a	Figura C-6a
	8	Figura C-5b	Figura C-6b
	10	Figura C-5c	Figura C-6c
net-c-10c3-C2	5	Figura C-7a	Figura C-8a
	8	Figura C-7b	Figura C-8b
	10	Figura C-7c	Figura C-8c

Tabla C1. Gráficas de convergencia del procedimiento NONDP_TS.

En las gráficas presentadas por la Tabla C1 se pudieron observar los siguientes puntos:

- En todas las gráficas de desempeño puede verse la convergencia de una Búsqueda Tabú tradicional hacia la mejor solución encontrada.
- En todas las topologías evaluadas con un tamaño *Ttenure* = 10 de la lista

tabú, las gráficas respectivas demostraron tener una mayor diversificación de las mejores soluciones obtenidas en cada iteración; aunque éstas convergieron en un número mayor de iteraciones que con un tamaño de lista menor.

- En todas las topologías evaluadas sobre los dos tamaños del vecindario *Nei_Size* = {20, 30}, para un mismo tamaño de *lista tabú*, las gráficas de desempeño se comportaron de manera similar en cuanto a la mejor iteración presentada por el procedimiento heurístico NONDP_TS; por lo que, la implementación de cualquiera de estos dos diferentes valores no se considera un factor predominante en la exploración del espacio de búsqueda, pero sí en un factor determinante en el incremento de los recursos computacionales requeridos por un tamaño del vecindario mayor. Cabe aclarar, que aún cuando solo se presentan tres tamaños diferentes del vecindario en este trabajo (20, 30 y 40), para la conclusión de este experimento se agregó la evaluación de dos diferentes tamaños más {10 y 50}, presentando resultados similares a los obtenidos por esta evaluación.

- Todas aquellas topologías evaluadas sobre los parámetros combinados de *Ttenure* = *8* y *Nei_Size* = *20*, las gráficas respectivas mostraron que el procedimiento heurístico NONDP_TS convergió mucho antes que las otras combinaciones probadas, resultando en la misma solución en un menor tiempo de ejecución; encontrando además, una mayor exploración del espacio de búsqueda.

Finalmente, en base a las observaciones realizadas en este análisis, se concluyó que los valores de *Ttenure* = 8 y *Nei_Size* = 20 representaron un mayor beneficio tanto en la exploración del espacio de búsqueda como en la minimización de los recursos computacionales requeridos; por lo que, estos valores son fijados en todas las experimentaciones para evaluar la calidad de las soluciones obtenidas por el procedimiento NONDP_TS.

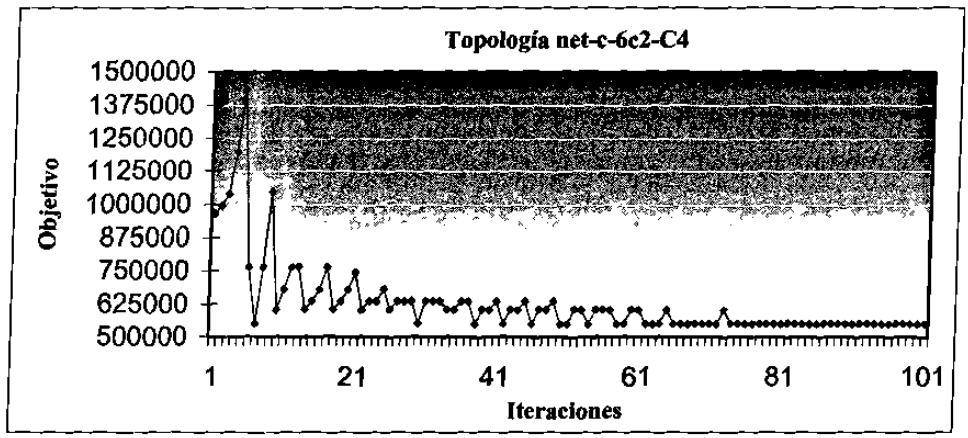

Figura C-1a: Convergencia de NONDP_TS en instancia net-c-6c2-C4 para *Nei_Size = 20* y *Ttenure = 5*.

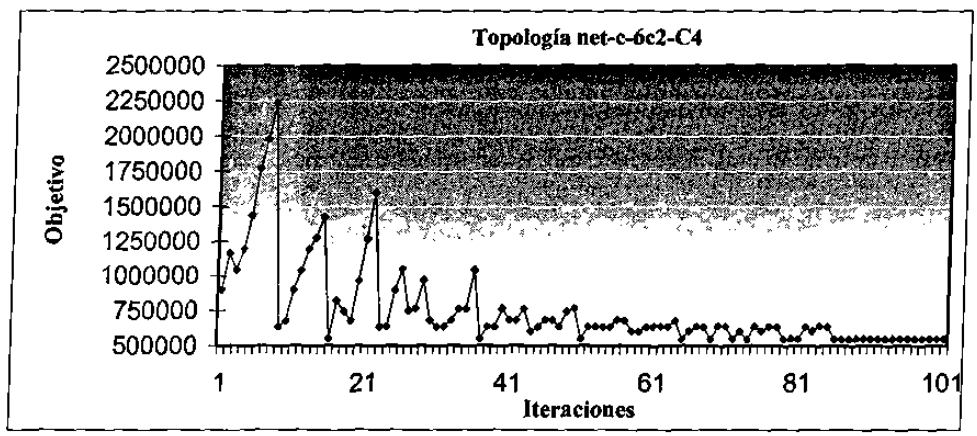

Figura C-1b: Convergencia de NONDP_TS en instancia net-c-6c2-C4 para *Nei_Size = 20* y *Ttenure = 8*.

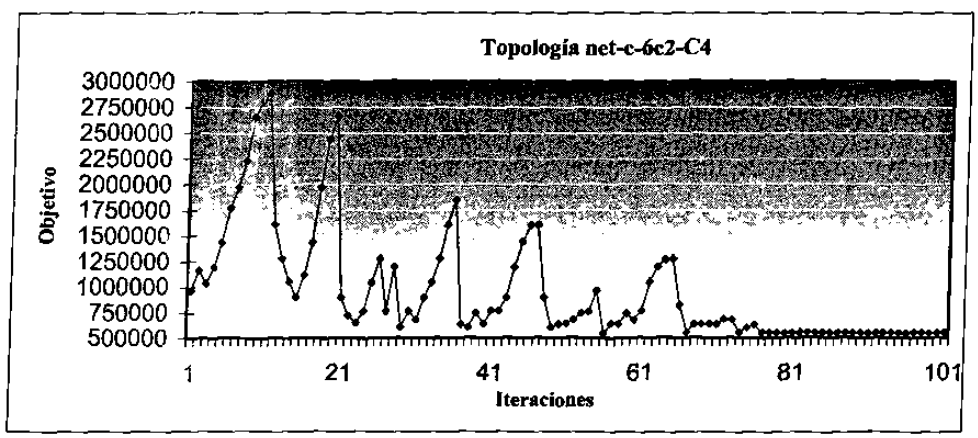

Figura C-1c: Convergencia de NONDP_TS en instancia net-c-6c2-C4 para *Nei_Size = 20* y *Ttenure = 10*.

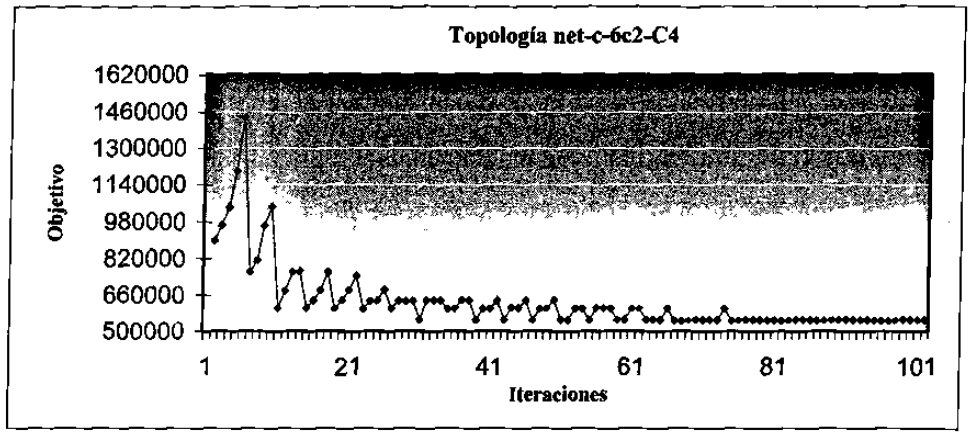

Figura C-2a: Convergencia de NONDP_TS en instancia net-c-6c2-C4 para *Nei_Size = 30* y *Ttenure = 5.*

