01168

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE INGENIERIA
DIVISION DE ESTUDIOS DE POSGRADO

# GRASP EN PARALELO PARA EL PROBLEMA DE ASIGNACION CUADRATICA

## TESIS

QUE PARA OBTENER EL GRADO DE:
**MAESTRO EN INGENIERIA**
**(Investigación de Operaciones)**

P R E S E N T A :

ROGELIO GONZALEZ VELAZQUEZ

DIRECTOR DE TESIS:
DR. LUIS B. MORALES MENDOZA

MEXICO, D. F. 2000

**UNAM – Dirección General de Bibliotecas**
**Tesis Digitales**
**Restricciones de uso**

**DERECHOS RESERVADOS ©**
**PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL**

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

# Dedicatoria

- A mi esposa **Hortencia** y a mis hijos **Rogelio René** y **Karla Denisse**, porque los amo y por el tiempo que les robe.
- A mis padres **Juan González Barrón** y **Reyna Velázquez Rangel**, por todo el amor, por su apoyo y porque me brindaron la oportunidad que ellos no tuvieron .
- A mis hermanos **Juan, Aurelio, Guadalupe, Jesús, Alicia, Elias, Adela** y **Rocio** que siempre los llevo conmigo.

*Rogelio González Velázquez*
*México, D. F., Octubre 2000*

**Agradecimientos:**

- Al **Dr. Luis B. Morales Mendoza** por aceptar ser mi director de tesis, por su ayuda, comentarios y recomendaciones para la culminación de este proyecto.
- A mis profesores **Dra. Idalia Flores de la Mota** y **Dr. Ricardo Aceves García** por su apoyo, por haber revisado este documento y aceptar estar presentes en mi jurado.
- A la **Dra. Ana Elena Narro Ramírez** y al **Dr. Sergio G. de los Cobos Silva** por ceder parte de su tiempo, revisar esta tesis y aceptar ser mis sinodales.
- A mi amigo **Oscar Viveros Nava** por su apoyo académico y porque siempre estuvo ahí.
- A mis amigos **Mat. Martín Estrada Analco** por las sugerencias que me ayudaron a depurar este trabajo y **M. C. Carlos Guillén Galván** y **Dr. Guillermo De Ita Luna** por las facilidades para la elaboración de este trabajo.
- A la **M. C. Pilar Álvarez Rivas** por haber revisado este manuscrito de tesis.
- A mi amigo **Fis. Primo País Flores** compañero de estudios de maestría.
- A mi amigo **Dr. José Luis Martínez Flores** quien siempre me alentó a obtener el grado.
- A la **Benemérita Universidad Autónoma de Puebla** por su apoyo económico
- A la **DEPFI** de la **Universidad Nacional Autónoma de México** por la formación que me brindo.
- Al **Consejo Nacional de Ciencia y Tecnología** por la beca que me concedió.
- A la **Dirección General de Servicios de Computo Académico** DGSCA de la UNAM por permitirme el uso de la computadora SGI/CRAY ORGIN 2000.

Me siento honrado por haber realizado la tesis: **GRASP en Paralelo para el Problema de Asignación Cuadrática**, bajo la dirección del Dr. Luis B. Morales Mendoza, por su ejemplo de profesionalismo, pero sobre todo por su calidad humana, mi admiración y respeto para el.

*Rogelio González Velázquez*
*México, D. F.,Octubre 2000*

# Contenido


**Introducción** .......... 2
1. **Conceptos Básicos** .......... 4
   1.1 Introducción .......... 4
   1.2 Óptimos locales y globales .......... 6
   1.3 Complejidad de los problemas .......... 7
   1.4 Técnicas heurísticas .......... 9
   1.5 Tipos de heurísticas .......... 10
   1.6 La metaheurística GRASP .......... 11
2. **Problema de Asignación Cuadrática** .......... 12
   2.1 Introducción .......... 12
   2.2 Antecedentes del *QAP* .......... 12
   2.3 Modelos del problema de asignación cuadrática .......... 13
      2.3.1 Modelo de optimización combinatoria para el *QAP* .......... 13
      2.3.2 El modelo de programación entera para el *QAP* .......... 14
   2.4 Aplicaciones del *QAP* .......... 16
   2.5 Complejidad del *QAP* .......... 16
3. **GRASP** .......... 18
   3.1 Introducción .......... 18
   3.2 Metaheurística GRASP .......... 18
      3.2.1 Fase de construcción y sus componentes .......... 19
      3.2.2 Fase de postprocesamiento .......... 21
   3.3 Aplicaciones de GRASP .......... 21
   3.4 Un GRASP para el *QAP* .......... 22
4. **Paralelismo** .......... 29
   4.1 Introducción .......... 29
   4.2 ¿Qué es la computación paralela? .......... 29
   4.3 Terminología .......... 30
   4.4 Modelos de computación paralela .......... 32
   4.5 Modelos de organización de memoria .......... 34
   4.6 Métodos de programación .......... 36
   4.7 Herramientas para memoria distribuida .......... 38
   4.8 MPI (Message Passing Interface) .......... 38
5. **Programación del GRASP para el *QAP*** .......... 40
   5.1 Introducción .......... 40
   5.2 Descripción general del programa .......... 40
6. **Resultados** .......... 46
   6.1 Introducción .......... 46
   6.2 Tablas de resultados .......... 46
   6.3 Comparación de los resultados .......... 48
      6.3.1 Graficas de comparación .......... 48
      6.3.2 Tablas de comparación .......... 50
   6.4 Permutaciones de mejor asignación .......... 52
   6.5 Factor de aceleración .......... 52
7. **Conclusiones** .......... 54

A. **Listado del programa** .......... 55

B. **Matrices de datos** .......... 69

**Bibliografía** .......... 77

# Introducción

Existe hoy en día, una gran variedad de herramientas que permite incrementar la probabilidad de tomar mejores decisiones en cualquier organización y muchas de estas situaciones pueden ser planteadas como problemas de optimización. Cabe mencionar que entre esta gama de herramientas se encuentran las metaheurísticas cuyo avance en la solución de problemas de alta complejidad ha sido impresionante. En los últimos años, la necesidad de dar respuesta a la administración eficiente de recursos escasos para incrementar la productividad a lo cual es preciso ofrecer algún tipo de solución ha provocado un auge excepcional de este tipo de técnicas aunado al incremento de la eficiencia computacional.

Esta obra trata sobre una de estas herramientas, específicamente la metaheurística Procedimiento de Búsqueda Voraz, Aleatorio, y Adaptativo, cuyas siglas en inglés son GRASP, la cual será aplicada a un problema de optimización combinatoria conocido como problema de asignación cuadrática (*QAP, quadratic assignment problem*) el cual pertenece a una clase de problemas para los cuales no se conoce un algoritmo que pueda resolverlo en tiempo polinomial. El *QAP* es objeto de estudio por parte de muchos investigadores debido a la adaptabilidad de su modelo matemático para resolver diversos problemas de aplicación practica , por ejemplo el problema del agente viajero.

Los principales objetivos de este trabajo son:

1. Plantear el problema de asignación cuadrática como un problema de optimización combinatoria.

2. Describir un GRASP para el *QAP*.

3. Aplicar la técnica GRASP para obtener soluciones del *QAP* y reportar la experiencia computacional obtenida.

Se proporciona una aplicación concreta en la que se muestra la importancia de metaheurística GRASP en los problemas de optimización combinatoria. Como aportaciones se pueden citar las siguientes:

1. Diseño para un GRASP de un programa en *lenguaje C* para el *QAP*.

2. Desarrollo de un programa para cada una de tres estructuras vecinales de búsqueda local.

3. Diseño de un programa en paralelo para reducir el tiempo de uso de CPU.

4. Se reporta la experiencia computacional sobre la eficiencia de nuestro trabajo de tesis.

En el capítulo1 se da la descripción de los problemas de optimización combinatoria, de las técnicas heurísticas y los problemas NP-completos. En el capítulo 2 se describe el problema de asignación cuadrática y se plantea como un problema de optimización combinatoria. En el capítulo 3 se introduce la técnica GRASP, se describen sus principales componentes y se aplica al problema de asignación cuadrática. En el capítulo 4 se dá una introducción a la computación paralela. En el capítulo 5 se describe cómo se desarrollan los programas secuencial y paralelo para GRASP. En el capítulo 6 se muestran las tablas de resultados obtenidos al aplicar tres programas a una serie de instancias del *QAP*, las gráficas y tablas de comparación y las permutaciones de mejor asignación. En el capítulo 7 se presentan las conclusiones y las posibles líneas de investigación. En el apéndice A se presenta el listado completo del programa para la estructura vecinal 2-intercambio. Finalmente en el apéndice B se muestran las matrices compactas de datos de las instancias propuestas a resolver en el presente trabajo.

# Capítulo1

## Conceptos Básicos

### 1.1 Introducción

Muchos investigadores han estudiado el problema de buscar soluciones óptimas a problemas que pueden ser estructurados como una función de algunas variables de decisión, y tal vez con la presencia de algunas restricciones. Tales problemas pueden ser formulados de la siguiente manera:

$$\text{Optimizar } f(x)$$

Sujeta a

$$g_i(x) \geq b_i \quad ; i = 1,\ldots,m$$
$$g_i(x) \leq c_i \quad ; i = m+1,\ldots,n$$

donde x es un vector de variables de decisión en $\boldsymbol{R}^n$ y $f(x)$, $g_i(x)$ son funciones generales.

Se dice que un punto $x_0$ es un mínimo global de $f$ si para todo vector de decisión $x$, $f(x_0) \leq f(x)$. El punto $x_0$ es llamado mínimo local si existe un subconjunto propio S de $\boldsymbol{R}^n$ tal que para todo $x$ en S $f(x_0) \leq f(x)$.

Existen muchas clases específicas de tales problemas, Los cuales se obtienen aplicando restricciones sobre el tipo de funciones consideradas y en los valores que las variables de decisión pueden tomar. Quizás los problemas más conocidos son aquellos que se obtienen restringiendo a que $f(x)$, $g_i(x)$ sean funciones lineales de las variables de decisión, Las cuales pueden tomar valores continuos, lo que conduce a problemas de programación lineal.

Consideraremos aquí otra clase de problemas: aquellos de naturaleza combinatoria.

Una permutación $P$ es un arreglo de enteros positivos donde cada una de sus componentes $p(i)$ representa el valor de la componente de $P$ en la posición $i$, de aquí en adelante esa será nuestra interpretación.

Diremos que un problema es de ***optimización combinatoria***, si en ellos las variables de decisión sólo admiten valores enteros y su espacio de soluciones está formado por permutaciones o subconjuntos de números naturales.

Los problemas de optimización combinatoria tienen nexos cercanos a la programación lineal y muchos intentos para resolverlos utilizan estos métodos, donde la pertenencia o no de una variable al subconjunto buscado se representa a través de su función característica para producir una formulación de programación entera. Tales formulaciones en ocasiones involucran un gran número de variables y restricciones, y aunque muchos de estos métodos han probado su convergencia teórica no pueden hacer frente a problemas muy grandes, que gneralmente son los que surgen en la practica de la vida real. Uno de los principales problemas de la programación entera es que la región de factibilidad es no convexa, por lo cual no hay garantía de que un problema de tipo entero pueda ser resuelto en un tiempo razonable para una computadora Prawda [21].

Un ejemplo de optimización combinatoria es el **problema de asignación,** un conjunto de n personas está disponible para realizar n tareas. Si la $i$-ésima persona realiza la $j$-ésima tarea, esto representa un costo $c_{ij}$ unidades, entonces el problema consiste en encontrar una asignación la cual minimice el costo total. Un planteamiento de tipo combinatorio es:

Sea $\aleph = \{1,..., n\}$ y $C = (c_{ij})$, la matriz de costos. El problema de asignación es:

$$\min_{p \in \Pi_{\aleph}} \sum_{i=1}^{n} c_{i\,p(i)}$$

donde $\Pi_{\aleph}$ es el conjunto de todas las permutaciones en $\aleph$.

Es común usar el término instancia para distinguir entre una situación particular de una general.

Por ejemplo, una situación que describe una instancia particular es, una empresa tiene cuatro personas $P_i$, $i = 1,2,3,4$ que debe asignar a cuatro tareas $T_j$, $j = 1,2,3,4$. Los costos estimados de asignación de cada trabajador se muestran en la tabla 1.1.1.

| | $T_1$ | $T_2$ | $T_3$ | $T_4$ |
|---|---|---|---|---|
| $P_1$ | 15 | 19 | 20 | 18 |
| $P_2$ | 14 | 15 | 17 | 14 |
| $P_3$ | 11 | 15 | 15 | 14 |
| $P_4$ | 21 | 24 | 26 | 24 |

Tabla 1.1.1.

Una solución óptima para esta instancia del problema de asignación es la permutación (1, 4, 3, 2 ), la cual indica que la tarea 1 ha sido asignada al trabajador 1, la tarea 4 a la persona 2,la tarea 3 a la persona 3 y la tarea 2 a la persona 4; y el costo es $c_{11} + c_{24} + c_{33} + c_{42} = 68$.

Quizas los dos problemas combinatorios más famosos sean el problema de la mochila (*knapsack problem*) y el problema del agente viajero (*TSP* siglas de *Travelling Salesman Problem* en inglés).

El **problema de la mochila** Consiste en seleccionar de entre n productos, cada uno con valor $c_i$ y volumen $V_i$, aquellos que quepan en un recipiente con volumen $V$, y que tengan el mayor valor posible, es decir se trata de determinar un subconjunto $I^* \subseteq \aleph$ para el cual:

$$\sum_{I^*} c_i = \max_{I \subseteq \aleph} \sum_{I} c_i$$

*con la restricción*

$$\sum_{I^*} V_i \leq V$$

El **problema del Agente Viajero** Consiste en que un vendedor tiene que buscar una ruta para visitar $n$ ciudades una y solo una vez, de modo que minimice el total de la distancia recorrida. El agente inicia el viaje en cualquier ciudad y lo finaliza cuando ha visitado las $n$ ciudades una y solo una vez, y ha regresado al punto de partida; es decir

Si $\aleph = \{1,2..., n\}$ y $D = ( d_{ij} )$ es la matriz de distancias de la ciudad $i$ a la ciudad $j$, entonces se debe encontrar una permutación $p \in \Pi_\aleph$ de tal manera que se minimice

$$\sum_{i=1}^{n} d_{ip(i)}$$

Por ejemplo, una instancia para el *TSP* es la matriz de distancias para seis ciudades; cuya distancia de ida y vuelta es la misma se presenta en la tabla 1.1.2.

| | | | | | |
|---|---|---|---|---|---|
| 0 | 16 | 16 | 33 | 12 | 15 |
| 16 | 0 | 16 | 10 | 9 | 29 |
| 16 | 16 | 0 | 8 | 7 | 7 |
| 33 | 10 | 8 | 0 | 21 | 20 |
| 12 | 9 | 7 | 21 | 0 | 20 |
| 15 | 29 | 7 | 20 | 20 | 0 |

Tabla 1.1.2

una solución óptima para esta instancia es la ruta (1,6,3,4,2,5) con un recorrido de 61 Km.

## 1.2 Óptimos locales y globales

Una característica de la mayoría de los problemas de optimización combinatoria es que pueden tener muchos óptimos globales, aunque tienen muchos más que son óptimos locales.

El método más simple para resolver una instancia de un problema de optimización combinatoria es listar todas las soluciones factibles de la instancia dada, evaluar su función objetivo y escoger la mejor solución.

Se puede pensar que la solución de un problema de optimización combinatoria únicamente se limita a buscar de manera exhaustiva el valor que minimice o maximice la función objetivo dentro de un conjunto finito de posibilidades y que utilizando una computadora muy rápida, el problema carecería de interés matemático sin detenernos a pensar por un momento en el tamaño de este conjunto finito de posibilidades.

Sin embargo, aunque este método nos lleva a una solución óptima buscada es obvio que este método de enumeración completa es muy ineficiente conforme crece el tamaño de la instancia del problema debido a la *explosión combinatoria* del espacio de soluciones, es decir, dado un conjunto de elementos se pueden obtener diferentes arreglos ordenados de éstos permitiendo una gran cantidad de posibilidades para cualquier entrada de tamaño no muy reducido (por ejemplo el de orden $n!$ ó $2^n$ ).

Para ilustrar esto, consideraremos los dos problemas más famosos de optimización combinatoria, primero el Problema del Agente Viajero cuyo espacio de soluciones es de tamaño $(n\text{-}1)!$ ó $(n\text{-}1)!/2$ si la distancia entre cada par de ciudades es la misma sin considerar la dirección del viaje. Pensando en una instancia con 21 ciudades tendríamos un espacio de soluciones factibles de 2432902023163674620 posibles soluciones.

Supongamos que una computadora puede listar todos los elementos del espacio de soluciones de una instancia. La tabla 1.2.1 nos muestra el crecimiento del tiempo de cómputo con respecto del tamaño de la entrada del problema Reeves [23] ; así la enumeración completa es ineficiente para obtener una solución óptima.

| Número de Ciudades | Tiempo de Cómputo |
|---|---|
| 20 | 1 hora |
| 21 | 20 horas |
| 22 | 17.5 días |
| 23 | 1.05 años |
| 24 | 24.6 años |
| 25 | 5.82 siglos |

Tabla 1.2.1: Crecimiento del tiempo de cómputo

Por otro lado consideremos el problema de la mochila. El número de subconjuntos del conjunto $\{1,2,..., n\}$ es $2^n$. Nuevamente si una computadora pudiera en un segundo generar un millón de esos subconjuntos y evaluar su valor sería necesario solamente un segundo para hallar la solución de un problema con $n = 20$ entradas ya que $2^{20} = 1{,}048{,}580$; para $n = 40$ entradas se necesitarían unas dos semanas, pero para $n = 60$ se tendrían $2^{60}$ soluciones lo cual llevaría un tiempo aproximado de 365 siglos. Por lo anterior, queda patente la necesidad de caracterizar los problemas en general.

## 1.3 Complejidad de los problemas

Los problemas se pueden clasificar en cuatro clases de acuerdo a su grado de dificultad como:

a) **Problemas indecibles**. Son aquellos problemas para los cuales no se puede escribir un algoritmo.
b) **Problemas intratables** (problemas que se demuestran que son difíciles). Son aquellos problemas para los cuales no se pueden desarrollar algoritmos polinomiales. En otras palabras, solo se pueden resolver con algoritmos exponenciales.
c) **Problemas NP** (donde NP se entiende por polinomial no determinístico). En general, esta clase incluye todos los problemas que tienen algoritmos exponenciales pero que no se ha probado que no se pueden resolver con algoritmos de tiempo polinomial.
d) **Problemas P** (donde P se entiende por polinomial). Esta clase incluye todos los problemas que tienen algoritmos de tiempo polinomial, y se considera que son resolubles eficientemente. Hay gente que considera a esta clase como una subclase propia de la clase c), es decir $P \subseteq NP$, Flores [10].

Un problema se dice polinomial si existe un algoritmo para el cual el tiempo requerido para su solución, esta acotado por una función polinomial del tamaño del problema. Por ejemplo el problema de asignación se puede resolver mediante el algoritmo "Húngaro" el cual necesita un tiempo de cálculo que crece según $n^3$, donde $n$ es el número de trabajadores y puestos a emparejar Díaz [5]. Sin embargo no todos los problemas combinatorios son polinomiales.

En 1971 Cook [4] demostró que hay problemas en NP que son especialmente difíciles, los denominados NP-completos de los cuales existe una larga lista y además la mayoría de estos son problemas del mundo real, Lenstra [15]. Hasta la fecha nadie ha demostrado que $NP \subseteq P$. Es decir no se han encontrado algoritmos eficientes para los problemas NP-completos, Díaz [5].

Problemas como el *TSP* o el de la la mochila se conocen como NP completos Coffman [3], el *TSP* ha sido objeto de atención de muchos investigadores ya que es representativo de una larga lista de problemas combinatorios específicamente Karp mostró que si se logra encontrar un algoritmo eficiente para el *TSP*, entonces existen algoritmos eficientes para problemas equivalentes.

Esquemáticamente se puede ver de la siguiente manera

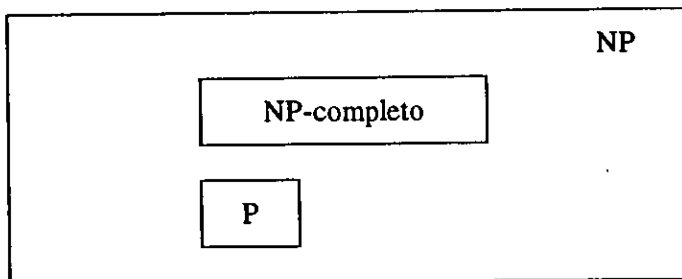



Figura 1.3.1

## 1.4 Técnicas heurísticas

Con la idea de combatir la enumeración total (que es el único método exacto para una serie de problemas combinatorios, para los cuales había de ofrecer una solución) surge la idea de heurística como método de enumeración parcial.

El término heurística se deriva del griego ***heurikein*** que significa encontrar o descubrir, sin embargo, las técnicas heurísticas no garantizan encontrar una solución óptima pero son muy útiles como estrategia para encontrar soluciones cercanas a una óptima. El término heurístico es usado en contraste a los métodos exactos que si garantizan encontrar una solución óptima global.

Una técnica heurística es aquella que ayuda a guiar al proceso de búsqueda mejorando en cada intento de aproximación su eficiencia. (ofreciendo un costo computacional razonable).

En los últimos años ha crecido el número de artículos publicados que tratan de cómo las heurísticas se han utilizado para resolver problemas de optimización combinatoria. Parece doble la causa de este gran interés:

a) Por un lado, el desarrollo del concepto de *complejidad computacional* ha proporcionado las bases para explorar las técnicas heurísticas.
b) Por otro lado, han surgido nuevas técnicas de propósito general muy eficientes para resolver problemas de optimización combinatoria en tiempos de computo razonables.

Las heurísticas son la respuesta a los problemas que presentan explosión combinatoria, sin embargo existen otros argumentos a favor del uso de las técnicas heurísticas.

a) Lo que se está optimizando en general es un modelo de un problema del mundo real, por eso no hay garantía de que la mejor solución del modelo sea también la mejor solución para el problema del mundo real.
b) Las técnicas Heurísticas son usualmente más flexibles y suelen hacer lo mismo con funciones objetivo y/o restricciones más complicadas que los algoritmos exactos.
c) Una heurística resulta adecuada cuando se carece de algún método .exacto para la solución.
d) Cuando requiere mucho espacio de memoria o mucho tiempo para resolverlo.
e) Cuando no se necesita la solución óptima.
f) Se usan como paso intermedio para generar una solución de buena calidad en la aplicación de otro algoritmo. A veces son usadas soluciones heurísticas como punto de partida de algoritmos exactos de tipo iterativo.
g) Las heurísticas generalmente ofrece mas de una solución, por lo cual se tienen mas opciones de decisión.
h) En contraste de las técnicas exactas, resulta sencillo fundamentar una técnica heurística.

Por el contrario, también presenta inconvenientes el uso de métodos heurísticos. Uno de ellos es que por lo general no es posible conocer la calidad de la solución, es decir, cuan

cerca se esta del optimo X*, la solución $X_{heu}$ que nos ofrecen. Existen métodos que nos pueden orientar con respecto a la calidad de la solución obtenida, por ejemplo podría generarse aleatoriamente varias soluciones y si son similares a $X_{heu}$ cabría poner en duda la efectividad de la heurística.

Debe quedar claro que la aplicación de un algoritmo exacto tiene prioridad a cualquier heurística, sobre todo si hay mucho dinero de por medio.

## 1.5 Tipos de heurísticas

Dado el carácter no polinomial de la mayoría de los problemas de optimación combinatoria se hace necesario contar con buenas heurísticas entre las cuales destacan:

i. Métodos constructivos. Este tipo de métodos construyen soluciones factibles usando reglas heurísticas en forma determinística y secuencial. El representante más famoso de estos son los Algoritmos Greedy ( glotones, voraces o ávidos).

ii. Métodos de descomposición. Consisten en dividir el problema en subproblemas más pequeños, siendo la salida de uno la entrada del siguiente, de forma que al resolverlos todos obtengamos una solución para el problema global. Por ejemplo en el caso del TSP, se podrían subdividir en viajes y finalmente unirlos.

iii. Método de búsqueda por vecindades. Tiene como objetivo perfeccionar una solución existente *s*, mediante una búsqueda local en una vecindad bien definida en torno de s en el espacio de factibilidad.

En los problemas de optimización combinatoria los métodos heurísticos sólo inspeccionan un pequeño subconjunto del espacio factible. Una heurística bien diseñada deberá explorar exclusivamente las soluciones más interesantes.

Se explicará un poco más de la última categoría de los métodos heurísticos debido a que será una herramienta básica que utilizaremos posteriormente para el desarrollo de este trabajo de tesis. Para ello definiré algunos los siguientes conceptos.

Una vecindad $V_k(s)$ de una solución *s* es un conjunto de soluciones que pueden ser alcanzadas desde s aplicando una simple operación *k*. Tal operación podría ser el quitar o agregar un objeto a la solución o hacer el intercambio de dos objetos en la solución.

Una operación *k*-intercambio, *k* entero positivo, aplicado a una solución s genera una vecindad de s formada por todas las permutaciones que son diferentes a s en *k* componentes y la cantidad de vecinos es $C_n^k$ ,donde *n* es el tamaño de la entrada y $k \leq n$.

Por ejemplo si $k = 2$, entonces la operación es 2-intercambio, que consiste en intercambiar 2 objetos en una solución; por ejemplo dada una solución $s = (1,2,3,4)$ y tomando $k = 2$, su vecindad $V_2(s)$ está formada por las soluciones $s_1 = (2,1,3,4)$, $s_2 = (3,2,1,4)$, $s_3 = (4,2,3,1)$, $s_4 = (1,3,2,4)$, $s_5 = (1,4,3,2)$ y $s_6 = (1,2,4,3)$.

El método de búsqueda local o de descenso se basa en buscar de entre los elementos de la vecindad $V_k(s)$ de la solución actual, aquel que tenga un mejor valor de acuerdo con

algún criterio predefinido; moverse a él y repetir la operación hasta que se considere que no es posible hallar una mejor solución, ya sea por que no hay ningún elemento en la vecindad de la solución actual, o bien porque se verifique algún criterio de parada ( por ejemplo número máximo de iteraciones). Esta solución final probablemente sea un óptimo global, aunque con respecto a su vecindad es un óptimo local supongamos que las soluciones de un ejemplo tienen los siguientes costos asociados 37, 45, 31, 26, 33, 35 y 30 respectivamente, si es un problema de minimización, entonces se observa que el óptimo en esta vecindad es la solución $s_3$, esto implica un movimiento de la solución actual $s$ a la solución óptima $s_3$ y ahora se obtiene la vecindad $V_2(s_3)$ para reiniciar el mismo procedimiento anterior.

Uno de los mayores inconvenientes con los que se enfrentan estas técnicas es la existencia de óptimos locales que no sean absolutos. Si a lo largo de la búsqueda se cae en un optimo local, en principio la heurística no sabría continuar pues se quedaría "pegada" en ese punto. Una primera posibilidad para salvar esa dificultad consiste en reiniciar la búsqueda desde otra solución inicial y confiar que, en esta ocasión, la exploración siga por otros derroteros.

Aunado a esto, muchos problemas de optimización combinatoria son problemas específicos, de manera que un algoritmo de una técnica heurística que funciona para un problema, puede no ser útil para resolver un problema diferente. Sin embargo en los últimos años se han desarrollado heurísticas de propósito general llamadas metaheurísticas que en cierta forman tratan de salvar los dos inconvenientes anteriores. La mayoría de las metaheurísticas están encuadradas en los métodos de búsqueda por vecindades, Díaz [3].

La palabra metaheurística fue acuñada por Fred Glober al mismo tiempo que surgió el termino Búsqueda Tabú (1986).

Una metaheurística es una estrategia maestra que guía y modifica otras heurísticas para producir soluciones más allá de las que normalmente son generadas por otros métodos.

## 1.6 La metaheurística GRASP

Existen varias metaheurísticas Algoritmos Genéticos, Recocido Simulado y Búsqueda Tabú que han tenido éxito en la solución de problemas combinatorios, la metaheurística GRASP (*Greedy Randomized Adaptative Search Procedures*) es la más reciente de estas técnicas, fue desarrollada originalmente por Feo y Resende [7] al estudiar un problema de cobertura de alta complejidad combinatoria. Cada iteración en GRASP consta generalmente de dos pasos: la fase de construcción y el procedimiento de búsqueda local. En el primero se construye una solución tentativa que luego es mejorada mediante un postprocesamiento para mejorar la solución obtenida en la primera etapa hasta que se llega a un óptimo local. En el capítulo 4 nos ocuparemos de esta metaheurística en detalle.

# Capítulo 2

# Problema de Asignación Cuadrática

## 2.1 Introducción.

Este capítulo describe el Problema de Asignación Cuadrática, (*QAP*, *Quadratic Assignment Problem*). Mencionaremos algunos datos históricos y métodos que han tratado de resolver el problema.

El *QAP* será planteado como un problema de optimización combinatoria y como un problema de programación entera. Se muestra un ejemplo para los dos planteamientos, se dan antecedentes de su complejidad y mencionare algunas de sus distintas aplicaciones.

## 2.2 Antecedentes del *QAP*.

El Problema de Asignación Cuadrática fue propuesto originalmente en 1957 por Koopmans y Beckman [13] analizando la localización de actividades económicas. En los años 60 fue trabajado por Lawer [14]. El *QAP* es un problema clásico en optimización combinatoria y básicamente consiste en encontrar una asignación óptima de $n$ recursos a $n$ localidades con el propósito de minimizar el costo de transporte, dados una matriz de requerimientos de unidades a transportar y el respectivo costo de transporte por unidad entre las localidades. El espacio de soluciones del *QAP* es de tamaño $n!$. En 1976 Sahni y González [28] probaron que el *QAP* pertenece a la clase de problemas NP-hard. El *QAP* en los últimos años ha recibido mucha atención por parte de algunos investigadores tratando de resolverlo, mediante métodos heurísticos ante la ineficiencia de los métodos exactos para dimensiones mayores a 20 Pardalos [20],. Algunos algoritmos exactos desarrollados para el *QAP* están basados en ramificación y acotamiento (branch and baund), por ejemplo Algoritmos de asignación simple, Asignación de parejas y Algoritmos de posicionamiento relativo Pardalos [20]. Los primeros algoritmos exactos en paralelo fueron propuestos por Roucairol [27] y Pardalos [19], pero sus experimentos computacionales se limitaron a problemas de tamaño $n \leq 15$.

Recientemente, ante el surgimiento de las metaheurísticas tales como la Búsqueda Tabú, Algoritmos Genéticos, Recocido Simulado y la más reciente de ellas el GRASP Feo[8] y

Resende [24], han sido empleadas para proponer soluciones de muy buena calidad. Esta última será objeto de estudio en el próximo capítulo.

Se encuentra disponible una página en Internet llamada QAPLIB, en donde se encuentran las disertaciones, artículos, resultados y problemas de prueba del *QAP* más recientes; la dirección es: http://www.imm.dtu.dk/~sk/qaplib/#address.

## 2.3 Modelos del Problema de Asignación Cuadrática.

A continuación se presentan dos formas de plantear el *QAP* con un ejemplo en cada caso.

### 2.3.1 El modelo de optimización combinatoria para el *QAP*

Matemáticamente una forma de presentar el *QAP* es la siguiente:

Sean $\aleph=\{1,2,\ldots, n\}$ y $F=(f_{ij})$ y $D=(d_{kl})$ dos matrices cuadradas de $n$ x $n$; se trata de encontrar la asignación de $n$ plantas a $n$ localidades, es decir una permutación $p \in \Pi_\aleph$ que minimice

$$\sum_{i=1}^{n}\sum_{j=1}^{n} f_{ij}\, d_{P(i)P(j)}$$

donde $\Pi_\aleph$ es el conjunto de todas las permutaciones del conjunto $\aleph$. $f_{ij}$ representa el flujo de materiales de la planta $i$ a la planta $j$ y $d_{kl}$ es la distancia de la ciudad $k$ a la ciudad $l$.

Supongamos que las matrices de flujo y distancias $F$ y $D$ respectivamente son simétricas, es decir $f_{ij} = f_{ji}$ y $d_{kl} = d_{lk}$, además $f_{ij} = 0$ y $d_{ij} = 0$, para $i = j$, entonces podemos escribir las instancias de datos en una sola matriz que compacte a $F$ y $D$ como sigue:

$$C = \begin{bmatrix} 0 & d_{12} & d_{13} & d_{14} & \cdots & d_{1n} \\ f_{21} & 0 & d_{23} & d_{24} & \cdots & d_{2n} \\ f_{31} & f_{32} & 0 & d_{34} & \cdots & d_{3n} \\ \vdots & \vdots & \vdots & \ddots & \cdots & \vdots \\ & & & & 0 & d_{n-1\,n} \\ f_{n1} & f_{n2} & \cdots & \cdots & f_{n\,n-1} & 0 \end{bmatrix}$$

**Ejemplo:** La instancia para $n = 5$, tomada de las matrices del artículo de Nugent [18] se da en la siguiente matriz compacta

$$c=\begin{bmatrix} 0 & 1 & 1 & 2 & 3 \\ 5 & 0 & 2 & 1 & 2 \\ 2 & 3 & 0 & 1 & 2 \\ 4 & 0 & 0 & 0 & 1 \\ 1 & 2 & 0 & 5 & 0 \end{bmatrix}$$

El espacio de soluciones es un conjunto que contiene 5! = 120 elementos, que a continuación se muestran en la tabla con sus respectivos costos calculados con la fórmula

$$\sum_{i=1}^{5}\sum_{j=1}^{5} f_{ij}\, d_{P(i)P(j)}$$

| Perm. | costo | Perm. | costo | Perm. | costo | Perm. | costo | Perm. | costo | Perm. | costo |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 12345 | 33 | 15324 | 36 | 24513 | **25** | 34125 | 35 | 42315 | 41 | 51423 | 45 |
| 12354 | 34 | 15342 | 41 | 24531 | 30 | 34152 | 35 | 42351 | 36 | 51432 | 45 |
| 12435 | 33 | 15423 | 41 | 25134 | 37 | 34215 | 35 | 42513 | 31 | 52134 | 35 |
| 12453 | 39 | 15432 | 41 | 25143 | 36 | 34251 | 40 | 42531 | 26 | 52143 | 34 |
| 12534 | 30 | 21345 | 29 | 25314 | 37 | 34512 | **25** | 43125 | 31 | 52314 | 45 |
| 12543 | 35 | 21354 | 30 | 25341 | 41 | 34521 | 30 | 43152 | 31 | 52341 | 39 |
| 13245 | 33 | 21435 | 39 | 25413 | 30 | 35124 | 37 | 43215 | 41 | 52413 | 38 |
| 13254 | 34 | 21453 | 35 | 25431 | 35 | 35142 | 36 | 43251 | 36 | 52431 | 33 |
| 13425 | 33 | 21534 | 36 | 31245 | 29 | 35214 | 37 | 43512 | 31 | 53124 | 35 |
| 13452 | 39 | 21543 | 31 | 31254 | 30 | 35241 | 41 | 43521 | 26 | 53142 | 34 |
| 13524 | 30 | 23145 | 30 | 31425 | 39 | 35412 | 30 | 45123 | 37 | 53214 | 45 |
| 13542 | 35 | 23154 | 31 | 31452 | 35 | 35421 | 35 | 45132 | 37 | 53241 | 39 |
| 14235 | 34 | 23415 | 40 | 31524 | 36 | 41235 | 36 | 45213 | 31 | 53412 | 38 |
| 14253 | 40 | 23451 | 41 | 31542 | 31 | 41253 | 32 | 45231 | 30 | 53421 | 33 |
| 14325 | 34 | 23514 | 37 | 32145 | 30 | 41325 | 36 | 45312 | 31 | 54123 | 39 |
| 14352 | 40 | 23541 | 37 | 32154 | 31 | 41352 | 32 | 45321 | 30 | 54132 | 39 |
| 14523 | 36 | 24135 | 35 | 32415 | 40 | 41523 | 38 | 51234 | 40 | 54213 | 33 |
| 14532 | 36 | 24153 | 35 | 32451 | 41 | 41532 | 38 | 51243 | 35 | 54231 | 32 |
| 15234 | 36 | 24315 | 35 | 32514 | 37 | 42135 | 31 | 51324 | 40 | 54312 | 33 |
| 15243 | 41 | 24351 | 40 | 32541 | 37 | 42153 | 31 | 51342 | 35 | 54321 | 32 |

### 3.2.2 El modelo de programación entera para el *QAP*

Hay suficiente evidencia para asegurar que los métodos de programación entera están en desventaja con los métodos heurísticos Rardin [22]. El planteamiento del *QAP* para programación entera [13] se puede representar como sigue:

Sea $c_{i,j,k,l}$ el costo de asignar $i$ a $j$ y $k$ a $l$ y $\aleph = \{1,2, \ldots n\}$.

Minimizar

$$\sum_i \sum_j \sum_{k>i} \sum_{l \neq j} c_{i,j,k,l} \, X_{i,j} \, X_{k,l}$$

Sujeto a las restricciones

$$\sum_i X_{ij} = 1 \quad \forall\, i \in \aleph$$
$$\sum_j X_{ij} = 1 \quad \forall\, j \in \aleph$$
$$X_{ij} = 0 \text{ ó } 1 \quad \forall\, i, j \in \aleph$$

**Ejemplo:** Se tienen cuatro locales se tienen para cuatro tiendas en una plaza comercial. Las distancias en metros entre los locales y el flujo de clientes, en miles, entre las tiendas se da en la siguiente tabla compacta.

$$C = \begin{bmatrix} 0 & 80 & 150 & 170 \\ 5 & 0 & 130 & 100 \\ 3 & 3 & 0 & 120 \\ 7 & 8 & 3 & 0 \end{bmatrix}$$

El gerente quiere encontrar una asignación óptima de las tiendas a los locales de manera que minimice los inconvenientes para los clientes.