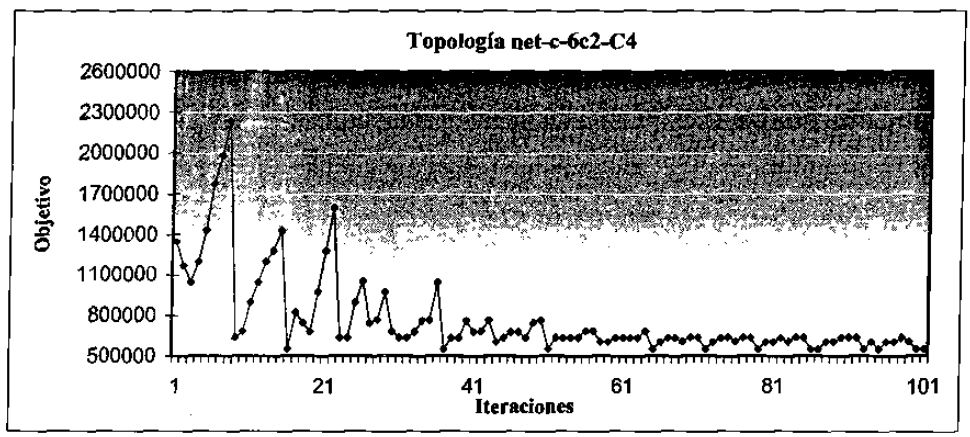

Figura C-2b: Convergencia de NONDP_TS en instancia net-c-6c2-C4 para *Nei_Size = 30* y *Ttenure = 8.*

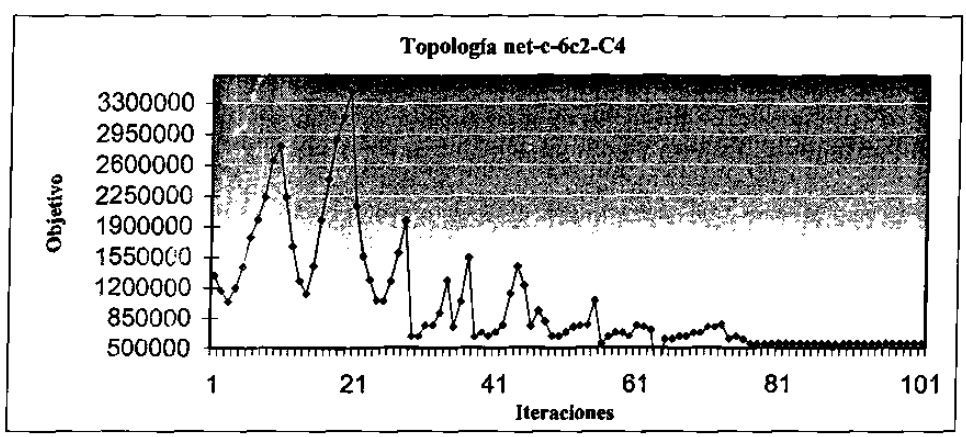

Figura C-2c: Convergencia de NONDP_TS en instancia net-c-6c2-C4 para *Nei_Size = 30* y *Ttenure = 10.*

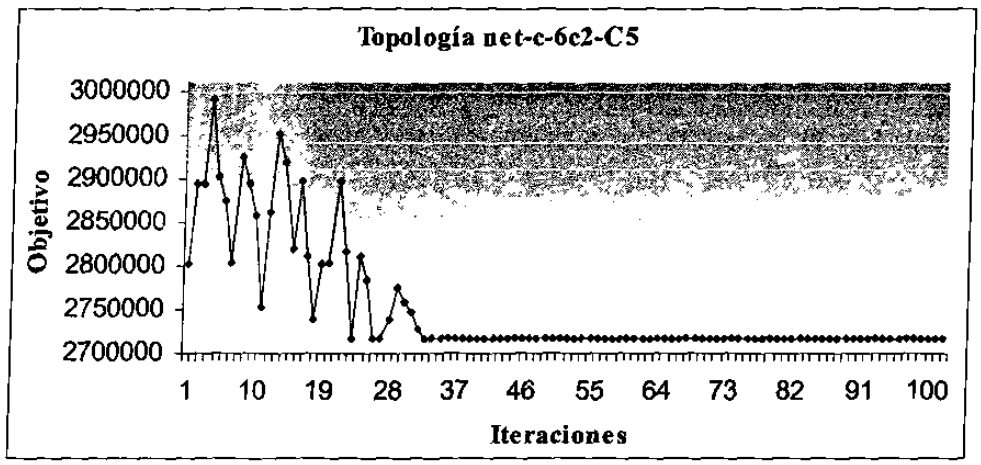

Figura C-3a: Convergencia de NONDP_TS en instancia net-c-6c2-C5 para *Nei_Size = 20* y *Ttenure = 5.*

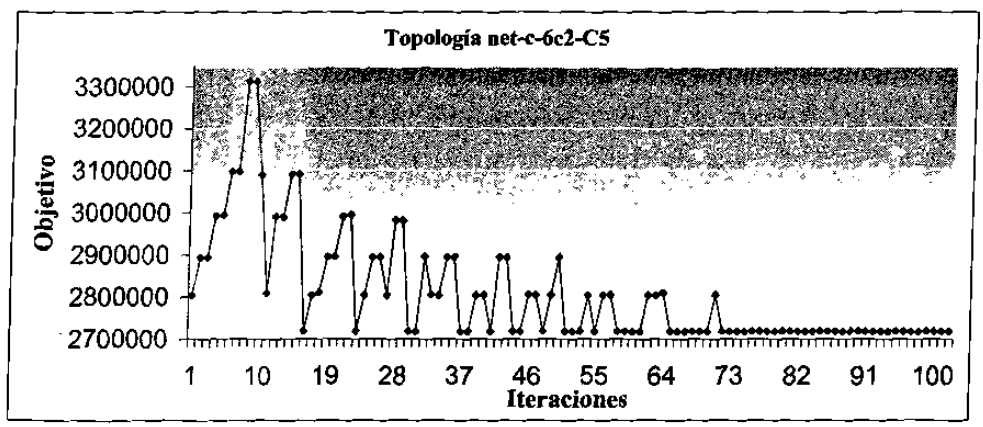

Figura C-3b: Convergencia de NONDP_TS en instancia net-c-6c2-C5 para *Nei_Size = 20* y *Ttenure = 8.*

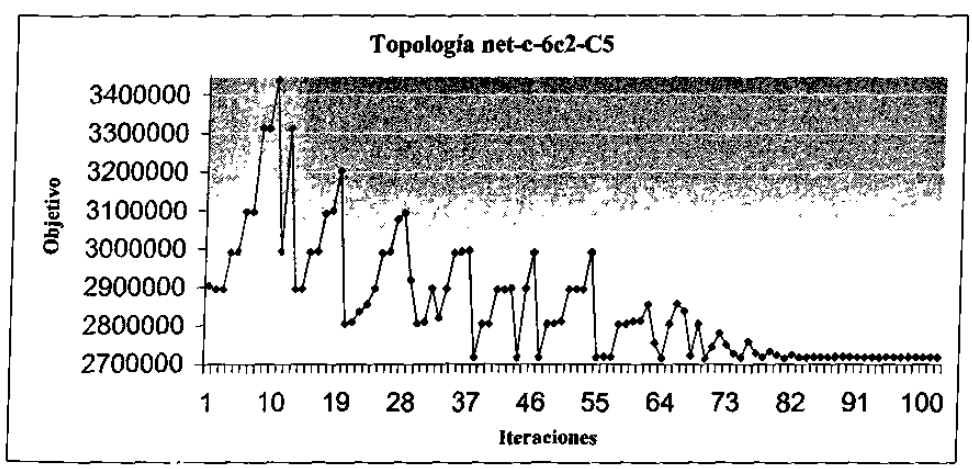

Figura C-3c: Convergencia de NONDP_TS en instancia net-c-6c2-C5 para *Nei_Size = 20* y *Ttenure = 10.*

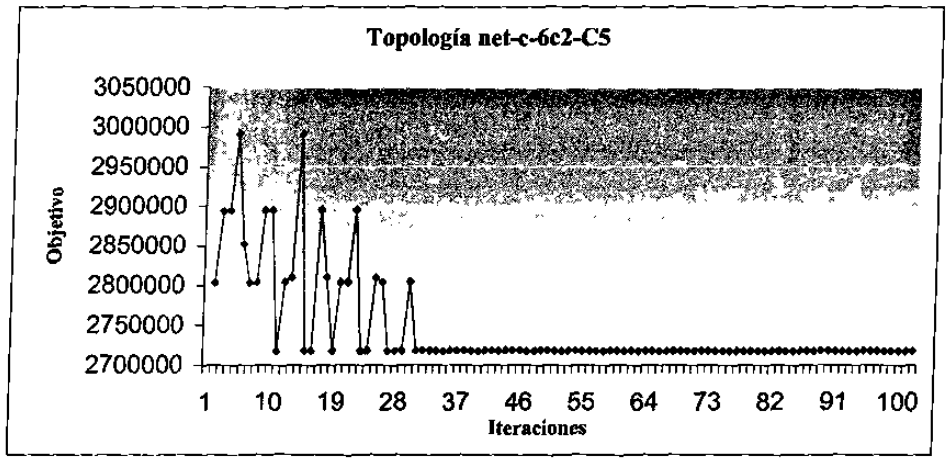

Figura C-4a: Convergencia de NONDP_TS en instancia net-c-6c2-C5 para *Nei_Size = 30* y *Ttenure = 5.*