Un planteamiento para este problema con el modelo de asignación cuadrática para programación entera es el siguiente:

Minimizar

$5(80X_{11}X_{22}+150X_{11}X_{23}+170X_{11}X_{24}+80X_{12}X_{21}+130X_{12}X_{23}+100X_{12}X_{24}+150X_{13}X_{21}+130X_{13}X_{22}+120X_{13}X_{24}+170X_{14}X_{21}+100X_{14}X_{22}+120X_{14}X_{23})$ +
$2(80X_{11}X_{32}+150X_{11}X_{33}+170X_{11}X_{34}+80X_{12}X_{31}+130X_{12}X_{33}+100X_{12}X_{34}+150X_{13}X_{31}+130X_{13}X_{32}+120X_{13}X_{34}+170X_{14}X_{31}+100X_{14}X_{32}+120X_{14}X_{33})$ +
$7(80X_{11}X_{42}+150X_{11}X_{43}+170X_{11}X_{44}+80X_{12}X_{41}+130X_{12}X_{43}+100X_{12}X_{44}+150X_{13}X_{41}+130X_{13}X_{42}+120X_{13}X_{44}+170X_{14}X_{41}+100X_{14}X_{42}+120X_{14}X_{43})$ +
$3(80X_{21}X_{32}+150X_{21}X_{33}+170X_{21}X_{34}+80X_{22}X_{31}+130X_{22}X_{33}+100X_{22}X_{34}+150X_{23}X_{31}+130X_{23}X_{32}+120X_{23}X_{34}+170X_{24}X_{31}+100X_{24}X_{32}+120X_{24}X_{33})$ +
$8(80X_{21}X_{42}+150X_{21}X_{43}+170X_{21}X_{44}+80X_{22}X_{41}+130X_{22}X_{43}+100X_{22}X_{44}+150X_{23}X_{41}+130X_{23}X_{42}+120X_{23}X_{44}+170X_{24}X_{41}+100X_{24}X_{42}+120X_{24}X_{43})$ +
$3(80X_{31}X_{42}+150X_{31}X_{43}+170X_{31}X_{44}+80X_{32}X_{41}+130X_{32}X_{43}+100X_{32}X_{44}+150X_{33}X_{41}+130X_{33}X_{42}+120X_{33}X_{44}+170X_{34}X_{41}+100X_{34}X_{42}+120X_{34}X_{43})$.

sujeto a las restricciones

$$X_{11}+X_{12}+X_{13}+X_{14} = 1$$
$$X_{21}+X_{22}+X_{23}+X_{24} = 1$$
$$X_{31}+X_{32}+X_{33}+X_{34} = 1$$
$$X_{41}+X_{42}+X_{43}+X_{44} = 1$$
$$X_{11}+X_{21}+X_{31}+X_{41} = 1$$
$$X_{12}+X_{22}+X_{32}+X_{42} = 1$$
$$X_{13}+X_{23}+X_{33}+X_{43} = 1$$
$$X_{14}+X_{24}+X_{34}+X_{44} = 1$$

La función objetivo calcula el total de flujo-distancia para todos los pares de tiendas y todas las posibles asignaciones a los locales. Las restricciones de asignación aseguran que una tienda será asignada sólo a un local. Una asignación óptima de tiendas a locales es la siguiente: la tienda 1 en el local 1, la tienda 2 en el local 2, la tienda 3 en el local 3 y la tienda 4 en el local 4 para un total de flujo-distancia de 3160 cliente-metro.

## 2.4 Aplicaciones del *QAP*

Se han hallado múltiples aplicaciones del *QAP* en áreas tan diversas como la ingeniería y la logística. Algunos de los problemas tratados son: la minimización total del cableado en el diseño de chips electrónicos, la localización óptima de plantas industriales, la secuenciación de trabajos, la distribución de servicios médicos en un hospital grande y el acomodo de trailers sobre plataformas de ferrocarril en un entorno de transporte intermodal. Para más aplicaciones consulte las referencias Feo[6] y Resende [25].

## 2.5 Complejidad del *QAP*

El *QAP*, computacionalmente, es uno de los problemas más difíciles de optimización combinatoria ya que su espacio de soluciones crece factorialmente a medida que $n$ crece, donde $n$ es la dimensión de la instancia a resolver .

La solución óptima, para una instancia de dimensión 20, es una permutación que tiene 20 componentes, la cual pertenece a un conjunto de tamaño 20! (2432902023163674620). Para una instancia de tamaño 21, su espacio de soluciones será de tamaño 21 veces el número que está entre paréntesis.

Es de pensar que un método exacto para resolver este problema es la búsqueda exhaustiva, sin embargo también es obvio que resulta prohibitivo por su explosión combinatoria.

Demostrar que un problema A $\in$ NP es NP-completo, consiste en mostrar que A contiene un conocido problema NP-completo como un caso especial suyo; esta técnica es conocida como de restricción ver Galve[11].

Afirmamos que el *QAP* es un problema NP-completo. En efecto, el *QAP* $\in$ NP pues dada una instancia concreta de un problema (veamos el ejemplo de la sección 2.3.1) es posible decidir en tiempo polinomial, si por ejemplo, una solución es la permutación (3,2,4,5,1), tiene un costo mayor que 25; para ello bastará calcularlo con la fórmula del mismo ejemplo.

Por otro lado si en el *QAP* se toma todos los flujos entre las plantas como la unidad, el problema queda reducido al problema del agente viajero, el cual es NP-completo; esto quiere decir que el *QAP* contiene al problema del agente viajero.

La prueba original de esta proposición se publicó en 1976 por Sahni y González [28].

# Capítulo3

# GRASP

## 3.1 Introducción.

En el presente capítulo se desarrolla la metaheurística GRASP, se menciona también algunas de las distintas áreas en las cuales ha sido aplicada esta técnica y se implementa al *QAP* bajo tres estructuras vecinales a saber 2-intercambio, λ-intercambio y Nstar.

## 3.2 Metaheurística GRASP.

GRASP es la más joven de las metaheurísticas surgidas durante la década de los ochentas con el fin de resolver problemas difíciles en el campo de la optimización combinatoria. GRASP son las siglas de *Greedy Randomized Adaptative Search Procedure* en inglés.

Una traducción literal de las palabras que forman el acrónimo seria "procedimientos de búsqueda glotones, aleatorizados y adaptativos",esta metaheurística, fue desarrollada originalmente por Feo y Resende en 1989 con el artículo *A probabilistic heuristic for a computationally dificult set covering problem*, Feo [7] el cual es considerado como el acta de nacimiento de GRASP.

Un GRASP es un proceso iterativo, cada iteración en GRASP consta de dos pasos: la fase de construcción y el procedimiento de búsqueda local. En el primero se construye una solución factible inicial, que luego es mejorada mediante un procedimiento de intercambio ,por ejemplo hasta que se llega a un óptimo local Li [17].

Una vez que se han ejecutado las dos fases, la solución obtenida se almacena y se procede a efectuar otra iteración, guardando cada vez la mejor solución que se haya encontrado hasta el momento. En la figura 3.2.1 se muestra un algoritmo que ilustra está metaheurística.

```
Procedure GRASP
    EntradaInstancia();
    While (criterio de paro no satisfecho) do
        ConstructSolucionGreedyRandomizeAdaptative();
        Postprocesamiento();
        ActualizarSolucion();
End {While}
    Return (Mejor Solucion)
End {GRASP}
```

Figura 3.2.1. Pseudocódigo genérico de GRASP.

### 3.2.1 Fase de construcción y sus componentes.

Las principales componentes de la fase de construcción son: una función de evaluación voraz, un procedimiento de elección al azar y un proceso de actualización adaptativo.

El objetivo de la fase de construcción es generar una solución factible de buena calidad, y entre más eficiente sea esta fase en términos de calidad se espera que el trabajo de la fase de postprocesamiento sea menor y así cada iteración GRASP ocurriría más rápido. En la fase de construcción, GRASP genera una lista de candidatos formada por elementos de alta calidad, esta lista es llamada lista restringida de candidatos ( LRC ). La solución inicial se construye iterativamente considerando un elemento cada vez. Es necesario visualizar cada solución como una estructura que puede obtenerse mediante incrementos de una base donde cada nuevo elemento se elige de un conjunto de posibles candidatos. En cada iteración del procedimiento constructivo, un elemento es elegido en forma aleatoria de la lista de candidatos para añadirlo a la subestructura como parte de la solución que se construye. La adición de un elemento a la subestructura se determina mediante una función de tipo voraz, esta función mide el beneficio de la selección del cada elemento, mientras la selección de un elemento de esa lista de candidatos depende de los que se hayan elegido previamente. Es decir, el método es a la vez voraz, aleatorio y adaptativo.

Basándose en la construcción eficiente de una solución inicial puede conseguirse una reducción en el numero de pasos necesarios para alcanzar un óptimo local en la fase de mejora. Se dice (y los resultados empíricos parecen demostrarlo) que el tiempo total de calculo en cada iteración GRASP puede reducirse a base de diseñar buenos procedimientos de construcción Díaz [5].

**La descripción general de las principales componentes de está fase es:**

**Greedy.**

La primer tarea del diseño de un GRASP es la elección de una función voraz que guíe la construcción de soluciones. La idea detrás de todo enfoque ávido es la de tomar la mejor decisión disponible en cada paso.

**Randomized.**

Para obtener diversidad en las soluciones un procedimiento que se dice voraz no puede usar reglas lexicográficas para romper empates entre los candidatos, ya que se obtendría un método determinístico y una aplicación repetida de tal procedimiento generará siempre la misma solución.

Se dice que un algoritmo esta aleatorizado si su respuesta esta determinada no solo por los datos de entrada sino por los valores producidos por un generador de números aleatorios, GRASP tiene esta característica.

Una estrategia aleatorizada viene a ser particularmente útil cuando existen varias formas según las cuales un algoritmo puede efectuar un paso determinado, pero resulta difícil garantizar que alguna de ellas sea la mejor elección. Es frecuente que un algoritmo tenga que hacer muchas elecciones durante su ejecución. Si el beneficio de las buenas elecciones supera el costo de las malas una selección al azar de buenas y malas elecciones puede a la larga producir buenos resultados.

Esta es la idea en la cual se apoya este componente de GRASP. Después de varias replicas de soluciones se habrá extraído en realidad una muestra de la población de todas las configuraciones posibles de un problema, y, por supuesto se espera que esta muestra no sea representativa sino que tenga un fuerte sesgo hacia los elementos de la más alta evaluación.

**Adaptative.**

Una vez elegido un movimiento, se aumenta la subestructura y se procede a recalcular los beneficios asociados con cada elemento. En este proceso resulta particularmente importante reflejar los cambios producidos al incorporar la selección previa como una componente de la solución.

**Lista restringida de candidatos.**

En cada etapa de la fase de construcción surge una lista de movimientos admisibles, entonces la estrategia que se toma es de ordenarlos de manera decreciente con respecto a su beneficio medido por la función voraz.

Una forma de restringir la lista es mediante cardinalidad: el candidato se elige aleatoriamente de los primeros $k$, donde $k$ es un número especificado por el usuario, otra forma es tomar los elementos que están en un cierto rango de la lista y otra es usar conjuntamente ambos enfoques, o sea es la lista restringida se forma con la intersección de los primeros $k$ elementos y aquellos que se encuentran dentro del rango previamente definido. El compromiso entre ser muy restrictivo o muy laxo depende de la elección de los parámetros que restringen la lista, se debe encontrar un equilibrio en el tamaño de la lista ya que si se tiene escasa variabilidad se corre el peligro de no encontrar buenas soluciones, por otra parte si cualquier candidato es elegible la calidad promedio de las soluciones será muy

pobre, pudiéndose requerir un número enorme de iteraciones GRASP para lograr un buen resultado. La experimentación es la forma de encontrar los valores para los parámetros.

```
Procedure ConstructSolucionGreedyRandomizeAdaptative
   1        Solucion = { };
   2            While (solucion incompleta) do
   3                CrearLCR();
   4                s = ElementoSeleccionadoAleatoriamente(LCR)
   5                Solucion = Solucion ∪ {s};
   6                AdaptativoAvido(s,LCR);
   7            end {while}
   8        Return (SolucionCompleta)
End {ConstructSolucionGreedyRandomizeAdaptative}
```

Figura 3.2.2. Pseudocódigo para la fase de construcción.

### 3.2.2 Fase de postprocesamiento.

En la fase de postprocesamiento, GRASP toma como solución de arranque la solución producida por la fase de construcción con el fin de mejorarla aplicándole procedimientos que van desde intercambio simple hasta formar híbridos con otras metaheurísticas como Búsqueda Tabú o Recocido Simulado.

La fase de construcción produce una diversidad de buenas soluciones aunque sin garantía de que sean óptimas, pero si se espera que al menos una de ellas este en una vecindad de la solución óptima, o al menos en la de una solución muy cercana a un valor óptimo. La figura 3.2.3 ilustra un algoritmo de búsqueda local para mejorar la solución factible inicial.

```
Procedure local ( p, V(p), s )
1   While s no sea optima local do
2       Encontrar una  mejor solución t ∈ V(s);
3       Sea s = t ;
4   End ;{ While }
5   Return ( s como optima local )
End local;
```

Figura 3.2.3. Pseudocódigo genérico para la búsqueda local.

## 3.3 Aplicaciones de GRASP.

GRASP ha sido implementado con gran éxito a problemas clásicos de optimización combinatoria y sus aplicaciones se pueden dividir en dos categorías:

**Problemas de investigación de operaciones:**

- Problemas de secuenciación de rutas
- Problemas de programación y planificación
- Lógica
- Problemas de partición
- Problemas de localización
- Problemas de asignación
- Problemas de flujo en redes no convexas

**Aplicaciones industriales:**

- Manufactura
- Transporte
- Telecomunicaciones
- Trazado Automático
- Sistemas de potencia eléctrica
- Militar
- Biología

Las aplicaciones con más detalle se pueden consultar en Resende [24] y en Festa [9] se describe la mayoría de los artículos publicados hasta 1999 usando GRASP.

## 3.4 Un GRASP para el *QAP*.

Se aplica la metodología de GRASP para resolver el problema clásico de asignación cuadrática que básicamente consiste en encontrar una asignación optima de *n* recursos en *n* lugares, con el propósito de minimizar el costo de total de transporte, como se describe en detalle en el capitulo 2.

El diseño de este GRASP ha sido empleado por algunos investigadores para resolver el *QAP* para diferentes instancias Li [17], Resende[24] y Pardalos [19], se puede resumir de la siguiente forma, etapa 1, se genera una lista de candidatos, la cual ha sido previamente restringida por dos parámetros, se toma aleatoriamente uno de estos candidatos del cual se desprenden dos primeras asignaciones, etapa 2, se van agregando una a una las restantes *n*-2 asignaciones con respecto de una función voraz, con lo cual habiendo completado la solución se ha producido una solución factible que se espera este en una vecindad de una solución óptima o al menos una muy cercana a una solución óptima, con esto finaliza la primera fase o fase constructiva. Empezamos la segunda fase o fase de mejoramiento tomando como solución inicial la solución emanada de la primera fase empleando algún procedimiento de búsqueda local, cuando finalice este procedimiento se tendrá una solución óptima local, que pudiera ser también óptimo global.

•**Fase de construcción para el *QAP*.**

**Etapa 1.** Las dos asignaciones iniciales se hacen simultáneamente, específicamente diremos que el recurso $i$ es asignado a la localidad $k$ y el recurso $j$ es asignado a la localidad $l$, cuando su costo correspondiente a este par de asignaciones es $f_{ij} d_{kl}$.

Sean α, β, ( 0 < α, β < 1 ) los parámetros que restringen la lista de candidatos, $F = (f_{ij})$ y $D = ( d_{kl} )$ las matrices simétricas $n$ x $n$ con ceros en la diagonal de entrada con las cuales se forma una matriz compacta cuadrada no simétrica.

$$C = \begin{bmatrix} 0 & d_{12} & d_{13} & d_{14} & \cdots & d_{1n} \\ f_{21} & 0 & d_{23} & d_{24} & \cdots & d_{2n} \\ f_{31} & f & 0 & d_{34} & \cdots & d_{3n} \\ \vdots & \vdots & \vdots & \ddots & \cdots & \vdots \\ & & & & 0 & d_{n-1\,n} \\ f_{n1} & f_{n2} & \cdots & \cdots & f_{n\,n-1} & 0 \end{bmatrix}$$

Sea [x] la parte entera de x. Sea m = $n$ ($n$-1)/2 el número de entradas en los triángulos inferior y superior de la matriz compacta. Se procede a listar estas entradas de las distancias y los flujos en orden creciente y decreciente respectivamente, es decir

$$d_{k_1 l_1} \leq d_{k_2 l_2} \leq d_{k_3 l_3} \leq \ldots \leq d_{k_m l_m}$$

$$f_{i_1 j_1} \geq f_{i_2 j_2} \geq f_{i_3 j_3} \geq \ldots \geq f_{i_m j_m}$$

Ahora tenemos dos listas ordenadas, usaremos el parámetro β para restringir ambas listas, por lo cual se cortan hasta el elemento [βm]. Se genera una nueva lista de costos multiplicando las distancias por los flujos en el orden correspondiente así, se tiene la nueva lista

$$f_{i_1 j_1} d_{k_1 l_1},\ f_{i_2 j_2} d_{k_2 l_2},\ f_{i_3 j_3} d_{k_3 l_3},\ \ldots,\ f_{i_m j_m} d_{k_m l_m}$$

$$f_{i_1 j_1} d_{k_1 l_1},\ f_{i_2 j_2} d_{k_2 l_2},\ \ldots,\ f_{i_{[\beta m]} j_{[\beta m]}} d_{k_{[\beta m]} l_{[\beta m]}}$$

La ultima lista es ordenada en forma creciente y usamos el parámetro α para obtener la lista restringida de candidatos (LRC) definitiva de la cual solamente se tomaran los primeros [αβm] elementos y se elegirá aleatoriamente un elemento $f_{ij}$ $d_{kl}$ que representa un

costo de hacer un par de asignaciones ( $i, k$ ) y ( $j, l$ ) , es decir tenemos dos componentes de la solución, que para simplificar será escrita como permutación, donde la componente $k$-ésima y la componente $l$-ésima son colocadas. Aquí se puede apreciar la componente aleatoria del método. Con esto concluye la primera etapa de la fase de construcción.

**Etapa 2.** En esta etapa lo que se pretende es completar la solución inicial calculando las $n$-2 asignaciones restantes, mediante un procedimiento ávido que produce una a una las asignaciones que tienen el costo mínimo con respecta a las asignaciones ya existentes y que en caso de empate se romperá aleatoriamente y apoyándose en una componente adaptativa que se encarga de actualizar la solución a medida que esta se va construyendo.

Sea

$$\Gamma = \{ (j_1, l_1), (j_2, l_2), \ldots, (j_r, l_r) \}$$

el conjunto de asignaciones que esta en construcción. La etapa 2 inicia con $|\Gamma| = 2$ a consecuencia de los resultados de la etapa 1.

Sea

$$C_{ik} = \sum_{(j,l)\in\Gamma} f_{ij} d_{kl}$$

el costo de asignar la fabrica $i$ a la localidad $k$ con respecto a las asignaciones ya existentes. Seleccionamos de las parejas ($i$, $k$) no asignadas la que tenga el costo mínimo $C_{ik}$, en esto consiste el procedimiento voraz.

En esta parte también hay una lista restringida de candidatos se ordenan los $C_{ik}$ en forma creciente y se toma aletoriamente uno de los primeros [$\alpha z$], donde z es la cantidad de parejas aun no asignadas, nuevamente aparece la componente aleatoria.

La componente adaptativa de GRASP tiene como función actualizar el conjunto $\Gamma$ adicionando nuevas parejas asignadas, es decir

$$\Gamma = \Gamma \cup \{( i, k )\}$$

Al finalizar esta etapa concluye también la primera fase, se ha construido una solución contenida en el conjunto

$$\Gamma = \{ (j_1, l_1), (j_2, l_2), \ldots, (j_n, l_n) \}$$

ordenando las primeras componentes de las parejas, tomamos las segundas componentes para formar la permutación equivalente a la solución. En resumen tenemos una solución de buena calidad para arrancar la segunda fase.

```
Procedure etapa2(α,(j1,l1),(j2,l2))
  1    Γ= {(j1,l1),(j2,l2)};
  2    While |Γ| ≤ n  do
  3    z = 0;
  4       for i = 1 to n;
  5          for j = 1 to n;
  6             if (i,j)∉ Γ then
  7                   C_ik = Σ_{(i,j)∈Γ} f_ij d_kl ;
  8                   inheap (C_ik);
  9                   z = z +1;
  10             end{if};
  11          end{for};
  12      end{for};
  13      s = random(1,..., [αz]);
  14      C_ik = outheap(s)
  15      Γ = Γ ∪ {(i,k)};
  16    end{while};
end{estapa2};
```

Figura 3.4.1. Algoritmo de la etapa 2 de GRASP de la fase de construcción.

- **Fase de postprocesamiento para el *QAP*.**

Esta fase tiene como misión mejorar la solución que se produjo en la fase de construcción, para nuestro caso aplicaremos un procedimiento de búsqueda local el cual se muestra. en la figura 3.2.3. Este procedimiento también conocido como método de descenso que ya fue mencionado en el capítulo 1, en al sección 1.5 trabaja en forma iterativa remplazando sucesivamente la solución actual por la mejor solución en su vecindad, esta búsqueda termina cuando no se encuentre una mejor solución con respecto a la función de costo.

La llave del éxito de un algoritmo de búsqueda local consiste en la elección adecuada de una estructura vecinal, una técnica eficiente de búsqueda vecinal y la solución inicial. La fase de construcción del GRASP juega un papel importante en el ultimo punto ya que produce buenas soluciones iniciales para la búsqueda local.

• **Búsqueda local para el *QAP*.**

Como mencionamos antes es necesario establecer la estructura vecinal para llevar a cabo la búsqueda local par lo cual definimos lo siguientes conceptos.

Dadas dos permutaciones $p$ y $q$, la diferencia entre $p$ y $q$ se define como

$$\delta(p,q) = \{ i \mid p(i) \neq q(i) \}$$

y la distancia entre $p$ y $q$, esta definida como $d(p,q) = |\delta(p,q)|$. Para diseñar una buena estructura vecinal debemos guiarnos en los siguientes tres principios:

(a) *Tamaño razonable de la vecindad* es decir explorar la vecindad en un número razonable de cálculos.

(b) *Gran varianza en la vecindad* esto implica que si una vecindad especifica consiste de permutaciones $p_1, p_2, \ldots, p_n$ entonces la distancia máxima de todas las permutaciones es grande.

(c) *Alta conectividad en la vecindad* implica que cuando hay un movimiento de $p_i$ a $p_j$ existe una sucesión $\{p_k\}$, con pequeña $\delta(p_k, p_{k+1})$, $k = i, \ldots, j-1$.

Para mayor profundidad en estos conceptos consultar Resende [17] y Pardalos [20].
La vecindad k-intercambio para una permutación $p \in \Pi_N$ se define como

$$V_k(p) = \{ q \mid d(p,q) \leq k, \ 2 \leq k \leq n \}.$$

La vecindad $k$-intercambio es usada en muchos problemas de optimización combinatoria tales como el problema del agente viajero (*TSP*) entre otros. La búsqueda local en la vecindad $k$-intercambio inicia en con una permutación inicial $p$ y examina todas las permutaciones generadas en el intercambio de $k$ elementos en $p$, tomando la mejor permutación como optimo local. El tamaño de esta vecindad es $C_k^n = [n!/k!(n-k)!]$ lo cual implica que, para $k$ grande el esfuerzo computacional puede ser enorme.

Las tres estructuras vecinales para la fase de postprocesamiento son de tamaño del orden de $o(n^2)$, $o(n^3)$ y $o(n^4)$, Resende [17] y se denotan por $V_2(p)$, $V_\lambda(p)$ y $V^*(p)$ respectivamente.

La estructura vecinal 2-intercambio ya se ejemplifico en el capitulo 1. λ–intercambio y Nstar son estructuras más elaboradas y se basan en detectar movimientos buenos para ya no

ejecutarlos por ejemplo si $(i, j)$ es un buen movimiento en una permutación λ-intercambio deja fijos los elementos correspondientes a $i$ y $j$ permitiendo el intercambio solamente entre los elementos restantes de la permutación, en cambio Nstar permite el movimiento de los elementos correspondientes de $i$ o $j$ con los demás elementos, pero no permite el intercambio entre ellos.

Definimos al conjunto de todos los posibles intercambios de parejas en una permutación como sigue:

$$E=\{(i,j) | i \neq j,\, i,j=1,..,n\}.$$

Para una permutación $p$ dada denotamos $V_2(p,E')$ al conjunto de permutaciones en $V_2(p)$ obtenidas por los mejores intercambios en el subconjunto de intercambios $E'$.

La estructura vecinal λ-intercambio esta definida como la unión de una colección de subconjuntos de la vecindad $V_2(p)$. En cada paso de la búsqueda local la permutación actual es denotada por $p_0$ y el conjunto $E_0 = E$. En el paso k de la búsqueda se tienen $p_1,...,p_k$ permutaciones construidas. Sea $p_{k+1}$ la mejor solución en $V_2(p_k,E_k)$. Ahora se construye el conjunto $E_{k+1}$ teniendo $E_k$ de la siguiente forma:

$$E_{k+1}=E_k-\{(i,j) | i \text{ o } j \in \bigcup_{l=0}^{k} \delta(p_l,p_{l+1}),\, i,j=1,...,n\}.$$

El proceso continua hasta que $f(p_k) > f(p_0)$ o $E_k = \varnothing$. Entonces la vecindad de λ-intercambio esta integrada como sigue:

$$V_\lambda(p_0)=\bigcup_{i=0}^{k} V_2(p_i,E_i)$$

La estructura vecinal para Nstar se define de la misma forma que $V_\lambda(p_0)$ solo que en la definición del conjunto $E_{k+1}$ la condición $i$ o $j$, se cambia por $i$ y $j$. Las figuras 5 y 6 muestran los algoritmos para la búsqueda local en estas estructuras vecinales.

```
Procedure λ-intercambio(n,p)
1   Loop = true;
2   While Loop = true do
3           p0 = p ; E0 = E; Done = false;
4           While Done = false and Ek ≠ ∅ do
5                   Encontrar mejor solución pk+1 ∈ V2(pk, Ek);
6                   E_{k+1} = E_k − {(i, j)| i o j ∈ ⋃_{l=0}^{k} δ(p_l, p_{l+1}), i, j = 1,...,n};
7                   if f(pk+1) ≥ f(p0) then Done = true ;
8                   k = k+1;
9           end;{while}
10          p = min{f(p1),..., f(pk)};
11          if f(p) = f(pk) then Loop = false;
12  end; {while}
13  return(p como permutación óptima local) ;
end λ-intercambio;
```

Figura 5. Algoritmo para la búsqueda local en $V_\lambda(p)$.

```
Procedure Nstar(n,p)
1   Loop = true;
2   While Loop = true do
3           p0 = p ; E0 = E; Done = false;
4           While Done = false and Ek ≠ ∅ do
5                   Encontrar mejor solución pk+1 ∈ V2(pk, Ek);
6                   E_{k+1} = E_k − {(i, j)| i and j ∈ ⋃_{l=0}^{k} δ(p_l, p_{l+1}), i, j = 1,...,n};
7                   if f(pk+1) ≥ f(p0) then Done = true ;
8                   k = k+1;
9           end;{while}
10          p = min{f(p1),..., f(pk)};
11          if f(p) = f(pk) then Loop = false;
12  end; {while}
13  return(p como permutación óptima local) ;
end Nstar;
```

Figura 6. Algoritmo para la búsqueda local en $V^*$.

# Capítulo 4

# Paralelismo

## 4.1 Introducción.

En este capítulo se presentan las características principales de la computación paralela, como son los diferentes tipos de memoria, y se mencionaran algunas herramientas de programación y bibliotecas

Una demanda por sistemas más rápidos y eficientes, tanto en el área de computación científica como en el área comercial, ha ido en constante aumento al correr de los años. Los limites físicos existentes para un aumento de la velocidad en un único procesador y su alto costo nos llevan al surgimiento de nuevas arquitecturas para sistemas de computo de alto desempeño. Estas maquina se pueden clasificar en tres tipos:

Multiprocesadores vectoriales: Poseen un numero pequeño de procesadores, donde cada uno es un procesador vectorial de alto desempeño.

Sistemas MPP (Macciavely Parallel Processors): Poseen de centenas a millares de procesadores interconectados, constituyendo una línea principal de computadoras paralelas. Pueden ser de memoria distribuida.

Estaciones de trabajo conectados por redes: Maquinas ínterligadas por red de media y alta velocidad, que pueden trabajar como una maquina virtual de alto desempeño.

## 4.2 ¿Qué es computación paralela?

Computación paralela es el uso de mas de una Unidad de Procesamientos Central (CPU) para la solución de un problema, figura 4.1. La computación paralela abarca el campo entero de las maquinas. Desde dos computadoras personales conectadas vía Ethernet a miles de los más poderosos procesadores en una supercomputadora paralela.

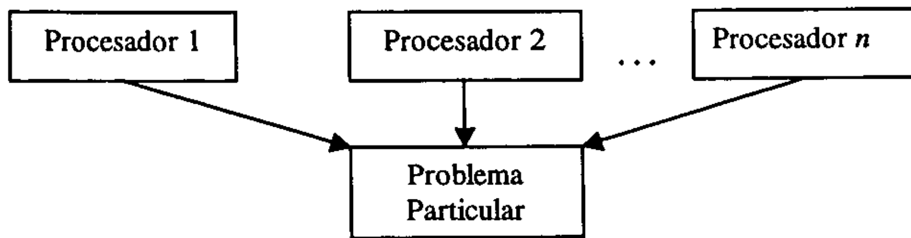



Fig 4.1 Computadora paralela usando mas de un procesador para resolver en conjunto un problema particular.

## 4.3 Terminología.

Algunos de los términos utilizados en procesamiento paralelo son descritos a continuación:

- Tarea o proceso: Es la unidad lógica mínima definible en un programa.
- Ejecución secuencial: Ejecución de un programa o tarea en un único procesador, donde una instrucción es ejecutada independientemente.

- Paralización de código: transformación de un programa secuencial en uno paralelo, a través de la identificación de tareas independientes para un programa, de manera que se ejecuta de modo paralelo. Generalmente requiere modificaciones al código de un algoritmo utilizado en modo secuencial.

- Aceleración de un programa (Speedup): Razón entre los tiempos de ejecución de las versiones secuencial y paralela de un mismo programa o algoritmo. Para un determinado algoritmo, se considera la mejor implementación secuencial como referencia para comparar la implementación paralela.

$$\text{aceleración (Speedup)} = \frac{\text{tiempo de ejecucion de un programa secuencial}}{\text{tiempo de ejecucion de la version paralela}}$$

- Eficiencia de un programa: Razón entre la aceleración de un programa y el número de procesadores utilizados para la ejecución del mismo.

$$\text{eficiencia} = \frac{\text{aceleración}}{\text{número de procesadores}}$$

Intuitivamente este parámetro o medida es la fracción de tiempo que un procesador esta siendo utilizado para ejecutar el algoritmo.

Idealmente, si $p$ procesadores utilizan el 100% de su tiempo solamente para operaciones de computación del algoritmo la eficiencia seria igual a 1. Como un programa paralelo requiere comunicación entre los procesadores, parte del tiempo de los procesadores es gastado en esta actividad llevando a valores de eficiencia menores a 1.

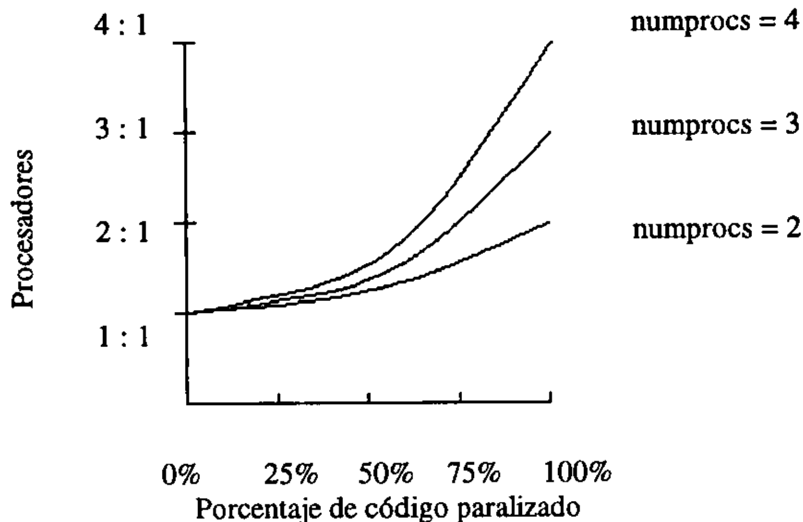
- Ley de Amdahl: Cantidades posibles de aceleración debido a la paralelización de porciones de código. Es decir, la cantidad que podemos acelerar el proceso en relación con tiempo secuencial, dependiendo de la cantidad de código que es factible paralelizar y el numero de procesadores. Como se muestra en la grafica a continuación:

$$Tp = Ts\left(\frac{Pp}{Np}\right) + Ps$$

donde

| | |
|---|---|
| Tp | Tiempo de corrida de programa en paralelo |
| Ts | Tiempo de corrida de programa secuencial |
| Pp | Porcentaje de tiempo usado corriendo en paralelo |
| Np | Numero de procesadores |
| Ps | Porcentaje de tiempo usado en secuencial |

La gráfica 4.1 nos muestra la aplicación de la ley de Amdahl, en donde se ve la eficiencia de los procesadores utilizados en relación con porcentaje de código que ha sido paralelizado.




Grafica 4.1. Ley de Amdahl

Si la porción de código que es factible programarse en paralelo es el 100% y utilizamos 4 procesadores entonces: el tiempo de CPU seria la cuarta parte del tiempo que se utilizó en la versión secuencial, hay que recordar que este seria el caso ideal.

- Sincronización: Coordinación temporal entre procesadores, usada para un paso de información entre ellos. La sincronización es un factor de decremento de la aceleración exigida por un programa paralelo, debido a que algunos procesadores pueden estar inactivos, esperando a que otros procesadores terminen sus respectivas tareas. Una comunicación entre procesadores se dice asíncrona cuando exige una coordinación entre tareas.

- Granulado: Volumen de procesamiento realizado por cada tarea, en relación con el volumen de comunicación existente entre las tareas. Un programa de granulado muy fino posee tareas que realizan pocas instrucciones y necesitan comunicarse mucho, en cuanto que un programa de granulado grueso posee tareas que ejecutan muchas instrucciones y se comunican poco. Cuando la comunicación entre los procesadores es asíncrona el granulado afecta el tiempo de ejecución de un programa, esto porque programas con granulado fino necesitan sincronización mas frecuente, introduciendo periodos de inactividad de los procesadores y aumentando el tiempo de ejecución.

- Escalabilidad: Este es un sistema definido por un algoritmo paralelo en una maquina paralela y escalable cuando la aceleración obtenida crece proporcionalmente al numero de procesadores utilizados por el hardware. Los factores que contribuyen para la escalabilidad de un sistema son: el hardware, el algoritmo utilizado y el código implementado.

- Balanceo de carga: Distribución de las tareas entre los procesadores, a modo de garantizar la ejecución lo más eficiente posible del programa en paralelo. Este balance puede ser visto de manera estática, antes del inicio de la ejecución o de manera dinámica durante el tiempo de ejecución.

- SPMD (*Single Program Multiple Data*): Modelo de programación bastante utilizado donde todos los procesadores ejecutan un mismo programa sobre diferentes tipos de datos

- Determinismo: Un algoritmo o programa es determinístico si su ejecución con una entrada especifica siempre resulta en una misma salida. Se dice que un programa es no determinístico si existe la posibilidad de obtener salidas diferentes para varias ejecuciones del programa, utilizándose siempre una misma entrada.

## 4.4 Modelos de computación paralela.

En este apartado se presenta una breve clasificación de los modelos de computo paralelo.

**Clasificación según Flynn: SISD, MISD, SIMD y MIMD**

En 1966 Flynn Hatcher [12] publica sus estudios sobre arquitecturas en paralelo, los cuales establecieron la terminología tradicional. Debido a los avances tecnológicos de las

ultimas dos décadas, la relevancia de esta taxonomía ha disminuido, pero la terminología ha sido muy usada para describir las computadoras en paralelo.

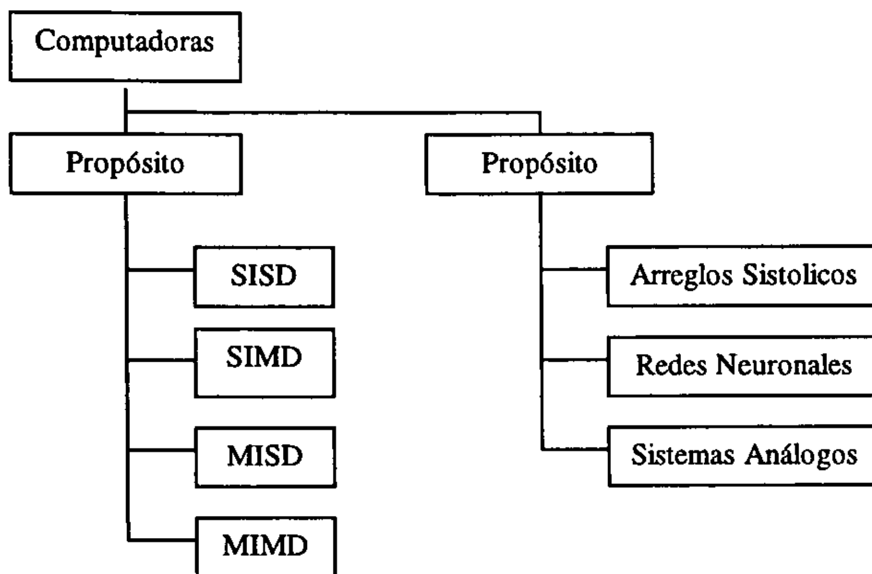



Fig. 4.2 Diagrama de la taxonomia computacional. La computadora de propósito general mas común en la SISD.

En 1996 Flynn Hatcher [12] establece las cuatro clases de arquitecturas de maquinas Fig. 4.2

| SISD | Single Instruction Single Data |
|---|---|
| SIMD | Single Instruction Multiple Data |
| MISD | Multiple Instruction Single Data |
| MIMD | Multiple Instruction Multiple Data |

Haremos referencia a ellos por sus siglas en ingles. A continuación las describimos:

La arquitectura SISD es la arquitectura no paralela, por ejemplo la PC convencional.

La arquitectura SIMD consiste de un numero de procesadores idénticos procesando un ciclo de pasos sincronizados y realizando el proceso sobre diferentes datos.

El modelo MISD es la arquitectura teórica propuesta por Flynn pero que no tuvo uso.

El modelo MIMD cubre un vasto número de esquemas de interconexión de procesadores y arquitecturas. El punto clave es que cada procesador opera de forma independiente de los otros, potencialmente corriendo programas totalmente diferentes. Los procesadores de MIMD se comunican utilizando una red de alta velocidad. La red permite que los procesadores compartan datos y sincronicen calculo. Raro es el problema en paralelo que

no necesita ninguna comunicación o sincronización entre procesadores. Las maquinas MIMD modernas utilizan explícitamente un paradigma de paso de mensajes para comunicar los procesadores. Un procesador especifico envía a un procesador destino, el cual recibe el dato.

## 4.5 Modelos de organización de memoria.

Existen dos tipos básicos de organización de memoria; memoria compartida y memoria distribuida Chandy [2], los cuales se describen a continuación

### Memoria compartida

En las maquinas que utilizan memoria compartida, todos los procesadores pueden acceder a cualquier dirección de memoria, conforme a lo mostrado en la Figura 4.3. La sincronización entre tareas es realizada a través de un control de las operaciones de lectura y escritura ejecutadas por las tareas de memoria compartida. Una ventaja de este tipo de acceso es que la compartición de datos puede ser de manera rápida. Una desventaja es el nivel de escalabilidad del sistema, el limitado numero de caminos existentes entre los procesadores y la memoria.

Una manera de eliminar esta desventaja consiste en proveer a cada procesador con una memoria local, la cual almacena un programa para ser ejecutado y la estructura de datos que no son compartidos entre los procesadores. Las estructuras globales están almacenadas en memoria compartida, conforme a lo ilustrado en la Figura 4.4.

Una extensión del tipo de arquitectura anterior consiste en eliminar totalmente el espacio de memoria que se comparte, conforme a lo mostrado en la Figura 4.5. En ambos casos, estas referencias vista por un procesador en la memoria de otro procesador son controladas por el hardware de interconexión de memoria con los procesadores. Se debe observar que los tiempos de acceso a memorias locales son típicamente menores que los tiempos a memoria remota. Un factor importante para la aplicación correcta de este modelo y el control sobre el acceso de memoria compartida de las tareas. Por ejemplo una tarea no puede alterar un dato mientras otra tarea esta leyendo este mismo dato.