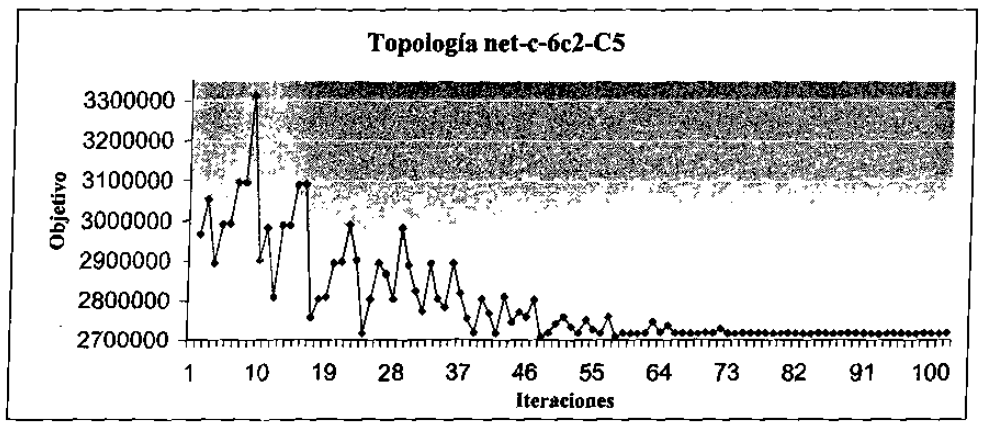

Figura C-4b: Convergencia de NONDP_TS en instancia net-c-6c2-C5 para *Nei_Size = 30* y *Ttenure = 8.*

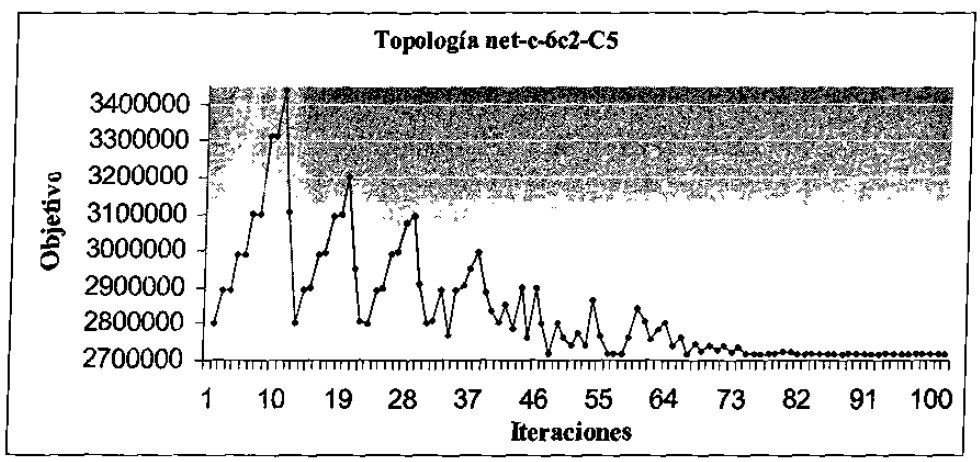

Figura C-4c: Convergencia de NONDP_TS en instancia net-c-6c2-C5 para *Nei_Size = 30* y *Ttenure = 10.*

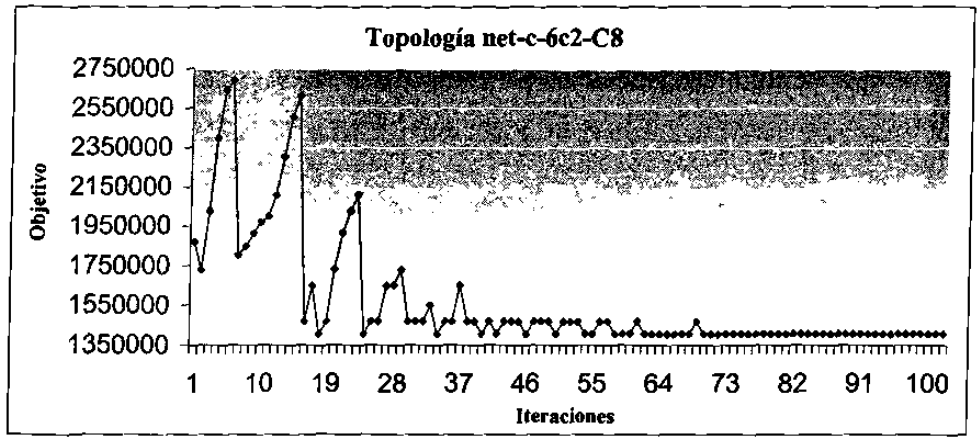

Figura C-5a: Convergencia de NONDP_TS en instancia net-c-6c2-C8 para *Nei_Size = 20* y *Ttenure = 5*.

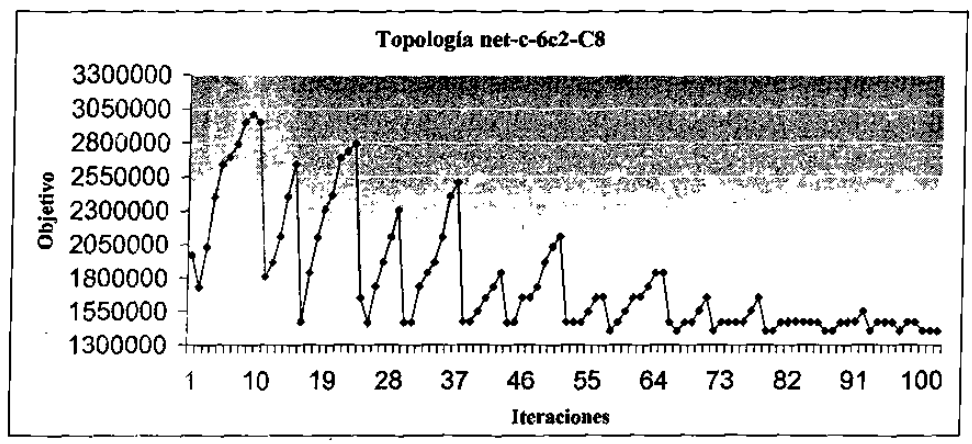

Figura C-5b: Convergencia de NONDP_TS en instancia net-c-6c2-C8 para *Nei_Size = 20* y *Ttenure = 8*.

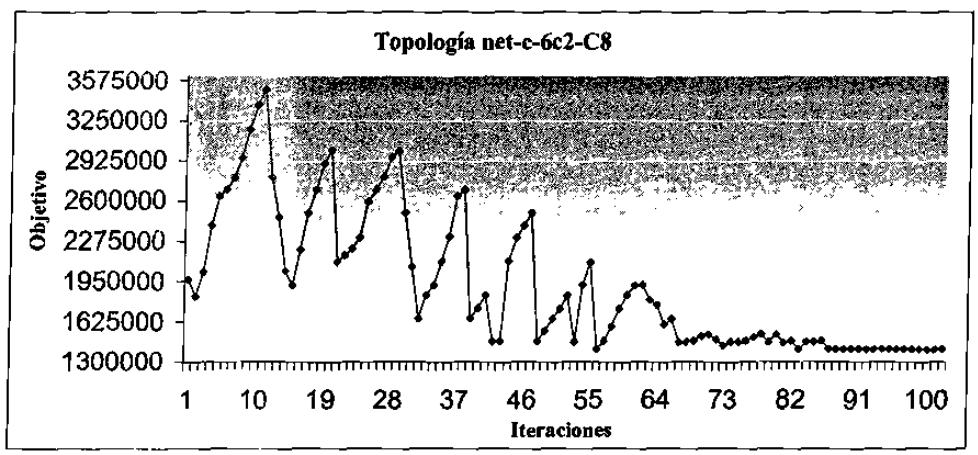

Figura C-5c: Convergencia de NONDP_TS en instancia net-c-6c2-C8 para *Nei_Size = 20* y *Ttenure = 10*.

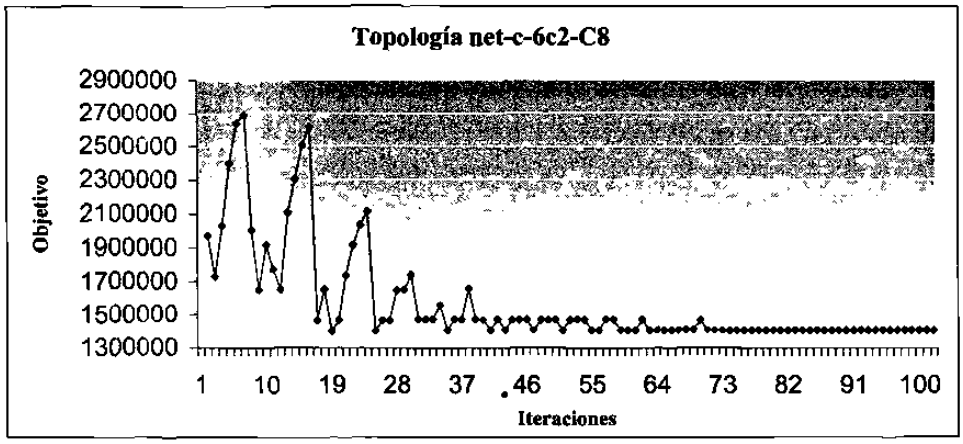

Figura C-6a: Convergencia de NONDP_TS en instancia net-c-6c2-C8 para *Nei_Size = 30* y *Ttenure = 5*.

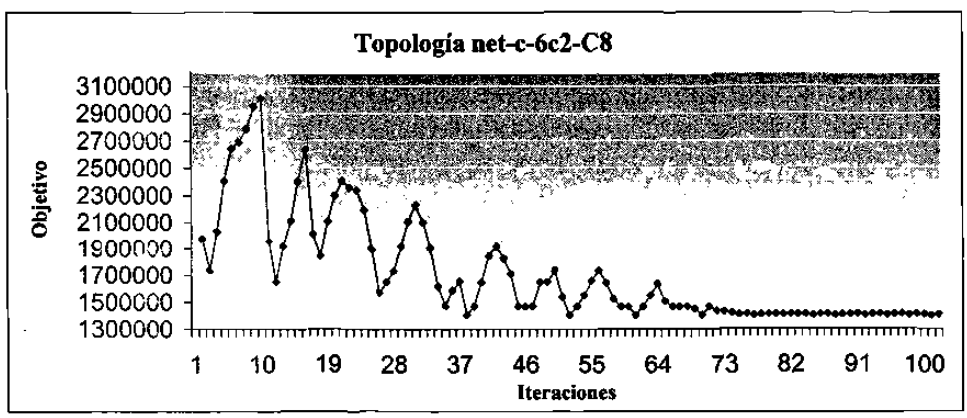

Figura C-6b: Convergencia de NONDP_TS en instancia net-c-6c2-C8 para *Nei_Size = 30* y *Ttenure = 8*.

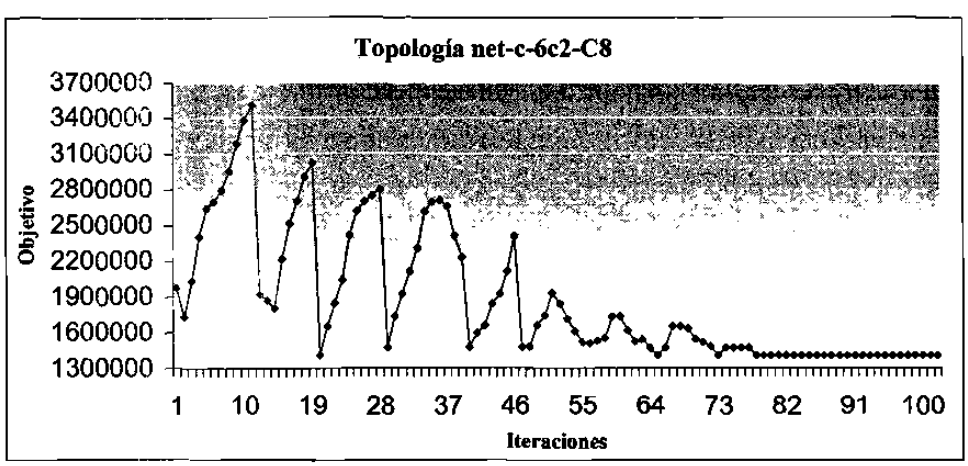

Figura C-6c: Convergencia de NONDP_TS en instancia net-c-6c2-C8 para *Nei_Size = 30* y *Ttenure = 10*.

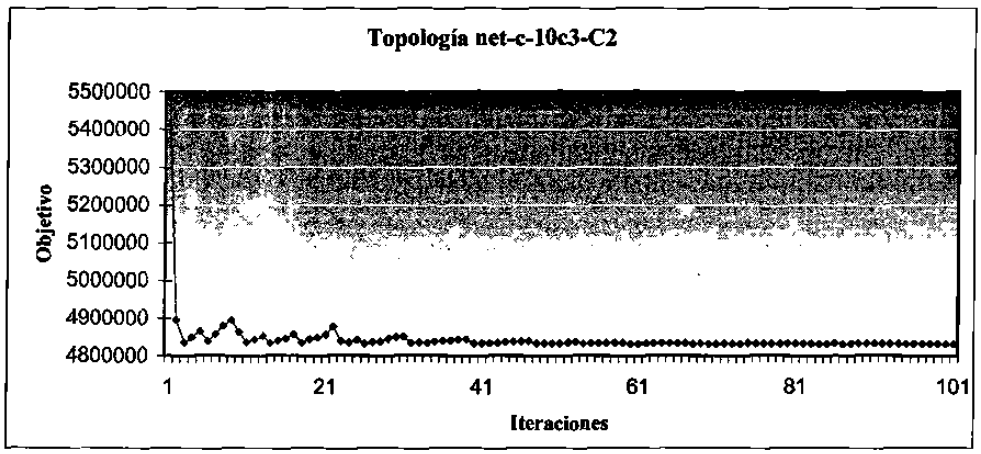

Figura C-7a: Convergencia de NONDP_TS en instancia net-c-10c3-C2 para *Nei_Size = 20* y *Ttenure = 5*.

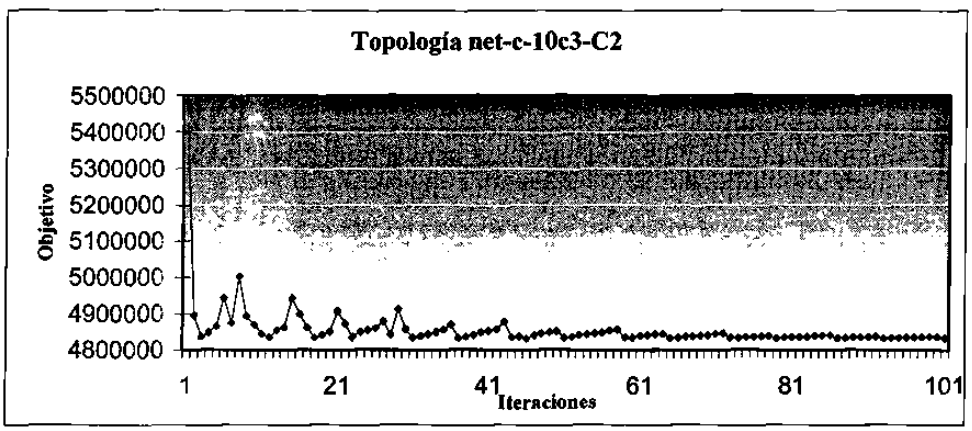

Figura C-7b: Convergencia de NONDP_TS en instancia net-c-10c3-C2 para *Nei_Size = 20* y *Ttenure = 8*.

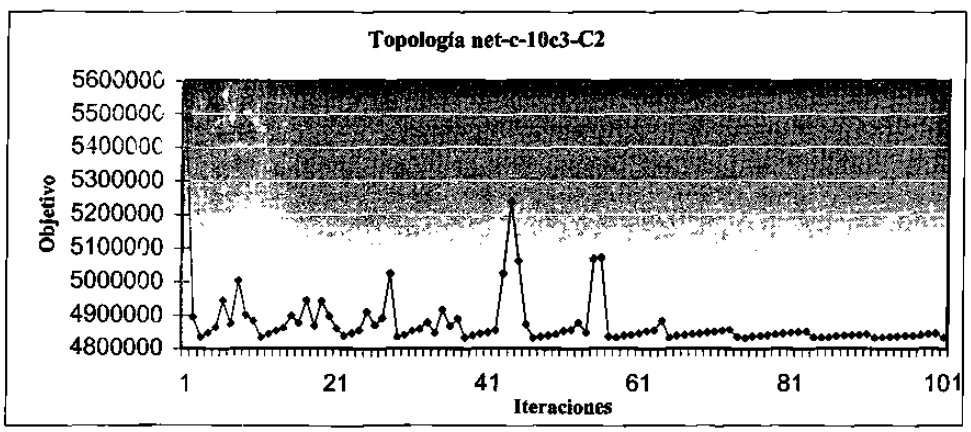

Figura C-7c: Convergencia de NONDP_TS en instancia net-c-10c3-C2 para *Nei_Size = 20* y *Ttenure = 10*.

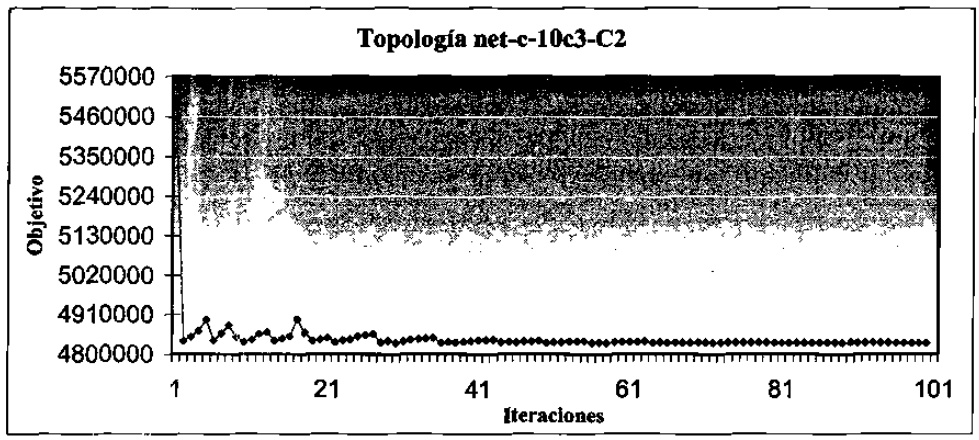

Figura C-8a: Convergencia de NONDP_TS en instancia net-c-10c3-C2 para *Nei_Size = 30* y *Ttenure = 5*.