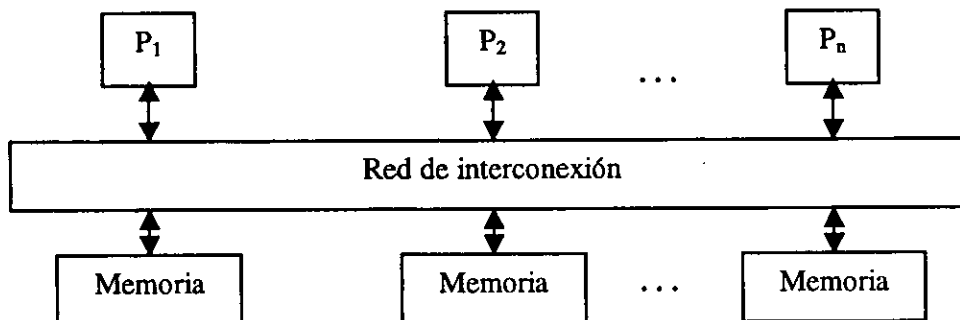



Fig. 4.3 Arquitectura de memoria compartida.

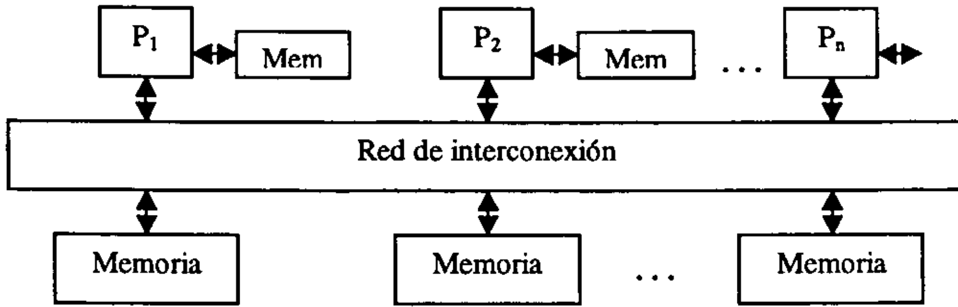



Fig. 4.4 Arquitectura de memoria compartida con memoria local en los procesadores.

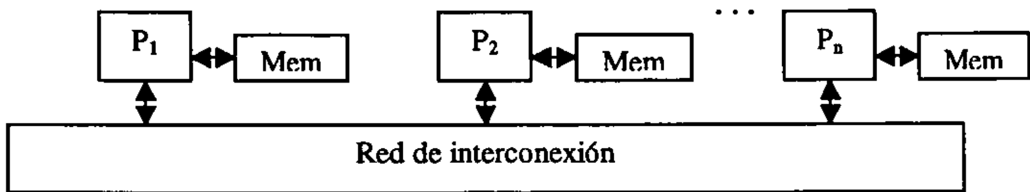



Fig. 4.5 Arquitectura de memoria compartida solamente con memoria local en los procesadores.

**Memoria distribuida**

En maquinas que utilizan memoria distribuida y distribución de memoria física entre los procesadores, cada memoria local solo puede dar acceso al procesador al cual esta asociada. El intercambio entre las tareas se realiza a través de mensajes entre las redes de comunicación, conforme a lo ilustrado en la Figura 4.6

El envío y recibimiento de mensajes entre los procesadores pueden ser vistos en forma con-espera y sin-espera. En el envío con-espera la ejecución de una tarea que genera el envío de un mensaje es interrumpida antes que el mensaje sea recibido por el receptor, en cuanto que en la forma sin-espera inicia el proceso de envío y continua su ejecución sin esperar a confirmar que el mensaje fue recibido por el receptor. En el recibimiento con-espera, una tarea es interrumpida cuando espera la llegada de un determinado mensaje. En la forma sin-espera se verifica si el mensaje dejado esta disponible, en el caso de que no esté, se continua con el procesamiento.

Las ventajas de utilizar programación distribuida son la escalabilidad del sistema, la posibilidad de mayor confianza en el sistema a través del duplicado de datos entre los procesadores y, consecuentemente, tolerancia a fallas y la facilidad de la incorporación de procesadores mas especializados. Las desventajas son la difícil implementación de algunos algoritmos, tales como la descomposición de datos en un arreglo, y la reestructuración de implementaciones ya existente, por ejemplo en aplicaciones secuenciales o en aquellas que utilizan memoria compartida.

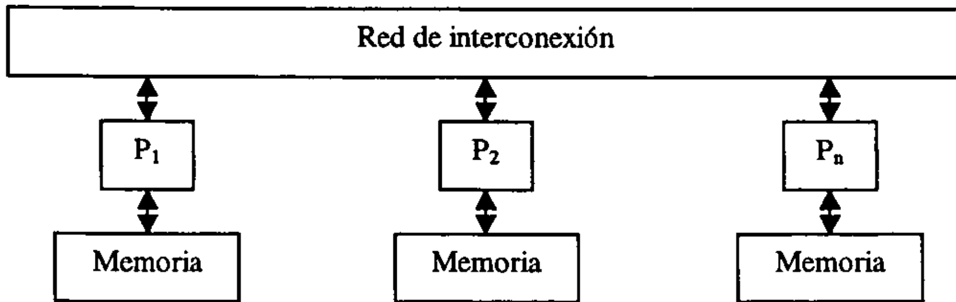



Fig. 4.6 Arquitectura de memoria distribuida.

Debe hacerse notar la semejanza entre la arquitectura de memoria compartida mostrada en la Figura 4.5 y la de memoria distribuida mostrada en la Figura 4.6. La mayor diferencia entre ellas es que, la red de interconexión permite que los procesadores ejecuten operaciones de lectura y de escritura en memorias pertenecientes a otros procesadores. En la segunda el acceso a memoria de otro procesador tiene que ser visto de manera explicita a través del paso de mensajes entre los procesadores.

## 4.6 Métodos de programación.

Como ya comentamos anteriormente, la programación paralela es una disciplina relativamente reciente, donde los patrones todavía no están bien definidos. De esta forma general los métodos de programación paralela, actualmente utilizados, pueden ser clasificados en tres modelos Lewis [16].

- Paralelismo de datos: Los datos son distribuidos entre los procesadores que ejecutan una misma secuencia entre ellos. Un factor importante para la implementación eficiente de este tipo de programación es un direccionamiento de la localización de datos. Para que un programa de este tipo se ejecute eficientemente en maquinas de memoria distribuida, los datos deben estar alojados en la memoria de procesadores próximos a los que van a utilizar estos datos. Este modelo también es conocido como descomposición de dominio.

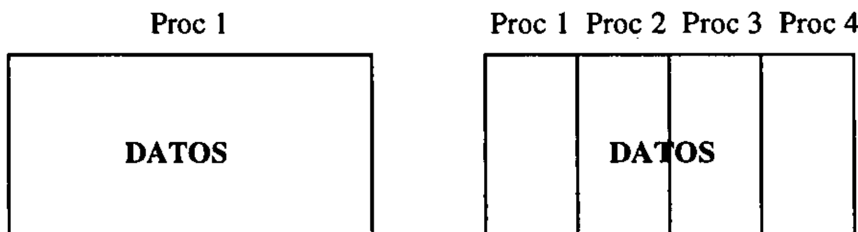



Fig. 4.7 Un conjunto de datos puede ser particionado en varios pedazos

- Paralelismo de tareas: El programa es fraccionado en tareas corporativas. Estas tareas pueden ejecutar diferentes procedimiento entre sí, cuyo procesamiento puede ser visto de manera asíncrona. Para mayor eficiencia de la implementación de los programas debe tomarse en cuenta el direccionamiento de la localización de los datos en un nivel de granularidad de cada tarea. Este modelo también es conocido como descomposición funcional.

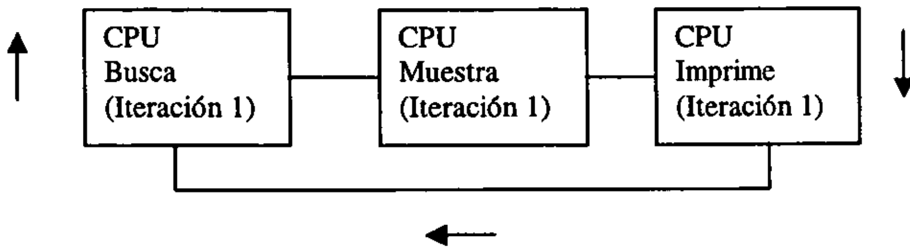



Fig. 4.8 Secuencia del proceso en serie

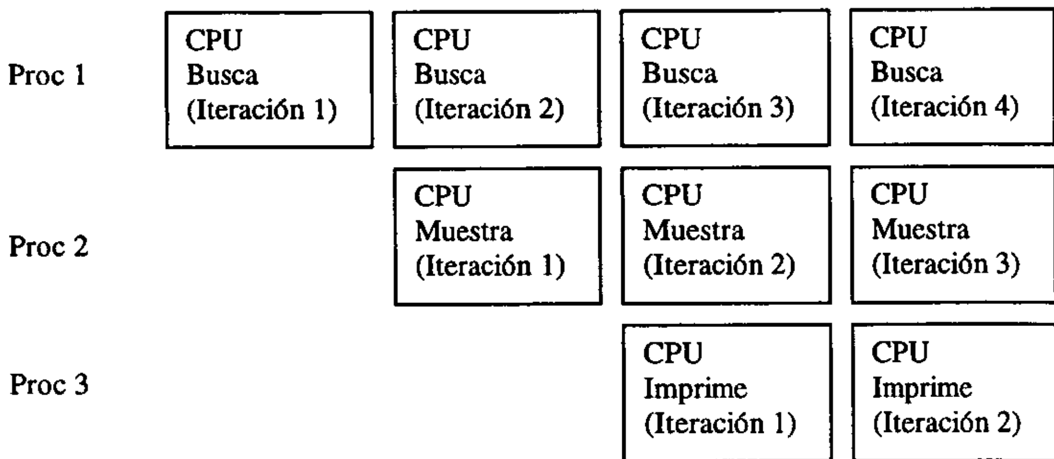



Fig. 4.9 Secuencia de un proceso en el paralelo con 3 procesadores

- Paralelismo de objetos: En este caso, el paralelismo puede ser realizado de diferentes formas. Por ejemplo, (1) una función miembro de un objeto puede llamar una función publica de otro objeto remoto o no, y (2) un tipo abstracto de datos, como un arreglo puede ser implementado de una manera distribuida. Son necesarios los mismos cuidados relacionados a la eficiencia discutidos con anterioridad con relación a la localidad de los datos y a la granularidad de las tareas.

Dependiendo de la aplicación, un modelo podrá ser mas adecuado que otro. En muchas aplicaciones, una combinación de los paradigmas puede ser útil para un mejor desempeño del programa.

## 4.7 Herramientas para ambientes de memoria distribuida.

Como en el caso de programación secuencial, existen varios lenguajes y herramientas para la implementación de programas en paralelo, siendo que cada una de estas representa una mejor opción para una determinada clase de problemas. Estas están divididas en dos clases: lenguajes paralelos y bibliotecas para paso de mensajes. La primera clase abarca los lenguajes que poseen instrucciones especiales para la creación de aplicaciones paralelas por los usuarios. La segunda clase contempla las herramientas que ofrecen funciones de una biblioteca de funciones para que el usuario realice la comunicación entre las tareas de su programa paralelo.

Lenguajes paralelos: La mayor parte de estos lenguajes poseen extensiones vistas en lenguajes secuenciales ya existentes, de modo de tornar transparentes para el usuario las primitivas de comunicación utilizada en el sistema de computación, como una tentativa de facilitar la implementación de programas paralelos. Estos lenguajes poseen un compilador que transforma el programa del usuario y envía el resultado a un compilador de lenguaje secuencia; como ejemplos de lenguaje paralelo pueden citarse a: HPF (High Performance Fortran), Fortran-M, Network Linda, C/C++, Regis, SR, Hatcher [12].

Bibliotecas para paso de mensajes: Una manera simple de agregar nueva funcionalidad a un lenguaje de programación a través de la creación de bibliotecas de software. El usuario realiza llamadas a nuevas rutinas de bibliotecas para procesar las nuevas funciones, sin que el lenguaje original tenga que ser modificado. Debido a la facilidad de implementación antes mencionado, han sido desarrolladas varias herramientas basadas en bibliotecas de paso de mensajes para facilitar el desarrollo de programas paralelos; por ejemplo: PVM (Parallel Virtual Machine), P4, Express, ISIS, MPI, Hatcher [12].

## 4.8 MPI (Message Passing Interface).

MPI no es propiamente un sistema implementado, pero si una propuesta de estandarización para una interfaz de paso de mensajes para maquinas paralelas con memoria distribuida. La idea de estandarizar esta interfaz es la de hacer posible la portabilidad de los programas entre las diferentes maquinas y de facilitar su uso.

El núcleo de MPI esta formado por rutinas para comunicación punto a punto entre pares de proceso. Estas rutinas pueden ser llamadas de forma con-espera o sin-espera. Existen tres modos de comunicación: lectura (ready), en el cual un mensaje puede ser enviado a una operación de recepción correspondiente sin haber sido iniciada; estándar (standard), en la cual un mensaje puede ser enviado independientemente de haber una operación de recepción para él; y sincronizado (synchronous), que funciona de manera semejante al

modo estándar, con la diferencia de que una operación de envío no será considerada terminada hasta que la operación de recepción correspondiente sea iniciada en el proceso destino.

Otras facilidades que ofrecen las rutinas para grupos de procesos, son las que coordinan la comunicación entre los procesos pertenecientes a un mismo grupo. Estas rutinas permiten una transmisión colectiva de datos y otorgan las siguientes facilidades:

- envío de un mismo dato por un participante del grupo para todos los demás (broadcast)
- envío de datos diferentes de un determinado participante de un grupo para todos los demás.
- envío de datos de todos los participantes de un grupo para un determinado participante
- envío de datos de cada participante para todos los demás.
- envío de datos diferentes de cada participante para todos los demás.

# Capítulo5

# Programación del GRASP para el *QAP*

## 5.1 Introducción

A continuación explicaremos cómo desarrollamos el programa para resolver el problema de asignación cuadrática. El listado del programa aparece en el apéndice A.

El lenguaje de programación utilizado para desarrollar el programa fue el *Lenguaje C*, básicamente porque *C* es un lenguaje que permite correr nuestra aplicación en varios ambientes (UNIX, MS-DOS) y equipos, prácticamente sin hacer ninguna modificación, es decir por su *portabilidad*.

Desafortunadamente el tiempo de CPU que consume el programa secuencial de GRASP aumenta a medida que crece el tamaño de la instancia a resolver por lo cual también programamos una versión en paralelo del algoritmo con la finalidad de reducir el tiempo de CPU utilizado.

Los resultados del programa se obtuvieron al correr un programa ejecutable, generado por el compilador *gcc* (v2.7) en una maquina SGI/CRAY ORGIN 2000 localizada en la DGSCA UNAM con 32 procesadores R 10000 a 195 Mhz., con sistema operativo IRIS 6.4, físicamente una maquina de memoria distribuida, pero lógicamente resulta ser una maquina de memoria compartida. Para la versión en paralelo se uso el modelo de programación paralela MIMD.

Se generan *m* semillas para números aleatorios, donde cada procesador toma una semilla. Con el objetivo de que los *m* procesadores compartan datos las siguientes variables se declaran como globales: *n*, optimo, max_iter, distancia y flujo.

## 5.2 Descripción general del programa

Primero definiremos las variables que se van a utilizar en el programa, algunas son de carácter global y otras sólo se utilizan localmente en algunos procedimientos. Las variables globales son las siguientes:

- **mejor_total[32]** mejores costos de cada procesador.
- **perm_total[32][Max]** mejores permutaciones de cada procesador.

- **iter_total[32]** mejores iteraciones de cada procesador.
- **tproc[32]** tiempo de cada procesador.
- **max_iter** máximo de iteraciones GRASP.
- ***n*** dimensión de la matriz.
- **optimo** valor óptimo o mejor conocido.
- **Np** número de procesadores a utilizar.
- **distancia[Max] [Max]** matriz de distancias.
- **flujo[Max][Max]** matriz de flujos.

Las variables que se utilizan localmente en cada procedimiento o función, no se mencionan debido a que estas son índices o variables temporales que son utilizadas como auxiliares para realizar los cálculos.

Ahora describiremos las funciones y procedimientos más importantes y más usados por nuestro programa diseñado.

Dado que una de las componentes del GRASP es la aleatoriedad se diseñaron procedimientos y funciones con esta tarea: y son

*next (unsigned short state[], unsigned short mul0, unsigned short mul1, unsigned short mul2, unsigned short add)* calcula el siguiente número seudoaleatorio.

*mynrand48 (unsigned short state[3], unsigned short mul0, unsigned short mul1, unsigned short mul2, unsigned short add)* regresa un numero seudoaleatorio.

*mylrand48 (unsigned short mul0, unsigned short mul1, unsigned short mul2, unsigned short add, unsigned short in_s[])* regresa un número seudoaleatorio.

*mysrand48* (long seedval, *unsigned short *mul0, unsigned short *mul1, unsigned short *mul2, unsigned short *add, unsigned short in_s[])* crea una nueva secuencia de números aleatorios.

En la construcción de la permutación fue necesaria el diseño de procedimientos y funciones que realizaran la labor de manipular conjuntos tales como:

*Intset (tset*conjunto)* que lleva como parámetro el conjunto a ser inicializado.

*Addset (tset*conjunto, int x)* que lleva como parámetro el conjunto donde va a ser insertado el elemento *x*.

*Inset (tset*conjunto, int x )* verifica si en un conjunto se encuentra el elemento *x*.

*Outset (tset*conjunto, int x)* saca el elemento *x* del conjunto.

Los siguientes procedimientos se utilizaron para manipular las listas generadas durante la ejecución del programa:

*carga_listas (tlista ldist[], tlista lflujo[])* recibe como parámetros la lista de distancias y de flujos que se encuentran en las matrices de distancias y de flujos.

*Sortlflujo (tlist a[], int l, int r)* recibe como parámetros la lista de flujos a ser ordenada en forma decreciente y el límite inferior y superior de la lista.

*Sortldist (tlist a[], int l, int r)* recibe como parámetros la lista de distancias a ser ordenada en forma creciente y el límite inferior y superior de la lista.

*Sortcostos (costos a[], int l, int r)* recibe como parámetros la lista de costos a ser ordenada en forma creciente y los limites inferior y superior de la lista.

*Ordenaij (tlist a[], int max),* recibe como parámetro la lista de flujos o de distancias para ordenarlas en forma creciente con respecto a sus índices en los casos de empates y el numero de elementos de la lista.

*Mul_ldistlflujo (costos listacosto[], tlista ldist[], tlista lflujo[], const int parametro1)* recibe como parámetros las listas de costos, de distancias y de flujos ya ordenadas y multiplica los elementos correspondientes de las listas de distancias y flujos hasta parametro1 y guarda esta lista en *listacosto[].*

Describiremos las principales funciones y procedimientos del programa ya que en ella comienza y termina el proceso iterativo de GRASP.

*init_conjunto (tset *city, tset *fscility, tset coml, tset *comj)* recibe como parametros cuatro conjuntos a city y facility los pone vacios y a coml. y comj les inserta elementos de n elementos.