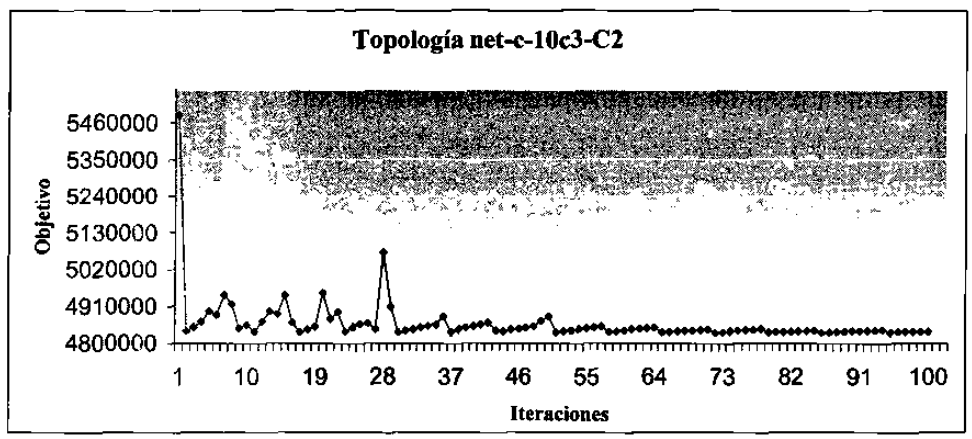

Figura C-8b: Convergencia de NONDP_TS en instancia net-c-10c3-C2 para *Nei_Size = 30* y *Ttenure = 8*.

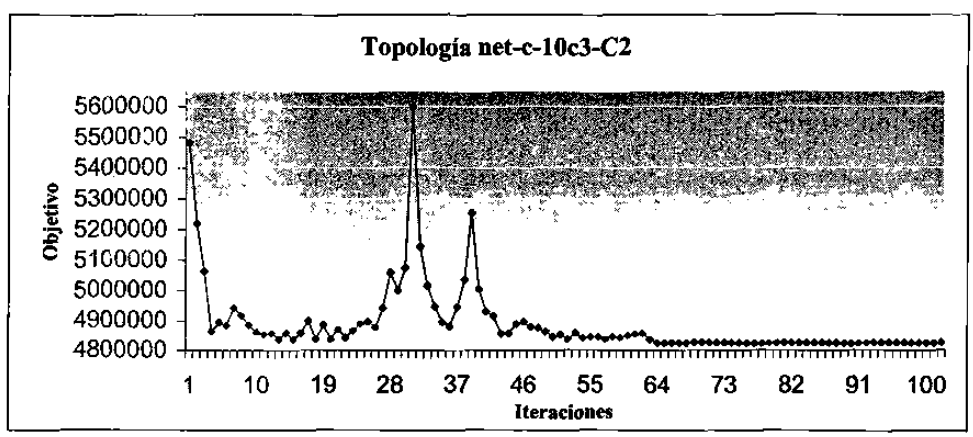

Figura C-8c: Convergencia de NONDP_TS en instancia net-c-10c3-C2 para *Nei_Size = 30* y *Ttenure = 10*.

APÉNDICE D: Teoría de Complejidad

En este apéndice se presentan los conceptos relativos al tema de la Teoría de Complejidad, a saber: complejidad y orden de los algoritmos. También se describen las diferentes clases de problemas de decisión (**P** y **NP**) y su correspondencia con los problemas de optimización.

D.1 Complejidad de los Algoritmos

La Teoría de Complejidad de los algoritmos concierne a la identificación de los problemas que, desde una perspectiva computacional, son fáciles de resolver y aquéllos que son difíciles de resolver (ver [13] y [37]).

En la Ciencia de la Computación, al hablar de la teoría de complejidad, no se está refiriendo a la dificultad que se tendría para diseñar un programa. Esta teoría está vinculada con la identificación de los algoritmos que son eficientes y aquéllos que son ineficientes desde un punto de vista computacional. Desde una perspectiva general, cuando se analiza y compara el desempeño de un algoritmo, su eficiencia se fija en términos de los recursos de cómputo que son necesarios para ejecutarlo, a saber: tiempo de ejecución y espacio. Al analizar la complejidad de un algoritmo, el tiempo se expresa en término de los pasos de computación elementales (asignaciones, comparaciones, multiplicaciones, etc.) que le toma al algoritmo calcular el resultado a partir de los datos de entrada; por ejemplo, una operación de asignación ocupa una unidad de tiempo para ejecutarse, un ciclo ocupa el número de iteraciones en que está definido, etc. El espacio es un indicador de la cantidad de memoria que se necesita para ejecutar el algoritmo. Sin embargo, en la teoría de la complejidad de los algoritmos, la eficiencia está usualmente expresada en términos de la complejidad del tiempo. La *complejidad temporal* tiene que ver con el tiempo que tarda un algoritmo en ejecutarse. La *complejidad espacial* estudia la cantidad de espacio de almacenamiento que es necesario para una operación.

Es evidente que al medir el tiempo que tarda un algoritmo en ejecutarse, no es conveniente especificar que tarda cinco segundos en una máquina de 166 MHz y que tarda menos en una máquina más rápida. Para que sea relevante el definir la complejidad temporal de un algoritmo, ésta debe ser expresada no en términos de la máquina, sino en

cuanto al *tamaño de la entrada*. Al referirse la complejidad con respecto al tamaño de entrada, esta medida resulta ser más representativa y útil.

Cuando se define la complejidad en función del tamaño de entrada, no se está considerando a los datos que forman esa entrada, sino al conjunto en su tamaño. En la práctica se observa que un algoritmo ocupa diferente tiempo de ejecución para entradas del mismo tamaño, pero con diferentes datos. Por ejemplo, un método de ordenación de valores puede tardar menos tiempo si su entrada está ordenada, o tardar mucho mayor tiempo incluso con los mismos valores, pero presentados en desorden; es decir, el mismo tamaño de entrada tiene diferente comportamiento. Por lo tanto, se ha adoptado el criterio de tomar siempre como base de análisis de la complejidad de un algoritmo, el caso en el cual consuma el mayor tiempo de ejecución, es decir, *el peor caso* (ver [32] y [33]), el cual es el número máximo de operaciones básicas que el algoritmo espera ejecutar para una entrada de tamaño n.

Orden de un Algoritmo

La función que define el tiempo de ejecución de un algoritmo proporciona información interesante para clasificar los diferentes algoritmos que existen para resolver problemas. Con esta función es posible comparar el desempeño de diferentes algoritmos desarrollados para un problema en particular.

Para simplificar el estudio de la complejidad, se han adoptado ciertas convenciones en su notación, una de ellas es la del *concepto de orden* [32], el cual se expresa usando la notación $O(T(n))$, donde n simboliza el tamaño de entrada, la cual es definida como sigue:

Una función $T(n)$ se dice ser $O(T(n))$, si hay una constante k, tal que $|T(n)| \leq k \cdot |T(n)|$, $\forall n \geq 0$. En otras palabras, $O(T(n))$ se refiere a las funciones que no crecen más rápido que $T(n)$ y la notación $O(T(n))$ indica que la complejidad del tiempo del *peor caso* del algoritmo está acotada por $T(n)$. Esta notación indica, de forma simple, el grado de complejidad de un algoritmo sin considerar por completo la función de tiempo. Por ejemplo, tomando el método de la burbuja para ordenamiento de datos, éste tiene una función de tiempo $n(n\text{-}1)/2 = (n^2\text{-}n)/2$, y se dice que es de orden n^2, o simplemente $O(n^2)$.

Ejemplo D1: La función que define el tiempo de ejecución del siguiente algoritmo es $T(n) = 4n + 3$.

```
Paso 1:  suma = 0;
Paso 2:  for (i = 1; i ≤ n; i++)
Paso 3:          suma+ = i;
```

- El Paso 1 realiza una asignación y ocupa una unidad de tiempo T(1).
- El Paso 2 tiene involucrada una asignación que utiliza una unidad de tiempo, $n+1$ comparaciones y n incrementos, por lo que ocupa $2n+2$ unidades de tiempo.
- El Paso 3 ocupa dos unidades de tiempo, una para la suma y otra para la asignación, y es ejecutada n veces, por lo que ocupa $2n$ unidades de tiempo.

Por tanto, el tiempo total del algoritmo es $T(n) = 1 + 2n + 2 + 2n$, esto es: $T(n) = 4n + 3$.

Para determinar el orden un algoritmo, a partir de su función de tiempo, se eliminan todos los términos excepto el de mayor grado y se eliminan todos los coeficientes del término mayor. Esto debido a que al aumentar el tamaño de entrada, es más significativo el incremento en el término de mayor orden que el de los demás.

Ejemplo D2: El orden de un algoritmo cuyo tiempo de ejecución está dado por $T(n) = 3n^3 + 2n + 6$ es: $O(T(n)) = O(n^3)$.

Los algoritmos que tienen una complejidad del tiempo descrita por una función polinomial (por ejemplo, $O(4n)$, $O(n^3)$, etc.) son considerados eficientes porque pueden ser ejecutados en una cantidad de tiempo razonable para entradas de tamaño considerable. Sin embargo, si la complejidad del tiempo del algoritmo está descrita por una función exponencial (por ejemplo, $O(3^n)$, $O(n^{\log n})$, etc.) entonces el algoritmo es considerado ineficiente porque puede ser ejecutado en una cantidad de tiempo razonable solo para entradas de longitudes pequeñas, pero para entradas más grandes ejecutar el algoritmo llegar a no resultar práctico.

La diferencia entre algoritmos de tiempo polinomial y algoritmos de tiempo exponencial es la razón en la cual su complejidad de tiempo computacional crece dado un incremento en el tamaño de la entrada (n). Hay algunos algoritmos de tiempo polinomial que no son muy útiles en la práctica, ya que n es típicamente grande para instancias prácticas. Por el contrario, hay algoritmos de tiempo exponencial que se consideran útiles, ya que el tiempo de su ejecución en las instancias prácticas se considera relativamente bueno, debido a que los valores de n son pequeños.

Existen algunas reglas que son útiles para determinar el orden de un algoritmo:

- Regla 1 - *Ciclos FOR*: El tiempo de ejecución de un ciclo *FOR* es al menos el tiempo de ejecución de las instrucciones dentro de él multiplicado por el número de iteraciones.

Ejemplo D3: El orden del siguiente algoritmo es $O(n)$.

```
Paso 1:  for (i = 1; i ≤ n; i++)
Paso 2:      suma+ = i;
```

La función de tiempo para el algoritmo es $T(n) = 4n + 2$, ya que el ciclo ocupa $2n + 2$ y la suma $2n$, por lo tanto tiene un orden $O(n)$.

- Regla 2 - *Ciclos FOR anidados*: Se analiza desde dentro hacia afuera. El tiempo total de una instrucción dentro de un conjunto de ciclos anidados es igual al tiempo de ejecución de las instrucciones internas multiplicado por el producto del tamaño de los ciclos.

Ejemplo D4: El orden del siguiente algoritmo es $O(n^2)$.

```
Paso 1:  for (i = 1; i ≤ n; i++)
Paso 2:      for (j = 1; j ≤ n; j++)
Paso 3:          suma + = ij;
```

- Regla 3 - *Condicional*: El tiempo de ejecución nunca es mayor que el tiempo de ejecución de la condicional más el mayor de los tiempos de ejecución de las alternativas.

Ejemplo D5: El orden del siguiente algoritmo es $O(n^2)$:

```
Paso 1:  if (i = = j)
Paso 2:      for (i = 1; i ≤ n; i++)
Paso 3:          suma + = i;
Paso 4:  else
Paso 5:      for (i = 1; i ≤ n; i++)
Paso 6:          for (j = 1; j ≤ n; j++)
Paso 7:              suma + = ij;
```

Debido a que el tiempo de ejecución de la condición (Paso 1) es de orden $O(1)$ más el mayor tiempo de ejecución de las alternativas, es decir, cuando se cumple la condición es $O(n)$ y cuando no se cumple es $O(n^2)$, por lo tanto el orden es $O(1)+O(n^2)$. Como ya se ha dicho, solo se toma el mayor, quedando el orden del algoritmo en $O(n^2)$.

Si se gráfica el orden de un algoritmo para varios tamaños de entrada, se puede observar su comportamiento. Como se muestra en la Figura D1, un algoritmo de orden $O(n^2)$ (línea gruesa) tiene una velocidad de crecimiento menor que uno con orden $O(2^n)$, representado por la línea delgada. El eje vertical representa $T(n)$ y el eje horizontal a n.

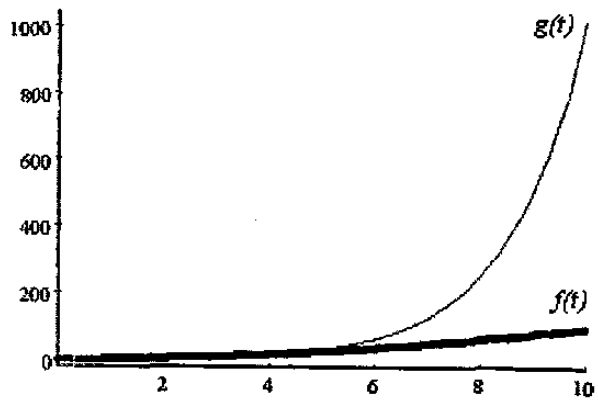

Figura D1: Comparación de complejidad: polinomial ($f(t)$) y exponencial ($g(t)$).

Se puede decir que un algoritmo tiene complejidad polinomial, o se ejecuta en *tiempo polinomial* si tiene un orden $O(n^x)$, donde x es un número fijo. Estos algoritmos se dice que son algoritmos eficientes y los problemas que se resuelven con estos algoritmos se dice que son problemas tratables.

Un algoritmo tiene complejidad exponencial si la función de tiempo $T(n)$ tiene un orden $O(x^n)$, donde x es un número fijo. Los problemas para los cuales no existe un algoritmo polinomial que los resuelva, es decir, aquellos que la única forma de resolverse es mediante algoritmos de *tiempo exponencial* se conocen como problemas intratables y a los algoritmos como algoritmos ineficientes.

¿Por qué es importante el tiempo de ejecución de un algoritmo?

Es importante porque si conocemos, o al menos tenemos una idea del tiempo de ejecución del algoritmo, podemos saber qué tanto va a tardar en entregarnos una respuesta, y por tanto, podríamos decidir si la esperamos y nos vamos a tomar un café, o si mejor regresamos después de una semana.

Aún cuando las computadoras de hoy en día son muy rápidas, comparadas con sus similares de hace algunos años, y son capaces de llevar a cabo millones de operaciones en un segundo, resulta fácil encontrar ejemplos de problemas y soluciones que tardarían años en solucionarse.

Cuando se analiza el tiempo de ejecución de un algoritmo, más que el tiempo exacto que tardará el algoritmo en milisegundos, lo que importa es la función de crecimiento del algoritmo. Esta función nos da una idea de qué tanto aumentará el tiempo de ejecución con respecto al incremento del tamaño de entrada.

Veamos como ejemplo el problema de ordenar una lista de números. Se tiene un conjunto $A = \{x_1, x_2, ..., x_n\}$, y un algoritmo de ordenación que debe entregar como salida una permutación del conjunto A, tal que $x_i \leq x_j$ para cualquier $i < j$. Es decir, debe ordenar de manera ascendente al conjunto A.

Ordenar una lista de números es quizá la operación más común en las computadoras, cientos de programas ordenan datos para poder trabajar con ellos. Debido a esto, la cantidad de algoritmos de ordenación que se encuentra en la literatura es extensa, así como el tiempo que se ha dedicado a estudiarlos. Observemos como ejemplo el comportamiento de dos algoritmos clásicos de ordenación: *Por inserción (Insertion Sort)* y *por mezcla (Merge Sort)* [26].

La ordenación *por inserción* tiene una función de crecimiento $T(n) = n^2$, mientras que la ordenación *por mezcla* tiene una función de crecimiento $T(n) = n \log n$ (donde *log n* representa el logaritmo base 2 de *n*). Para demostrar el por qué es necesario realizar un análisis de complejidad, comparemos estos dos algoritmos. Supongamos que el mejor programador del mundo implementó la ordenación *por inserción* en lenguaje máquina logrando una constante de 2, por lo que el tiempo total de corrida será $t = 2n^2$, donde *n* representa la cantidad de números que deseamos ordenar. Ahora supongamos que un programador promedio implementó la ordenación *por mezcla* utilizando un lenguaje de alto nivel obteniendo una constante de 50, por lo tanto, el tiempo total de ejecución será de $t = 50\ n \log n$. Supongamos además que tenemos una computadora capaz de ejecutar 10,000,000 de operaciones por segundo.

Comparemos ahora el desempeño de ambas soluciones, supongamos primero que queremos ordenar 1000 números.

inserción (t) = $2\ (1000)^2 = 2{,}000{,}000 = 0.2$ seg.

mezcla (t) = $50\ (1000)\ (11) = 550{,}000 = 0.055$ seg.

Como podemos observar, el desempeño de la ordenación por mezcla es más eficiente. La diferencia se va haciendo más radical conforme el tamaño de la entrada va incrementándose; por ejemplo, si quisiéramos ordenar 100,000,000 de números, los tiempos serían:

inserción (t) = $2\ (100000000)^2 = 20{,}000{,}000{,}000{,}000{,}000$ (*Un poco más de 63 años!*)

mezcla (t) = $50\ (100000000)\ (27) = 135{,}000{,}000{,}000$ (*3 horas con 45 minutos*)

Mientras que la ordenación *por inserción* tomaría todo el resto de sus vidas, la ordenación *por mezcla* apenas y necesitaría menos de 4 horas. De ahí que, realmente vale la pena realizar un análisis de complejidad de un algoritmo antes de implementarlo.

El estudio de complejidad está relacionado con el tipo de cómputo donde se aplica el algoritmo. Actualmente, los modelos de cómputo se pueden agrupar en dos grandes conjuntos (ver [32]): determinísticos y no determinísticos.

El modelo determinístico está asociado históricamente a la máquina de Turing (ver [33]), ya que esta máquina es un modelo matemático estándar que formaliza la noción de computadora.