*Randomiza (const costos listacosto[], parejas perm[], tset *city, tset *facility, tset *coml, tset *comj, const int parametro2, unsigned short mul0, unsigned shotr mul1, unsigned short mul2, unsigned short add, unsigned short in[])* recibe como parámetro la lista de costos , un arreglo de dos campos, cuatro estructuras de conjuntos, parametro2 que sirve para restringir la elección aleatoria de los elementos de la lista de costos y recibe mul0, mul1, mul2, add, in[] para la utilización de la función aleatoria. El elemento elegido aleatoriamente de la lista de costos (LRC) lleva consigo cuatro subíndices identificados por los campos *i, j, k, l*, de donde se desprenden dos parejas (*i, k*) y (*j, l*), es decir las primeras dos asignaciones de la solución buscada. Los elementos *i, j* son sacados del conjunto *comj*, y metidos en el conjunto facility y los elementos *k, l* son sacados del conjunto *coml* e introducidos en el

conjunto *city*, los conjuntos comj y coml contienen los números de 1,2, ... ,*n* y los conjuntos *facility* y *city* están vacíos.

*stage2(parejas perm[Max], tset *city, tset *facility, tset *cpml, tset *comj, unsignied short mul0, unsigned short mul1, unsigned short mul2, unsigned short add, , unsigned short in[])* recibe como parámetros un arreglo con las primeras dos asignaciones, cuatro conjuntos y recibe mul0, mul1, mul2, add, in[] para la utilización de la función aleatoria. El propósito de este procedimiento es terminar de construir la solución obteniendo las *n*-2 restantes asignaciones, con lo cual se tiene la solución completa.

*two_exchange ( parejas perm[], const int costo, unsigned short mul0, unsigned short mul1, unsigned short mul2, unsigned short add, unsigned short in[])* recibe como parámetros un arreglo de parejas, un costo inicial de la solución y recibe mul0, mul1, mul2, add, in[] para la utilización de la función aleatoria. A este procedimiento le es proporcionada una permutación (actual) que se obtuvo de ordenar mediante *sortperm()* las primeras componentes del conjunto de parejas construido en *stage2()* y procede a obtener todos los vecinos de esta permutación con sus respectivos costos de asignación, toma al vecino con menor costo, lo compara con el costo de la permutación actual, si el costo del vecino es menor que el costo de la permutación actual, entonces se actualiza y el proceso se repite hasta que ya no mejore. Esta función regresa la permutación que tiene costo menor y su costo.

*Grasp()*. Algunas variables importantes para este procedimiento son: ru, t_inicial, t_final, timep, t, city, facility, coml, comj, perm[Max], iter_grasp, iter_mejor, mejorp[Max], best, total, band, my_ide, lflujo[maxlista], ldist[maxlista] y listacostos[maxlista].

Obtiene el identificador correspondiente al procesador que esta en uso en la variable my_ide mediante la función m_get_myid(); obtiene el tiempo inicial de ejecución mediante la función *getrusage(RUSAGE_SELF, &ru)*; crea una secuencia de números seudoaleatorios con la semilla correspondiente al procesador llamando al procedimiento *mysrand48()*; crea las listas distancias y de flujos tomando los datos de la matriz compacta de entrada llamando al procedimiento *carga_listas()*; ordena la lista de flujos en forma decreciente mediante el procedimiento *sortlflujo()*; ordena la lista de flujos con respecto a los índices de las filas y columnas de la matriz en caso de empates en forma creciente llamando al procedimiento *ordenaij()*; ordena la lista de distancias en forma creciente mediante el procedimiento *sortldist()* así mismo esta lista es ordenada con respecto a los índices de las filas y las columnas de la matriz de manera creciente en los casos que hay empates nuevamente con el procedimiento *ordenaij()*; calcula la lista de costos mediante el procedimiento *mul_ldistlflujo()*, el cual recibe como parámetros las listas de distancias y flujos ordenadas y *paramero1* el cual restringe la lista de costos, luego es llamado el procedimiento *sortcostos()* que ordena los costos en forma creciente.

*init_conjunto ()* es un procedimiento que inicializa a los conjuntos city, facility, coml y comj; se llama al procedimiento *randomiza ()* en el cual se realizan tres operaciones importantes: La primera es restringir ahora la lista de costos por parametro2, con lo cual se

obtiene la LRC, la segunda operación es elegir aleatoriamente un elemento de la LRC con la función *mylrand48()* y finalmente del elemento elegido se obtienen las primeras dos componentes de la permutación que pretendemos construir determinados por las parejas, ( listacosto[*i*].*i*, listacosto[*i*].*k*) y ( listacosto[*i*].*j*, listacosto[*i*].*l* ).

Con el procedimiento *stage2()* se construye el resto de la permutación, calculando el costo de cada componente no asignada con respecto a las componentes que ya fueron asignadas, estos costos se ordenan en forma creciente mediante el procedimiento *sortldist()*, se ha generado una nueva lista en la cual en general siempre hay empates en los elementos mínimos de los cuales se elige uno aleatoriamente mediante la función *mylrand48()*, estos costos llevan los índices *i*, *k* que son los que determinan la nueva asignación en la permutación en construcción, esto esta en un ciclo que concluye hasta que la permutación esta completa la permutación obtenida en realidad esta en un conjunto de parejas las cuales son ordenadas por el procedimiento *sortperm()* de manera creciente con respecto a la primera componente de cada pareja y las segundas componentes son las que forman la permutación; la permutación obtenida determina una solución del problema de asignación cuadrática cuyo costo es calculado por la función *costo_total ()*.

Con la función *two_exchange ()* se obtienen todas las permutaciones vecinas que también son soluciones factibles del *QAP* con sus respectivos costos de asignación con la función *costo_total ()*, se busca el vecino que tenga el menor costo, en caso de empate este se rompe aleatoriamente mediante la función *mylrand48()*, repetimos la función *two_exchange ()* con la permutación actual hasta que se considere que no se puede encontrar una mejor solución, se guardan la permutación con mejor costo encontrados hasta el momento, en caso que los criterios de paro si no se hayan cumplido se entra en un ciclo y se incrementa el contador de variable *iter_grasp* repitiendo todo el proceso desde el procedimiento *init_conjunto ()*.

Cuando llegamos al final del ciclo se toma el tiempo final mediante la función *getrusage(RUSAGE_SELF,&ru)* y se calcula el tiempo total de ejecución en la variable *timep*; los resultados se guardan en las variables globales en el lugar correspondiente al identificador de procesador con el valor almacenado en la variable *my_ide*.

## 5.3 GRASP en paralelo

Cualquier método de búsqueda local generalmente consume una gran cantidad de tiempo de CPU para obtener buenas soluciones. Esto es debido al intenso calculo de iteraciones. Sin embargo si la búsqueda no puede ser mejorada en un solo procesador , entonces es posible hacer uso de un sistema multiprocesador para acelerar la búsqueda.

Dado cada iteración del GRASP es independiente una de la otras, es posible ejecutar concurrentemente todas las iteraciones. Entonces la paralelización del GRASP es directa.

La versión en paralelo del GRASP particiona el conjunto de todas las posibles iteraciones en Np subconjuntos de aproximadamente el mismo tamaño, y cada partición es evaluada en un procesador. Así cada procesador encuentra una mejor solución, de las cuales se toma la mejor de estas como la solución final.

El diseño del GRASP en paralelo se realizo en una computadora con arquitectura paralela con 32 procesadores, paralelizando las tareas, sin paso de mensajes ni sincronización durante la ejecución del programa.

Se generan Np semillas que producen cada una, una secuencia de números aleatorios. Estas semillas junto con la instancia de datos y el programa son distribuidas a los Np procesadores y cada procesador resuelve independientemente la instancia usando GRASP, así se obtienen Np soluciones de las cuales se toma la mejor solución. El algoritmo en paralelo de muestra en la figura 5.3.1.

Con el objetivo de que todos los procesadores comparatan datos las siguientes variables se declaran como globales: *n*, optimo,max_iter, distancia y flujo.

*main (arg c, char *arg[]).* Recibe como parámetros el nombre del archivo binario de datos; n, dimensión de la matriz; max_iter, número máximo de iteraciones en las que ya no mejora; óptimo, valor óptimo o mejor conocido; Np, número de procesadores a utilizar; nombre del archivo de salida.

Se llama a la subrutina *m_set_procs* para fijar número de procesadores a utilizar; se crea un ciclo para generar Np semillas para números aleatorios para los procesadores; se calculan los parámetros para cortar las listas de distancias, flujos y costos; se calcula el tiempo inicial de ejecución *getrusage(RUSAGE_SELF,&ru).*

Si Np > 1, entonces se llama a una subrutina vacia con *m_fork* en cada uno de los procesadores y se suspenden estos procesos con *m_park_procs*. Se llama al control de planificación para que los procesos se ejecuten al mismo tiempo, con *schedctl(SCHEDMODE, SGS_GANG)*. Se ejecuta el procedimiento GRASP en cada procesador con *m_fork (GRASP)* para evaluar sus correpondientes iteraciones de acuerdo con la semilla aleatoria asignada a tal procesador. Se toma el tiempo final de ejecución y el tiempo total de ejecución es calculado. Con *m_kill_procs ()*, se terminan los procesos creados previamente por *m_fork*.

Los resultados de cada procesador se guardan en el archivo de salida, específicamente la permutación de asignación con costo menor y el número de iteraciones.

**Procedure** GRASP _Paralelo(n, *C*, Np)

| | |
|---|---|
| 1 | MejorSol = $\infty$ |
| 2 | GenerarSemillas($s_1, s_2, \ldots, s_{Np}$) |
| 3 | **Do** ( en paralelo ) $i$ = 1,...,Np; |
| 3 |     GRASP(n,C,$s_i$, $costo_i$, $perm_i$, ); |
| 4 |     If $costo_i$ < MejorSol |
| 5 |         ActualizarMejorSol(costo, MejorSol, perm, MejorPerm); |
| 6 | **end**{Do}; |
| 7 | **return**( MejorSol,MejorPerm) |

Figura 5.3.1. Pseudocódigo generico para GRASP en paralelo para el *QAP*.

# Capítulo 6

# Resultados

## 6.1 Introducción

En esté capítulo se muestran los resultados obtenidos para quince instancias, doce propuestas por Nugent [18] y tres por Skorin-Kapov [29]. Se muestra una tabla de datos para cada una de las tres estructuras vecinales para las cuales se diseño el programa en la tesis. También se muestra tres graficas comparativas de las estructuras vecinales en cuanto al número de iteraciones, el tiempo de ejecución y el porcentaje en que alcanzan un óptimo o mejor valor conocido. Además una tabla de comparación con resultados obtenidos por otros autores para el mismo problema, las mismas instancias y la misma metaheurística. Asimismo se presenta una tabla con los porcentajes del PECM. También las permutaciones de las mejores asignaciones encontradas en nuestro trabajo y finalmente la el factor speedup que muestra la ventaja del diseño en paralelo de un GRASP. Todos los resultados mostrados en este capítulo se obtuvieron restringiendo la lista de candidatos con los parámetros $\alpha = 0.5$ y $\beta = 0.1$ los cuales se determinaron experimentalmente.

## 6.2 Tablas de resultados

Las estadísticas de las tablas 6.2.1, 6.2.2 y 6.2.3 se obtuvieron de realizar 20 corridas del programa secuencial, para cada instancia en cada una de los implementaciones de las estructuras vecinales 2-intercambio, $\lambda$-intercambio y Nstar. Cada tabla contiene:

**Problema**: nombre de la instancia.
***n***: la dimensión de la matriz compacta.
**VOMVC**: valor óptimo o mejor valor conocido.
**MVE**: mejor valor encontrado en este trabajo de tesis.
**CM**: cota mínima.
**PECM**: porcentaje de error con el que se excede el MVE de la cota mínima.
**PSO**: porcentaje en que se obtuvo la solución óptima o el mejor valor conocido.
**PIGRASP**: promedio de iteraciones GRASP en que se obtuvo VOMVC.
**TPCPU**: tiempo promedio de ejecución de CPU

| Problema | $n$ | VOMVC | MVE | CM | PECM | PSO | PIGRASP | TPCPU |
|---|---|---|---|---|---|---|---|---|
| Nug5 | 5 | 25 | 25 | 25 | 0% | 100% | 2 | 0.00 |
| Nug6 | 6 | 43 | 43 | 43 | 0% | 100% | 3 | 0.00 |
| Nug7 | 7 | 74 | 74 | 74 | 0% | 100% | 5 | 0.00 |
| Nug8 | 8 | 107 | 107 | 97 | 9.34% | 100% | 4 | 0.00 |
| Nug12 | 12 | 289 | 289 | 264 | 8.65% | 95% | 65 | 0.23 |
| Nug15 | 15 | 575 | 575 | 542 | 5.73% | 90% | 67 | 0.79 |
| Nug20 | 20 | 1285 | 1285 | 1119 | 12.91% | 100% | 73 | 3.61 |
| Nug21 | 21 | 1219 | 1219 | 1004 | 17.63% | 10% | 485 | 32.97 |
| Nug22 | 22 | 1798 | 1798 | 1417 | 21.19% | 85% | 250 | 24.05 |
| Nug24 | 24 | 1744 | 1744 | 1419 | 18.63% | 70% | 435 | 63.89 |
| Nug25 | 25 | 1872 | 1872 | 1532 | 18.16% | 55% | 420 | 56.65 |
| Nug30 | 30 | 3062 | 3062 | 2886 | 5.75% | 5% | 529 | 224.78 |
| Sko42 | 42 | 7906 | 7918 | 7467 | 5.69 % | 0% | 3823 | 9294.33 |
| Sko64 | 64 | 24249 | 24330 | 22868 | 6.00% | 0% | 256 | 5989.71 |
| Sko81 | 81 | 45499 | 45658 | 43036 | 5.74% | 0% | 324 | 110638.15 |

Tabla 6.2.1. Resumen de los resultados obtenidos con la ejecución del programa secuencial con la estructura vecinal 2-intercambio.

| Problema | $n$ | VOMVC | MVE | CM | PECM | PSO | PIGRASP | TPCPU |
|---|---|---|---|---|---|---|---|---|
| Nug5 | 5 | 25 | 25 | 25 | 0% | 100% | 1 | 0.00 |
| Nug6 | 6 | 43 | 43 | 43 | 0% | 100% | 4 | 0.00 |
| Nug7 | 7 | 74 | 74 | 74 | 0% | 100% | 5 | 0.00 |
| Nug8 | 8 | 107 | 107 | 97 | 9.34% | 100% | 5 | 0.00 |
| Nug12 | 12 | 289 | 289 | 264 | 8.65% | 100% | 55 | 0.81 |
| Nug15 | 15 | 575 | 575 | 542 | 5.73% | 100% | 56 | 2.65 |
| Nug20 | 20 | 1285 | 1285 | 1119 | 12.91% | 100% | 77 | 14.21 |
| Nug21 | 21 | 1219 | 1219 | 1004 | 17.63% | 55% | 359 | 99.87 |
| Nug22 | 22 | 1798 | 1798 | 1417 | 21.19% | 100% | 101 | 34.82 |
| Nug24 | 24 | 1744 | 1744 | 1419 | 18.63% | 95% | 241 | 117.35 |
| Nug25 | 25 | 1872 | 1872 | 1532 | 18.16% | 60% | 339 | 205.3 |
| Nug30 | 30 | 3062 | 3062 | 2886 | 5.75% | 5% | 363 | 561.18 |
| Sko42 | 42 | 7906 | 7918 | 7467 | 5.69% | 0% | 2203 | 20171.16 |
| Sko64 | 64 | 24249 | 24379 | 22868 | 6.20% | 0% | 256 | 4858.48 |
| Sko81 | 81 | 45499 | 45758 | 43036 | 5.95% | 0% | 324 | 21752.87 |

Tabla 6.2.2. Resumen de los resultados obtenidos con la ejecución del programa secuencial con la estructura vecinal λ-intercambio.

| Problema | $n$ | VOMVC | MVE | CM | PECM | PSO | PIGRASP | TPCPU |
|---|---|---|---|---|---|---|---|---|
| Nug5 | 5 | 25 | 25 | 25 | 0% | 100% | 3 | 0.00 |
| Nug6 | 6 | 43 | 43 | 43 | 0% | 100% | 4 | 0.00 |
| Nug7 | 7 | 74 | 74 | 74 | 0% | 100% | 3 | 0.00 |
| Nug8 | 8 | 107 | 107 | 97 | 9.34% | 100% | 4 | 0.00 |
| Nug12 | 12 | 289 | 289 | 264 | 8.65% | 90% | 55 | 0.87 |
| Nug15 | 15 | 575 | 575 | 542 | 5.73% | 100% | 121 | 6.1 |
| Nug20 | 20 | 1285 | 1285 | 1119 | 12.91% | 100% | 15 | 6.59 |
| Nug21 | 21 | 1219 | 1219 | 1004 | 17.63% | 35% | 428 | 117.20 |
| Nug22 | 22 | 1798 | 1798 | 1417 | 21.19% | 100% | 215 | 82.00 |
| Nug24 | 24 | 1744 | 1744 | 1419 | 18.63% | 95% | 408 | 230.14 |
| Nug25 | 25 | 1872 | 1872 | 1532 | 18.16% | 70% | 270 | 217.35 |
| Nug30 | 30 | 3062 | 3062 | 2886 | 5.75% | 10% | 1647 | 2720.32 |
| Sko42 | 42 | 7906 | 7926 | 7467 | 5.79% | 0% | 3263 | 28186.28 |
| Sko64 | 64 | 24249 | 24436 | 22868 | 6.41% | 0% | 256 | 5262.38 |
| Sko81 | 81 | 45499 | 45738 | 43036 | 5.90% | 0% | 324 | 23451.24 |

Tabla 6.2.3. Resumen de los resultados obtenidos con la ejecución del programa secuencial con la estructura vecinal Nstar

Los números en las tablas de la columna VOMVC son los reportados en la literatura como los valores óptimos desde $n = 5$ hasta $n = 25$ y para las dimensiones 30,42,64 y 81 como los mejores conocidos (distintas fuentes coinciden en esto). Los de la columna MVE son los mejores valores encontrados con nuestro programa, además se puede observar que obtuvimos éxito para doce de las quince instancias y para las restantes tres no se alcanzó el mejor valor conocido, sin embargo para estas obtuvimos mejores resultados que otros autores como se vera más adelante; para las instancias cuya dimensión es menor a 42, el criterio de paro utilizado fue el siguiente: si el mejor valor encontrado no mejora después de $n^2$ iteraciones, entonces finaliza y se reporta el mejor valor encontrado; y para las últimas dos instancias el criterio de paro fue un número fijo de $4n$ iteraciones, estos criterios de paro se obtuvieron experimentalmente. Los números en la columna CM se consideraron de la tabla QAPLP Statistics [25], la cual se generó con un algoritmo de puntos interiores para programación lineal. Los de la columna PECM se obtienen mediante la formula PECM = [(MVE-CM)/MVE]*100%. La columna PSO tiene los porcentajes del número de veces que se obtuvo el óptimo o mejor valor conocido. observe que en la tabla 6.2.1 en todas las corridas se obtuvo un óptimo para las instancias de dimensión 5,6,7,8 y 20, igualmente en la tabla 6.2.2 siempre se alcanzó un óptimo para $n = 5,6,7,8,12,15,20$ y 22, asimismo la tabla 6.2.3 muestra que un óptimo siempre fue alcanzado para las dimensiones 5,6,7,8,15,20 y 22.Por otro lado nunca se alcanzo el mejor valor conocido para las instancias de dimensión 42,64 y 81. La columna de números en el apartado PIGRASP representan el promedio de iteraciones de las corridas que alcanzaron un óptimo, el mejor valor conocido para los casos desde $n =5$ hasta $n = 30$, pero para $n = 42,64$ y 81 solo se escribió el número de iteraciones de la corrida que tenía el mejor valor. Finalmente la columna correspondiente a TPCPU contiene el promedio del tiempo de ejecución de las corridas que alcanzaron el mejor valor de la función objetivo.

## 6.3 Comparación de los resultados

### 6.3.1 Graficas de comparación

Las graficas de las figuras 6.3.1.1, 6.3.1.2 y 6.3.1.3 muestran los comportamientos de los datos de las tablas 6.2.1, 6.2.2 y 6.2.3, en cuanto al PIGRASP, TPCPU y POS respectivamente, tomando en cuenta solo las dimensiones de las instancias en que se alcanza un valor óptimo.