Una máquina de Turing determinista está formada por un control basado en estados finitos, una cinta infinita en ambos sentidos dividida en celdas, y una cabeza de lectura y escritura. El determinismo estriba en que para una misma entrada, el resultado de aplicar repetidas veces un algoritmo, el resultado siempre es el mismo. El modelo no determinístico es equivalente a decir que pueden obtenerse resultados diversos para una misma entrada. Este modelo contiene dos fases: una fase de adivinación y una de verificación. La fase de adivinación (no determinista) escoge al azar una estructura que puede ser la solución, y la fase de verificación (determinista) consiste en determinar si la estructura escogida es una solución.

Clases P y NP

La complejidad computacional de un problema es determinado por el mejor algoritmo que puede encontrarse para resolver el problema (ver [13]). En un nivel de abstracción alto, si puede encontrarse un algoritmo de tiempo polinomial para un problema dado, entonces el problema es considerado tratable o no tan duro. Pero si tal algoritmo no puede encontrarse, es decir, solo algoritmos en tiempo exponencial pueden ser construidos, el problema se considera intratable o muy duro, aún si el problema es soluble. La teoría de la complejidad computacional ha sido desarrollada considerando principalmente a los problemas de decisión. Muchos de los problemas de optimización pueden ser expresados como un problema de decisión. Un problema de decisión es un problema para el cual la respuesta es "*sí*" o "*no*" de acuerdo a si las entradas satisfacen o no las condiciones dadas en el problema.

Algunos ejemplos de problemas de decisión son dados a continuación.

- *Par.* Dado un número natural n, ¿Es n un número par? La respuesta es "sí", si n es par, o "no", si n es impar.
- *Primo.* Dado un número natural n, ¿Es n un número primo? La respuesta es "sí", si n es primo, o "no", si n es compuesto.
- *Satisfactibilidad.* Dada una expresión *booleana* $f(x_1, x_2, ..., x_n)$, ¿pueden las variables $x_1, x_2, ..., x_n$ ser fijadas a valores que hagan el valor de f *"verdadero"*? La respuesta es "sí", si hay una configuración de las variables que hagan a f *"verdadero"*, y "no" de otro modo.
- *Ciclo Hamiltoniano.* Dado un grafo $G(V, E)$ con N nodos, ¿hay un ciclo de aristas en G que incluya cada uno de los N nodos? La respuesta es "sí", si tal ciclo existe, y "no" de otro modo.
- *Asignación de Espacio.* Dada n entidades y m cuartos disponibles, ¿es posible construir una asignación de las n entidades a los m cuartos, de tal manera que todas las restricciones existentes (duras y suaves) sean satisfechas y el espacio desperdiciado sea a lo mucho W? La respuesta es "sí", si tal asignación existe, y "no" de otro modo.

Hay dos clases en las cuales los problemas son clasificados (ver [13] y [37]): clases **P** y **NP**. La clase **P** está constituida por todos aquellos problemas para los cuales un algoritmo determinístico en tiempo polinomial ha sido encontrado. La clase **NP** (por sus siglas en inglés, *Non-deterministic Polynomial*) incluye a todos aquellos problemas para los cuales un algoritmo de tiempo polinomial no determinístico es conocido para resolver el problema. Un algoritmo no determinístico puede ser descrito como consistencia de dos fases. La primera fase supone una estructura para el problema y la segunda fase verifica si la estructura dada es o no una solución del problema. Entonces, el algoritmo se dice ser un algoritmo de tiempo polinomial no determinístico si para cada instancia del problema existe la conjetura de que puede ser verificado por una fase determinística para una respuesta "*sí*" en tiempo polinomial.

Entonces, si la clase **P** son los problemas resueltos en tiempo polinomial por un algoritmo determinístico y la clase **NP** son los problemas resueltos en tiempo polinomial por un algoritmo no determinístico, la pregunta es si $\mathbf{P} = \mathbf{NP}$ o $\mathbf{P} \neq \mathbf{NP}$. De hecho, esta es la pregunta abierta más importante en la teoría de la complejidad computacional. Es claro que $\mathbf{P} \subseteq \mathbf{NP}$, lo cual significa que los algoritmos no determinísticos son más poderosos que los algoritmos determinísticos. Si hay un algoritmo determinístico para

un problema, uno no determinístico puede ser construido simplemente al no usar una fase suposición.

Hay muchos problemas conocidos con certeza que pertenecen a la clase **NP**, para los cuales un algoritmo eficiente no ha sido encontrado, y estos problemas son considerados ***NP-duros*** en sentido fuerte, es decir, estos son intratables. Esto confirma la creencia de que $\mathbf{P} \neq \mathbf{NP}$, pero ésta es simplemente una conjetura que aún no ha sido demostrada.

Si es verdad que $\mathbf{P} \neq \mathbf{NP}$, entonces los problemas en el conjunto $\mathbf{NP} \setminus \mathbf{P}$ son intratables. Por tanto, cuando un problema en particular es abordado, es importante saber si el problema pertenece a la clase de los problemas tratables o intratables. Una manera de hacer esto es determinar si el problema de interés, está o no relacionado a otro problema que ha sido anteriormente demostrado ser tratable o intratable. Reduciendo un problema a otro, es la técnica usada para demostrar si los dos problemas se relacionan o no. La reducción es provista por una transformación que permite el mapeo de una instancia del primer problema dentro de una instancia del segundo problema. Esta transformación permite convertir un algoritmo que resuelve uno de los problemas a un algoritmo que resuelva al otro problema.

Transformación Polinomial

Una transformación polinomial es una función que permite cambiar la representación de problema D_1 a otro problema D_2 aplicando un algoritmo determinista de tiempo polinomial. Lo anterior se puede representar como "D_1 *se transforma a* D_2" o $D_1 \propto D_2$. Cualquier elemento del problema D_1 tiene un elemento equivalente en D_2.

Las transformaciones polinomiales son importantes porque sirven para determinar la pertenencia de los problemas a las clases **P** y **NP**, y permiten definir la clase ***NP-completo***.

Si D_1 se transforma a D_2 y D_2 pertenece a la clase P, entonces D_1 también pertenece a la clase **P**, porque si para cambiar de D_1 a D_2 se utiliza un algoritmo de tiempo polinomial y si D_2 (conocido con certeza dentro de la clase **P**) tiene un algoritmo de solución que se ejecuta en tiempo polinomial, D_1 debe tener asociado un algoritmo que lo resuelva también en tiempo polinomial, y D_1 debe pertenecer a la clase **P**. Se puede decir en general, que un problema D_1 está en **P** si cualquier problema que se sabe está en **P**, se puede transformar a D_1. También se puede decir que un problema D_1 está en **NP**, si cualquier problema que se sabe pertenece a la clase **NP**, se puede transformar a D_1.

Un problema D_1 es ***NP-completo*** si pertenece a la clase **NP**, y otro problema D_2 que también pertenece a **NP**, se puede transformar a D_1. Los problemas ***NP-completos*** son los problemas más difíciles en la clase **NP**. La palabra *completo* significa que la solución de un problema de decisión **NP**, contiene de alguna forma, la solución a todos los problemas de decisión de la clase **NP**.

Si D_1 se transforma a D_2 y D_2 es un problema ***NP-completo***, entonces D_1 también es un problema ***NP-completo***. Esto es importante, ya que establece que se puede probar que un problema de decisión D_1 es ***NP-completo***, si algún problema que se sabe es ***NP-completo*** puede transformarse al primero.

Problemas NP-duros

En ocasiones puede demostrarse que los problemas en **NP** pueden transformarse a algún problema A ($D_{NP} \propto A$), pero no puede decirse que A sea **NP** o ***NP-completo***. Sin embargo, es indudable que A es tan difícil (duro) como cualquier problema en **NP**, y sea entonces intratable. Para este tipo de problemas se asocia el término de ***NP-duros***.

Dentro de los problemas considerados ***NP-completos***, el primer trabajo reportado en la literatura fue el problema de Satisfactibilidad (SAT) demostrado por Cook en 1971 (ver [7]).

A saber, el problema de Satisfactibilidad es un problema central en la lógica matemática y la teoría de la computación. SAT es fundamental para la solución de varios problemas en razonamiento automático, diseño y manufactura asistida por computadoras, planificación, visión computacional, bases de datos, robótica, diseño de circuitos integrados, de arquitectura de computadoras y de redes de computadoras, entre otros

La satisfactibilidad proposicional es el problema de decidir si existe una asignación de valores de verdad a los átomos de una fórmula proposicional que la hacen verdadera.

Entre otros resultados, Cook probó que cualquier problema en **NP** puede ser reducido al problema de Satisfactibilidad. Esto significa que si hay un algoritmo eficiente para resolver el problema de Satisfactibilidad, entonces cualquier problema en **NP** puede también ser resuelto por un algoritmo eficiente. Estos problemas se dicen ser ***NP-completos*** y son considerados los más duros dentro de la clase NP. Esto es, porque si un problema ***NP-completo*** no tiene un algoritmo eficiente para resolverlo, entonces ninguno de ellos tiene un algoritmo eficiente y son todos intratables. Muchos problemas han sido demostrados ser ***NP-completos*** (o reducidos al problema de Satisfactibilidad),

pero esto aún no prueba que estos problemas sean intratables. Sin embargo, generalmente se asume que encontrar un algoritmo eficiente para cualquier problema dentro de la clase ***NP-completo*** es improbable.