La figura 6.3.1.1 muestra que el programa con la estructura vecinal λ-intercambio produce mejores resultados que las otras implementaciones en términos del promedio de iteraciones por ejemplo en $n = 30$, se ve muy clara la mejoría aunque fue superado en algunos casos como en $n = 22$ y 25 por el programa con la estructura Nstar.

La figura 6.3.1.2 muestra que el programa con la estructura vecinal 2-intercambio produce mejores resultados que las implementaciones de, λ-intercambio y Nstar en términos del promedio de tiempo de ejecución para todos los casos considerados. Se observa que los resultados colocan a λ-intercambio como la segunda opción en TPCPU.

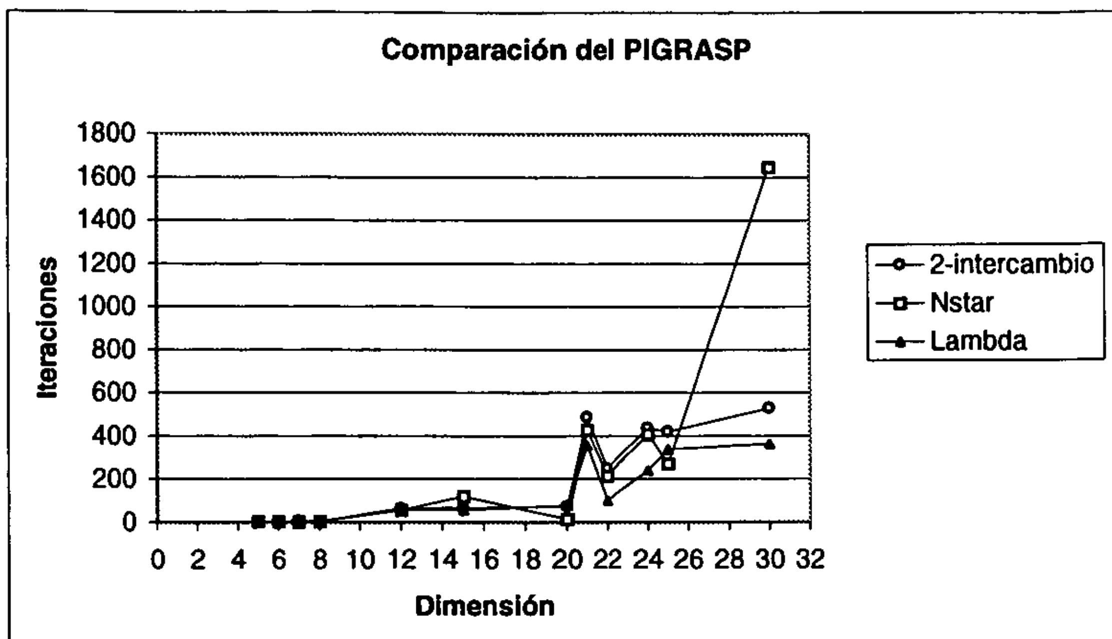



Figura 6.3.1.1. Comportamiento de los resultados del programa secuencial para las estructuras vecinales 2-intercambio, λ-intercambio y Nstar, en cuanto PIGRASP.

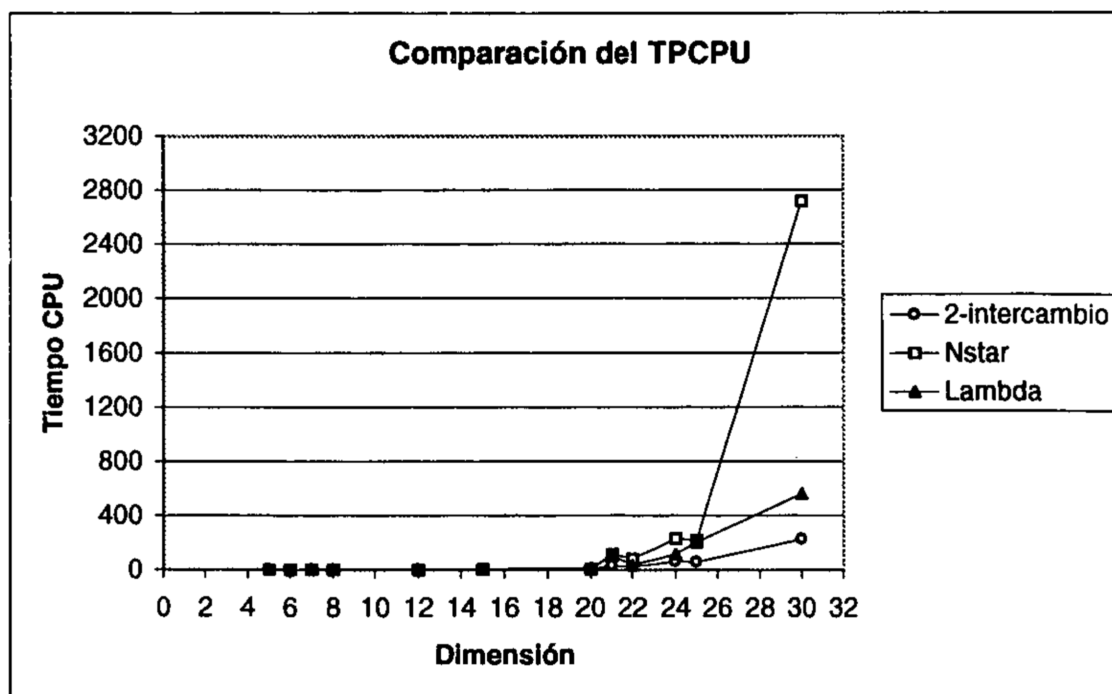



Figura 6.3.1.2. Comportamiento de los resultados del programa secuencial para las estructuras vecinales 2-intercambio, λ-intercambio y Nstar, en cuanto TPCPU.

La figura 6.3.1.3 indica que el programa para 2- intercambio pierde poder a medida que n crece, en cambio λ-intercambio y Nstar obtienen mejores resultados. λ-intercambio supera a 2-intercambio y Nstar en los casos $n$ = 12 y 21, pero Nstar se impone para $n$ = 25 y 30 que son los casos de dimensiones mayores.

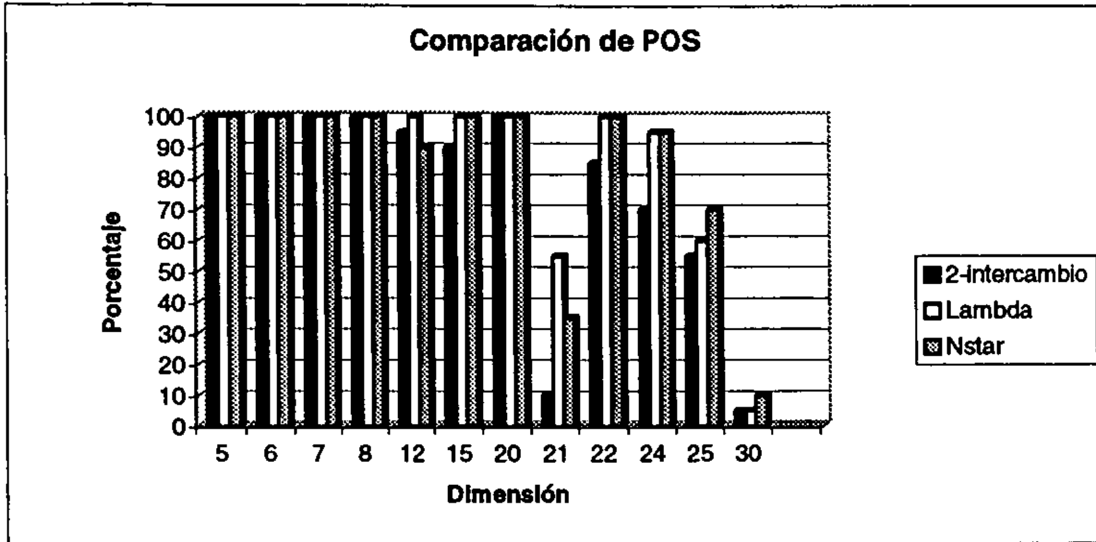



Figura 6.3.1.3. Comportamiento de los resultados del programa secuencial para las estructuras vecinales 2-intercambio, λ-intercambio y Nstar, en cuanto al POS.

## 6.3.2 Tablas de comparación

En esta sección comparamos nuestros resultados con los obtenidos por otros autores que utilizaron las mismas instancias para resolver el *QAP*. En la tabla 6.3.2.1 se muestran los resultados obtenidos por Li [17], Resende [26] y por nuestro diseño, en todos los casos utilizando la misma metaheurística. El criterio de paro usado por Li en cuanto al número de iteraciones fue de 100,000 sólo para las instancias de dimensión $n$ = 5,6,7,8,12,15,20 y 30, Resende realizo corridas para un número fijo de iteraciones y reporto el mejor valor encontrado para matrices de tamaño $n$ = 8, 12,15, 20, 30, 42, 64 y 81. Cabe aclarar que estos autores reportan resultados para otras instancias y nosotros seleccionamos solo aquellas que coinciden con nuestro trabajo de tesis. No localizamos artículos que reportaran resultados para $n$ = 21, 22, 24 y 25, por lo cual no se pudo hacer las comparaciones correspondientes a estos casos. Los conceptos que se comparan en la tabla son: MVE, mejor valor encontrado; ITER, iteraciones reportadas y cuando aparezca *nd* significa que el dato no esta disponible.

Los resultados que aparecen en las columnas correspondientes a GRASP Tesis corresponden a una selección de las tablas 6.2.1, 62.2, y 6.2.3, tomando en cuenta que si alcanzo un óptimo, entonces escribimos el que tuviera menor tiempo de ejecución y el menor número de iteraciones, en otro caso el mejor valor encontrado con su respectivo número de iteraciones y tiempo de ejecución.

| Problema | GRASP Li[17] | | | GRASP M. Resende[26] | | | GRASP Tesis | | |
|---|---|---|---|---|---|---|---|---|---|
| | MVE | ITER | T_CPU | MVE | ITER | T_CPU | MVE | ITER | T_CPU |
| Nug5 | 26 | 1 | 0.00 | *nd* | *nd* | *nd* | 25 | 1 | 0.00 |
| Nug6 | 43 | 1 | 0.00 | *nd* | *nd* | *nd* | 43 | 3 | 0.00 |
| Nug7 | 74 | 1 | 0.00 | *nd* | *nd* | *nd* | 74 | 3 | 0.00 |
| Nug8 | 107 | 4 | 0.01 | 107 | 32 | *nd* | 107 | 4 | 0.00 |
| Nug12 | 289 | 100 | 0.64 | 289 | 32 | *nd* | 289 | 65 | 0.23 |
| Nug15 | 575 | 20 | 0.27 | 575 | 32 | *nd* | 575 | 67 | 0.79 |
| Nug20 | 1285 | 736 | 30.12 | 1285 | 2048 | *nd* | 1285 | 77 | 3.61 |
| Nug21 | *nd* | *nd* | *nd* | *nd* | *nd* | *nd* | 1219 | 485 | 32.97 |
| Nug22 | *nd* | *nd* | *nd* | *nd* | *nd* | *nd* | 1798 | 250 | 24.05 |
| Nug24 | *nd* | *nd* | *nd* | *nd* | *nd* | *nd* | 1744 | 435 | 63.89 |
| Nug25 | *nd* | *nd* | *nd* | *nd* | *nd* | *nd* | 1872 | 420 | 56.65 |
| Nug30 | 3062 | 79861 | 14406.48 | 3068 | 2048 | *nd* | 3062 | 529 | 224.78 |
| Sko42 | *nd* | *nd* | *nd* | 7944 | 2048 | *nd* | 7918 | 3823 | 9294.33 |
| Sko64 | *nd* | *nd* | *nd* | 24395 | 2048 | *nd* | 24330 | 256 | 5989.71 |
| Sko81 | *nd* | *nd* | *nd* | 45774 | 2048 | *nd* | 45658 | 324 | 110658.15 |

Tabla 6.3.2.1. Muestra los resultados obtenidos por diferentes autores para el *QAP* usando GRASP.

De los resultados de la tabla 6.3.2.1 se observa la mejora que aportamos en los casos de dimensión $n$ = 20, 30, 42, 64 y 81, por ejemplo para $n$ = 30 Li[17] alcanza el óptimo en 79861 iteraciones y Resende[26] no alcanza el óptimo en 2048 iteraciones pero con nuestra implementación no solo alcanzamos el óptimo sino que mejoramos en el número de iteraciones. Aunque se exhiben los tiempos de CPU , estos no son comparables pues fueron obtenidos en diferentes plataformas de computo.

| PECM | | | | |
|---|---|---|---|---|
| problema | Skorin-Kapov[2] | 2-intercambio | λ-intercambio | Nstar |
| Sko42 | 5.56% | 5.69% | 5.69% | 5.79% |
| Sko64 | 5.70% | 6.0% | 6.20% | 6.41% |
| Sko81 | 5.41% | 5.74% | 5.95% | 5.90% |

Tabla 6.3.2.2. Resumen del PECM.

En la tabla 6.3.2.2 se exhibe el PECM para las dimensiones de las instancias $n$ = 42, 64 y 81 para los cuales no pudimos alcanzar el mejor valor conocido con nuestro programa para ninguna de las estructuras vecinales que implementamos. Los PECM de la columna de Skorin–Kapov fueron tomados de la pagina de Internet QAPLIB y fueron calculadas con respecto al los mejores valores conocidos, es decir este investigador si alcanza estos valores, para las instancias en cuestión, pero no con GRASP. Según la tabla anterior nuestras aproximaciones propuestas no exceden de las mejores ni en el 1%, de error con respecto a la cota mínima.

## 6.4 Permutaciones de mejor asignación

La tabla 6.4.1 contiene las permutaciones de mejor asignación obtenidas con nuestra implementación de GRASP para el *QAP*, para las instancias de dimensión $n$ = 5, 6, 7, 8, 12, 15, 20, 21, 22, 24, 25, 30, 42, 64 y 81.

| Problema | Permutación de mejor asignación |
|---|---|
| Nug5 | 3,4,5,1,2 |
| Nug6 | 6,5,4,3,2,1 |
| Nug7 | 1,2,5,3,4,7,6 |
| Nug8 | 3,4,8,2,1,5,6,7 |
| Nug12 | 5,9,1,8,12,11,3,7,2,10,6,4 |
| Nug15 | 1,2,7,6,14,13,9,4,5,11,10,15,3,8,12 |
| Nug20 | 19,7,4,6,17,20,18,14,5,3,9,8,15,2,12,10,16,1,11,13,10 |
| Nug21 | 3,16,19,21,15,9,4,5,18,10,12,2,17,14,8,11,1,13,6,7,20 |
| Nug22 | 16,22,17,3,11,9,18,14,20,19,6,8,10,2,1,7,12,4,15,13,21,5 |
| Nug24 | 7,6,17,23,5,3,9,20,10,8,21,2,4,18,13,11,19,16,14,24,12,15,22,1 |
| Nug25 | 24,4,13,11,5,18,17,14,19,22,10,6,21,12,20,8,1,3,23,15,7,25,16,2,9 |
| Nug30 | 17,26,3,7,30,28,15,16,21,22,10,29,27,2,6,9,18,5,14,1,25,11,12,23,13, 24,4,19,20,8 |
| Sko42 | 3,27 2,20,23,1,38,8,18,40,39,32,42,19,16,6,37,24,33,34,10,14,7,4,22,25,35,28,29, 12,41,17,21,31,26,5,11,9,13,15,36,30 |
| Sko64 | 10,15,61,38,43,46,20,16,9,24,56,60,22,25,64,28,13,17,40,34,29,27,19,35,6,14,58,54, 33,51,7,48,4,62,1,57,3,39,52,50,63,30,53,12,18,41,47,42,31,5,44,26,32,55,36,2,8,11, 21,49,59,45,23,37 |
| Sko81 | 25,21,78,46,16,70,33,63,80,4,62,39,32,67,60,20,71,30,40,38,69,18,55,23,75,57,35, 6,17,73,43,59,66,56,22,7,47,49,58,2,51,52,15,41,26,36,1,72,50,79,48,53,5,3,9,54, 45,31,14,37,77,34,29,28,74,81,64,44,65,61,8,27,24,76,13,19,11,42,12,10 |

Tabla 6.4.1. Permutaciones encontradas por GRASP

## 6.5 Resultados del algoritmo en paralelo

En esta sección se muestran los resultados del diseño en paralelo para el programa con la estructura vecinal 2-intercambio en un modelo MIMD de computación paralela en una maquina de 32 procesadores. La versión en paralelo fue probada para para las instancias Nug12, Nug20 y sko42 (apéndice B) y para 1,2,4,8,16 y 28 procesadores.

Las correspondientes curvas de aceleración (speedup) del algoritmo en paralelo para las instancias $n$ = 12, 20 y 42 son descritas en la figura 6.5.1, como se ve en la figura la diferencia entre la velocidad para $n$ = 20 y 42 no es significativa cuando el número de procesadores es menor que 8, sin embargo cuando el número de procesadores crece la aceleración decrece cuando el tamaño de la instancia disminuye, esto sucede debido a que al aumentar el número de procesadores la carga de iteraciones se reduce para cada procesador.

Así para $n$ = 42 la aceleración se acerca al caso ideal, mientras que para $n$ = 12 y 20 la eficiencia es menor que la ideal. Por ejemplo cuando se utilizaron 28 procesadores, se obtuvo respectivamente una aceleración de 20.68 y 27.22, para las instancias de 20 y 42.

Esto significa que el uso de 10352.55 segundos de CPU del programa secuencial para la instancia $n$ = 42 distribuye el esfuerzo para cada procesador en 649.40 segundos de CPU en promedio en paralelo para obtener una permutación cercana a un mínimo global.

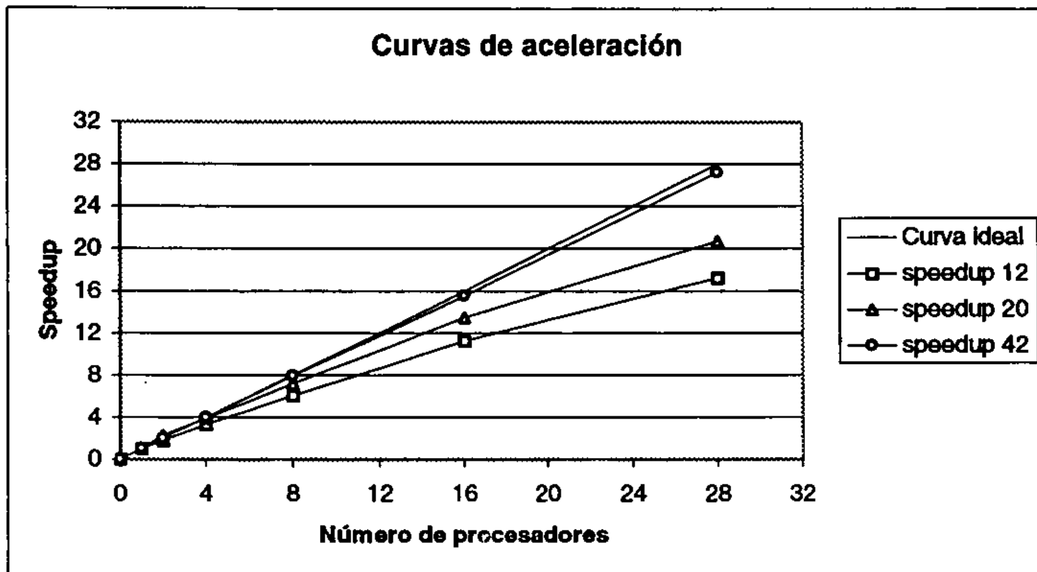



Figura 6.5.1 Curva del Speedup para 1,2,4,8,12,16, y 28 procesadores.