Por tanto, si un problema es ***NP-completo*** y **P** ≠ **NP**, entonces el problema pertenece al conjunto **NP** \ **P**. En otras palabras, el problema (y todos en ***NP-completo***) pueden pertenecer a **P**, sí y solo sí **P** = **NP**. Así, si se asume que los problemas ***NP-completos*** son intratables, es decir, **P** ≠ **NP**, y un problema ha sido demostrado ser ***NP-completo***, entonces el enfoque no debe estar en encontrar un algoritmo eficiente, sino en diseñar algoritmos que produzcan soluciones de alta calidad, sin garantizar la optimalidad; es decir, diseñar algoritmos útiles que aborden al problema en la práctica.

Conclusiones

Es muy importante que siempre que se piense en un algoritmo, se realice un análisis de complejidad del algoritmo, aunque no todos los algoritmos son tan sencillos de analizar, para la mayoría es fácil determinar la función de crecimiento del mismo.

Determinar la complejidad del algoritmo puede dar una idea muy clara de cuánto tiempo va a tardar el mismo, y en base a los tamaños de entrada poder determinar si el algoritmo se ejecutará en un tiempo práctico o no.

D.2 Análisis de Complejidad de los Procedimientos Desarrollados

La metodología de solución del PMCC puede ser descrita por la secuencia de los algoritmos presentada por la Figura D2.

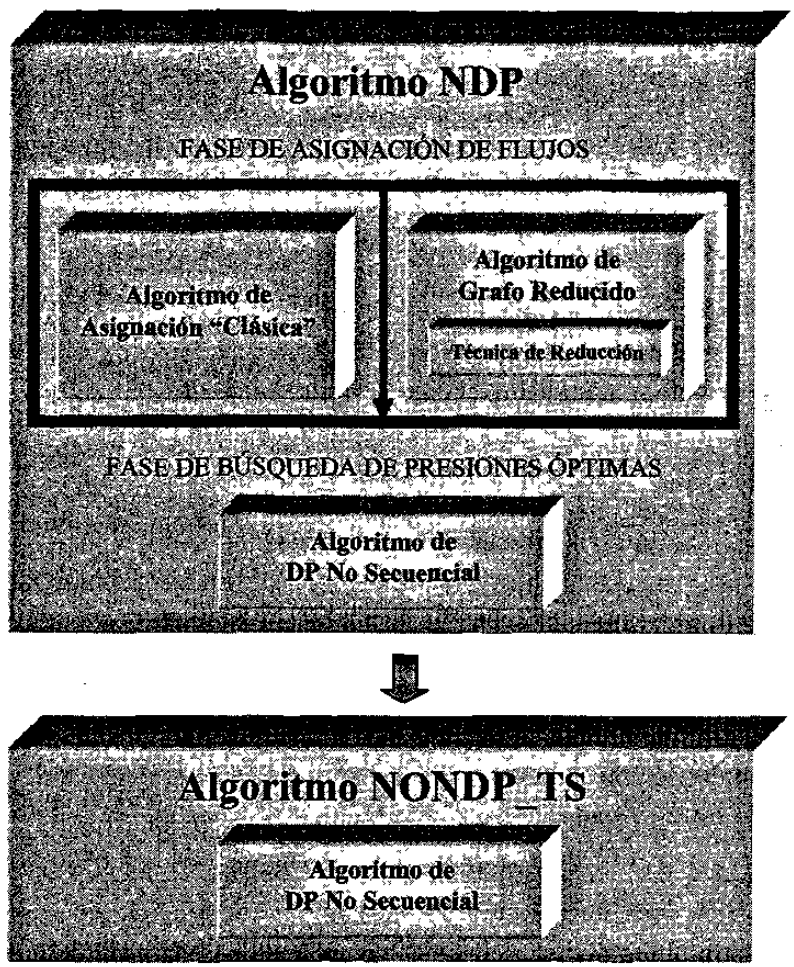

Figura D2: Secuencia de los algoritmos que conforman la metodología de solución.

Revisemos ahora el costo computacional de los algoritmos partiendo desde el centro hacia fuera de la secuencia presentada por la Figura D2.

Técnica de Reducción

Complejidad Computacional: Para referencia, véase Sección 4.1.3 y Fig. 4.1. El Paso 1 del algoritmo toma $O(|A_c|)$ operaciones. Para los Pasos 2 y 3, hay que visitar todos los arcos y nodos restantes por lo que toma un tiempo $O(|V|+|A_p|)$. El Paso 4, al igual que el Paso 1, toma un tiempo $O(|A_c|)$. Por consiguiente, asumiendo que |V| esta acotado por |A|, el procedimiento es $O(|A|)$.

Algoritmo de Grafo Reducido

Complejidad Computacional: Para referencia, véase pág. 42 y Fig. 4.10. El Paso 1 toma $O(|A|)$ operaciones como ya se vio en la Sección 4.1.3. En el resto del algoritmo, nótese que cada arco se visita una sola vez. En esta visita, se hacen cálculos y asignaciones de $O(1)$ (Pasos 3-5), y una vez asignado el flujo al arco, este se elimina y no vuelve a tomarse en cuenta. Por tanto, el algoritmo es $O(|A|)$.

Algoritmo de Asignación Clásica

Complejidad Computacional: Para referencia, véase pág. 38 y Fig. 4.5. El algoritmo consiste básicamente en encontrar una ruta de un nodo oferta a un nodo demanda y enviar el flujo requerido. Como se asume que siempre hay un flujo factible, y en el peor de los casos puede enviarse una unidad de flujo en cada ocasión (asumiendo flujos enteros), el algoritmo tiene complejidad $O(|B|)$, donde $B = \sum_{i \in V_d} B_i$. Sin embargo, en la práctica nunca se ha observado tal comportamiento, ya que casi siempre es posible enviar la cantidad completa disponible de B_k (en el Paso 2) en una sola iteración.

Algoritmo de DP No Secuencial

Complejidad Computacional: Para referencia, véase Sección 4.2.2 y Fig. 4.14. Asumiendo una rango de presiones ΔP, el número máximo de puntos está acotado por $pmax/\Delta P = N_p$, donde *pmax* es el máximo valor de las cotas superiores de las presiones.

Así las cosas, cada "optimización" (en el Paso 2) toma en el peor de los casos $O(N_p^2)$ operaciones. Para $|A_c|$ compresores, esto se repite $|A_c|$-1 veces, por tanto la complejidad es $O(|A_c| \cdot N_p^2)$. Esto implica que el procedimiento global NDP (Fig. 4.2) es $O(|A| \cdot N_p^2)$.

Algoritmo NONDP_TS

Complejidad Computacional: Para referencia, véase Sección 5.3.3 y Fig. 5.2. Como puede verse, el tiempo de cómputo está dominado por el trabajo de generación de los vecinos y cálculo de su correspondiente valor de la función objetivo. El número de vecinos está acotado por *Nei_Size* (parámetro definido por el usuario) y para calcular su valor de la función objetivo se aplica el procedimiento NDP, que toma, como ya se vio, $O(|A| \cdot N_p^2)$ operaciones. Si se implementa inteligentemente, no es necesario recalcular el valor del objetivo para una misma solución más de una vez. Como el ciclo del algoritmo está delimitado por *Iter_max*, el procedimiento NONDP_TS tiene una complejidad de $O(Iter\_max \cdot Nei\_Size \cdot |A| \cdot N_p^2)$.

Ficha Autobiográfica

Conrado Borraz Sánchez

Candidato para el Grado de Maestro en Ciencias en Ingeniería de Sistemas

Universidad Autónoma de Nuevo León

Facultad de Ingeniería Mecánica y Eléctrica

Tesis

Una Metodología de Solución basada en Programación Dinámica No Secuencial y Búsqueda Tabú para la Operación Eficiente de Sistemas de Transporte de Gas Natural en Estado Estable

Nacido en Tuxtla Gutiérrez, Chiapas. Hijo del Sr. Conrado de Jesús Borraz León y Sra. Magnolia Sánchez de Borraz, segundo de cuatro hermanos. Graduado en el Instituto Tecnológico de Tuxtla Gutiérrez (1997-2001) como Ingeniero en Sistemas Computacionales, y en el Centro de Estudios de Computación del Sureste (1993-1995) como Técnico Programador y Analista de Sistemas. Inició sus estudios de Maestría en Ciencias en Ingeniería de Sistemas en Febrero del 2002, después de haber realizado un verano de investigación científico en la Universidad Autónoma de Nuevo León. Logrando así, una beca de manutención por el proyecto J33187-A de CONACYT; desempeñándose entonces como Asistente de Investigación durante sus estudios de Maestría.

Pensemos que:

"Sólo grandes pensamientos permiten resolver grandes problemas,
así, hay en la solución de todo problema, un poco de magnificencia"

...pero entonces, si se resuelve un problema y llega a excitar nuestra curiosidad,

"este género de experiencia, a una determinada edad, puede determinar el gusto del
trabajo intelectual y dejar, tanto en el espíritu como en el carácter
...una huella que durará toda una vida".