# Capítulo 7

## Conclusiones

En esta tesis se describieron los aspectos de una implementación de GRASP para resolver el problema de asignación cuadrática. El algoritmo fue probado para un conjunto de instancias de prueba conocidas en la literatura con dimensiones 5, 6, 7, 8, 12, 15, 20, 21, 22, 24, 25, 30, 42, 64, y 81 estas matrices se encuentran en el apéndice B. También se presenta una versión en paralelo cuyos resultados nos muestran la bondad del diseño en paralelo para distribuir las cargas en los procesadores y reducir los tiempos de ejecución.

Se incluyeron en la implementación de GRASP para el QAP tres diferentes estructuras vecinales: 2-intercambio, $\lambda$-intercambio y Nstar implementadas en la fase de postprocesamiento de la metaheurística GRASP. Los resultados obtenidos por estas implican que la primera ofrece soluciones en menor tiempo de ejecución, sin embargo es superada en su aproximación a los mejores valores conocidos por las ultimas dos estructuras a medida que la dimensión del problema crece. La estructura Nstar es la que finalmente mostró mejores resultados en cuanto al número de veces que se aproxima a mejor valor conocido.

De gran importancia ha sido la introducción de las matrices de distancias y flujos como una matriz compacta para lograr mejores resultados ya que esto incide directamente en el tamaño de la lista restringida de candidatos (LRC) sin afectar la diversidad de las soluciones construidas. Otro elemento no menos importante es la selección al azar en los casos de empate. La bondad de esto se ve en la taba 6.3.2.1.

En la tabla 6.3.2.2 del capítulo 6 se comparan nuestros resultados del PECM con los obtenidos por Skorin-Kapov mediante metaheurístcas más complejas y sofisticadas. Estos resultados muestran que GRASP es una herramienta poderosa para la construcción soluciones de muy alta calidad.

Existen estructuras vecinales mas elaboradas las cuales aún cuando requieren un tiempo adicional para ser calculadas pueden ser computacionalmente atractivas tal es el caso de 3-intercambio.

Aunque existen muchos aspectos por explorar en GRASP, la formación de híbridos de GRASP con otras heurísticas es un campo con mucho futuro ya que la naturaleza constructiva de GRASP permite sin conflicto alguno que los diferentes métodos de búsqueda se puedan mezclar.

Finalmente GRASP puede ser adaptado para resolver casos especiales de *QAP*, por ejemplo el problema del agente viajero.

# Apéndice A

# Programa

```
/* GRASP*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <task.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/prctl.h>
#include <sys/schedctl.h>
#include <string.h>

#define maxlista   3400                //maximo de elementos en listas
#define Max        83                  //maximo de elementos en matriz
#define true       1
#define false      0

// constantes simbolicas para numeros seudoaleatorios

#define RAND48_MULT0            (0xe66d)
#define RAND48_MULT1            (0xdeec)
#define RAND48_MULT2            (0x0005)
#define RAND48_ADD              (0x000b)

struct listnode{
        int valor,i,j;
        };

typedef struct listnode LISTNODE;
typedef LISTNODE tlista;

typedef struct tindice{
        int i,j;
        int valor;
        };
typedef struct tindice INDICE;
typedef INDICE indices;

struct tparejas{
          int j,l;
          };
typedef struct tparejas PAREJAS;
typedef PAREJAS parejas;

struct lcosto{
        int valor,i,j,k,l;
        };
typedef struct lcosto lista;
typedef lista costos;

struct tcset{
        int set[Max],aux;
        };
typedef struct tcset tset;
```

```
//variables globales visibles para todos los procesadores //////

unsigned int            seed[32];                  //semillas para cada procesador
int                     mejor_total[32];           //mejores iteraciones de cada procesador
int                     perm_total[32][Max];       //mejores permutaciones de cada procesador
int                     iter_total[32];            //mejores iteraciones para cada procesador
float                   tproc[32];                 //mejores tiempos para cada procesador
int       max_iter,                                //maximo de iteraciones GRASP
          n,                                       //dimension de la matriz
          optimo;                                  //funcion objetivo
int       Np;                                      //numero de procesadores
int       retval;
int       distancia[Max][Max],flujo[Max][Max];     //matrices de datos
int       res,parametro1,parametro2;               //parametros para recortar las listas

/* calcula el siguiente numero seudoaleatorio*/
static void next(unsigned short state[],unsigned short mul0,unsigned short mul1,
                                   unsigned short mul2,unsigned short add)
{
  unsigned short new_state[3];
  unsigned long tmp;

  tmp = state[0] * mul0 + add;
  new_state[0] = (unsigned short)(tmp  & 0xffff);

  tmp = (tmp >> 8*sizeof(unsigned short))
          + state[0] * mul1
          + state[1] * mul0;
  new_state[1] = (unsigned short)(tmp & 0xffff);

  tmp = (tmp >> 8*sizeof(unsigned short))
          + state[0] * mul2
          + state[1] * mul1
          + state[2] * mul0;
  new_state[2] = (unsigned short)(tmp & 0xffff);

  memcpy(state, new_state, 3*sizeof(unsigned short));
}

/*regresa un numero seudoaleatorio */
unsigned long mynrand48(unsigned short state[3],unsigned short mul0,unsigned short mul1,
                                   unsigned short mul2,unsigned short add)
{

  next(state,mul0,mul1,mul2,add);
  return( ((unsigned long)state[2]) * 0x8000
         + ( ((unsigned long)state[1]) >> 1 )
          );

}

/*regresa un numero seudoaleatorio */
unsigned long mylrand48(unsigned short mul0,unsigned short mul1,
                                   unsigned short mul2,unsigned short add,unsigned short in_s[])
{
  return(mynrand48(in_s,mul0,mul1,mul2,add));
}

/* crea una nueva secuencia de numeros seudoaleatorios*/
```

```
void mysrand48(long seedval,unsigned short *mul0,unsigned short *mul1,
                                        unsigned short *mul2,unsigned short *add,unsigned short in_s[])
{

 /* Restore default multipliers and additiver. */
 *mul0 = RAND48_MULT0;
 *mul1 = RAND48_MULT1;
 *mul2 = RAND48_MULT2;
 *add = RAND48_ADD;

 /* Setup the new state. */
 in_s[0] = 0x330e;
 in_s[1] = (seedval & 0xffff);
 in_s[2] = ( (seedval >> 16) & 0xffff);

}

/*intercambia dos numeros*/
void cambia(int *a,int *b)
{
        int aux;

        aux=*a;
        *a=*b;
        *b=aux;
}

/* inicializa un conjunto*/
void initset(tset *conjunto)
{
        int i;

        for (i=1;i<conjunto->aux;i++)
                conjunto->set[i]=0;
        conjunto->aux=1;
}

/*inserta un elemento a un conjunto*/
void addset(tset *conjunto,int x)
{
        conjunto->set[conjunto->aux]=x;
                conjunto->aux++;
}

/* verifica si un elemento esta en un conjunto*/
int inset(tset *conjunto,int x)
{
        int i;

        for (i=1;i<conjunto->aux;i++)
                if (conjunto->set[i]==x)
                        return 1;
        return 0;
}

/* saca un elemento de un conjunto*/
void outset(tset *conjunto,int x)
{
        int boo=false,i,j,aux[Max];

        for (i=1;i<conjunto->aux;i++)
                        if (conjunto->set[i]==x)
```

```
                        boo=true;
            if (boo==true)
            {
                    for (i=1,j=1;i<conjunto->aux;i++)
                            if (conjunto->set[i]!=x)
                            {
                                    aux[j]=conjunto->set[i];
                                    j++;
                            }
                            conjunto->aux--;
                            for (i=1;i<conjunto->aux;i++)
                                    conjunto->set[i]=aux[i];
            }
}

/* Intercambia dos registros de lista*/
void cambiaij(tlista *a,tlista *b)
{
        tlista aux;

        aux.valor=a->valor;
        aux.i=a->i;
        aux.j=a->j;
        a->valor=b->valor;
        a->i=b->i;
        a->j=b->j;
        b->valor=aux.valor;
        b->i=aux.i;
        b->j=aux.j;
}

/* ordena una lista de menor a mayor tomando como referencia los indice
        i, j de la matriz */
void ordenaij(tlista a[],int max)
{
        int j,i;

        for(j=1;j<=max;j++)
                for (i=1;i<=max;i++)
                if (i!=max)
                        if (a[i].valor==a[i+1].valor)
                          if (a[i].i==a[i+1].i)
                          {
                                        if (a[i].j>a[i+1].j)
                                                cambiaij(&a[i],&a[i+1]);
                          }
                          else
                                  if (a[i].i>a[i+1].i)
                                          cambiaij(&a[i],&a[i+1]);
}

/* Ordena la lista de distancias de menor a mayor tomando como referencia
        el elemento i, j de la matriz */
void sortldist(tlista a[],int l,int r)
{
        int aux1,aux2,aux3,aux4,i,j,x,y;

        i=l;
        j=r;
        x=a[(l+r)/2].valor;
        do {
                while (a[i].valor < x) i++;
```

```
                while (x < a[j].valor) j--;
                if (i<=j)
                {
                        y=a[i].valor;
                        aux1=a[i].i;
                        aux2=a[i].j;
                        a[i].valor=a[j].valor;
                        a[i].i=a[j].i;
                        a[i].j=a[j].j;
                        a[j].valor=y;
                        a[j].i=aux1;
                        a[j].j=aux2;
                        i++;
                        j--;
                }
        }while (i<=j);
        if (l<j) sortldist(a,l,j);
        if (i<r) sortldist(a,i,r);
}

/* Ordena la lista de flujos de mayor a menor tomando como referencia
        el elemento i, j de la matriz */
void sortlflujo(tlista a[],int l,int r)
{
        int aux1,aux2,aux3,aux4,i,j,x,y;

        i=l;
        j=r;
        x=a[(l+r)/2].valor;
        do {
                while (a[i].valor > x) i++;
                while (x > a[j].valor) j--;
                if (i<=j)
                {
                        y=a[i].valor;
                        aux1=a[i].i;
                        aux2=a[i].j;
                        a[i].valor=a[j].valor;
                        a[i].i=a[j].i;
                        a[i].j=a[j].j;
                        a[j].valor=y;
                        a[j].i=aux1;
                        a[j].j=aux2;
                        i++;
                        j--;
                }
        }while (i<=j);
        if (l<j) sortlflujo(a,l,j);
        if (i<r) sortlflujo(a,i,r);
}

/* ordena la lista de costos de menor a mayor */
void sortcostos(costos a[],int l,int r)
{
        int aux1,aux2,aux3,aux4,i,j,x,y;

        i=l;
        j=r;
        x=a[(l+r)/2].valor;
        do {
                while (a[i].valor < x) i++;
                while (x < a[j].valor) j--;
```

```
		if (i<=j)
		{
			y=a[i].valor;
			aux1=a[i].i;
			aux2=a[i].j;
			aux3=a[i].k;
			aux4=a[i].l;
			a[i].valor=a[j].valor;
			a[i].i=a[j].i;
			a[i].j=a[j].j;
			a[i].k=a[j].k;
			a[i].l=a[j].l;
			a[j].valor=y;
			a[j].i=aux1;
			a[j].j=aux2;
			a[j].k=aux3;
			a[j].l=aux4;
			i++;
			j--;
		}
	}while (i<=j);
	if (l<j) sortcostos(a,l,j);
	if (i<r) sortcostos(a,i,r);
}

/* calcula la lista de costos multiplicando los flujos grandes y las distancias peque¤as */
void mul_ldistlflujo(costos listacosto[],tlista ldist[],tlista lflujo[],const int parametro1)
{
	int i=1;

	while(i<=parametro1)
	{
		listacosto[i].valor=lflujo[i].valor*ldist[i].valor;
		listacosto[i].i=lflujo[i].i;
		listacosto[i].j=lflujo[i].j;
		listacosto[i].k=ldist[i].i;
		listacosto[i].l=ldist[i].j;
		i++;
	}
	sortcostos(listacosto,1,parametro1);
}

/* Toma un elemento aleatoriamente de la lista de costos
   con indices i, j, k, l y forma los primeros 2 elementos de la permutacion
   (i,k) (k,l) y los saca de sus respectivos conjuntos	*/
void randomiza(const costos listacosto[],parejas perm[],tset *city,tset *facility,tset *coml,tset *comj,const int parametro2,
				unsigned short mul0,unsigned short mul1,
				unsigned short mul2,unsigned short add,unsigned short in[])
{
	int i;

	do
	{
		i=(int)mylrand48(mul0,mul1,mul2,add,in)%(parametro2)+1;		//toma un elemento
										//aleatoriamente
	}
	while (i<=1&&i>=parametro2+1);

	// forma los primeros dos elementos de la permutacion
	perm[1].j=listacosto[i].i;
	perm[1].l=listacosto[i].k;
	perm[2].j=listacosto[i].j;
```

```
        perm[2].l=listacosto[i].l;

        // saca los elementos asignados de los conjuntos comj y coml
        outset(comj,listacosto[i].i);
        outset(comj,listacosto[i].j);
        outset(coml,listacosto[i].k);
        outset(coml,listacosto[i].l);

        // mete los elementos asignados a los conjuntos city y facility
        addset(facility,listacosto[i].i);
        addset(facility,listacosto[i].j);
        addset(city,listacosto[i].k);
        addset(city,listacosto[i].l);
}

/*funcion que calcula el costo de una permutacion*/
int costo_total(const parejas g[])
{
        int i,j,suma;
        int t=0;
        suma=0;
        for (i=1;i<=n;i++)
        {
                for (j=1;j<=n;j++)
                {
                        suma=suma+(flujo[i][j])*(distancia[g[i].l][g[j].l]);
                }
                t+=suma;
                suma=0;
        }
        return t / 2;
}

/* Funcion que devuelve el numero de elementos que tienen el mismo costo*/
int desempate(int num, indices arr[])
{
        int i=0,j=0;
        while (num==arr[i+1].valor)
        {
                i++;
                j++;
        }
        return j;
}

/* Construye la permutacion*/
void stage2(parejas perm[Max], tset *city,tset *facility,tset *coml,tset *comj,
                        unsigned short mul0,unsigned short mul1,
                                unsigned short mul2,unsigned short add,unsigned short in[])

{
        int num,aux,suma;
        int i,k,j,rep;
        int m,s;
        indices gamma[250000];
        num=2;
        for (rep=3;rep<=n;rep++)
        {
                m=0;
                aux=0;
                for (i=1;i<=n;i++)
                {
```

```
                suma=0;
                for (k=1; k<=n;k++)
                        if (inset(comj,i)&&inset(coml,k))
                        {
                                for (j=1;j<=num;j++)
                                        suma=suma+((flujo[i][perm[j].j])*(distancia[k][perm[j].l]));
                                aux++;
                                gamma[aux].valor=suma;
                                gamma[aux].i=i;
                                gamma[aux].j=k;
                                m++;
                                suma=0;
                        }
        }

        //ordena el arreglo gamma
        sortldist(gamma,1,m);

        //toma un elemento aleatoriamente
        s=(int)mylrand48(mul0,mul1,mul2,add,in)%(desempate(gamma[1].valor,gamma))+1;
        num++;

        //asigna el elemento a la permutacion
        perm[num].j=gamma[s].i;
        perm[num].l=gamma[s].j;

        //inserta el elemento asignado a los conjuntos facility y city
        addset(facility,gamma[s].i);
        addset(city,gamma[s].j);

        //saca el elemento asignado de los conjuntos coml y comj
        outset(coml,gamma[s].j);
        outset(comj,gamma[s].i);
    }
}

/* calcula los vecinos de la permutacion y sus costos y
devuelve la mejor permutacion y el mejor costo encontrado.
Si algun vecino tiene un costo igual al mejor encontrado, se decide el
movimiento de cambio con un volado*/
int two_exchange(parejas perm[], const int costo,
                        unsigned short mul0,unsigned short mul1,
                        unsigned short mul2,unsigned short add,unsigned short in[])
{
 int moneda;
 int i,j;
 int j0,i0;
 int total,aux=costo,cambio=false;
 for (i=1;i<n;i++)
 {
        for (j=i+1;j<=n;j++)
        {
                cambia(&perm[i].l,&perm[j].l);
                total=costo_total(perm);
                moneda=1;
                if (total==optimo)
                {
                        aux=total;
                        return aux;
                }
                if (total==aux)
```

```
                    //volado
                    moneda=(int)mylrand48(mul0,mul1,mul2,add,in) % 2;
                  if (total<=aux&&moneda==1)
                  {
                    j0=j;
                    i0=i;
                    aux=total;
                    cambio=true;
                  }
                  cambia(&perm[j].l,&perm[i].l);
        }
}
if (cambio==true)
{
        cambia(&perm[i0].l,&perm[j0].l);
        return aux;
}
else
        {
                return aux;
        }
}

/*inicializa los conjuntos coml, comj, city, facility */
void init_conjunto(tset *city,tset *facility,tset *coml,tset *comj)
{
        int i;
        initset(facility);
        initset(city);
        initset(comj);
        initset(coml);
        //inicializa los conjuntos coml y comj de 1..n
        for (i=1;i<=n;i++)
        {
                addset(coml,i);
                addset(comj,i);
        }
}

/* lee de archivo binario y carga en los datos en las matrices flujo y distancia*/
void carga_archivo(const char archivo[])
{
        int dato,i,j;
        FILE *arch;
        if ((arch=fopen(archivo,"rb"))==NULL)
        {
                printf("Error de archivo");
                exit(0);
        }
        i=j=1;
        while ((i<=n)&&(feof(arch)==0))
        {
                if (j==n+1)
                {
                        i++;
                        j=i;
                }
                if (i==j)
                {
                        distancia[i][j]=0;
                        j++;
                }
```

```
		else
		{
			fread(&dato,sizeof(dato),1,arch);
			distancia[i][j]=dato;
			if (distancia[i][j]==distancia[j][i])
				distancia[i][j]=distancia[j][i];
			distancia[j][i]=distancia[i][j];
			j++;
		}
	}
	i=j=1;
	while ((i<=n)&&(feof(arch)==0))
	{
		if (j==n+1)
		{
			i++;
			j=i;
		}
		if (i==j)
		{
			flujo[i][j]=0;
			j++;
		}
		else
		{
			fread(&dato,sizeof(dato),1,arch);
			flujo[i][j]=dato;
			if (flujo[i][j]==flujo[j][i])
				flujo[i][j]=flujo[j][i];
			flujo[j][i]=flujo[i][j];
			j++;
		}
	}
}

/* ordena la permutacion */
void sort_perm(parejas perm[])
{
	parejas aux;
	int i,j;
	for (i=1;i<=n-1;i++)
		for (j=1;j<=n-i;j++)
			if (perm[j].j>perm[j+1].j)
			{
				aux.j=perm[j].j;
				aux.l=perm[j].l;
				perm[j].j=perm[j+1].j;
				perm[j].l=perm[j+1].l;
				perm[j+1].j=aux.j;
				perm[j+1].l=aux.l;
			}
}

/* copia una permutacion de fuente a destino*/
void copiap(parejas fuente[],parejas destino[])
{

	int i;
	for (i=1;i<=n;i++)
	{
		destino[i].j=fuente[i].j;
				destino[i].l=fuente[i].l;
```

```
	}
}

/* carga las listas de distancias y flujos de la matriz de datos */
void carga_listas(tlista ldist[],tlista lflujo[])
{
  int i,j,k=1;
  for (i=1;i<=n;i++)
	for(j=i+1;j<=n;j++)
	{
		lflujo[k].valor=flujo[i][j];
		lflujo[k].i=i;
		lflujo[k].j=j;
		ldist[k].valor=distancia[i][j];
		ldist[k].i=i;
		ldist[k].j=j;
		k++;
	}
}

void grasp()
{
	//variables para funciones aleatorias
	static unsigned short internal_state[3] = {1, 0, 0};
	static unsigned short multiplier0 = RAND48_MULT0;
	static unsigned short multiplier1 = RAND48_MULT1;
	static unsigned short multiplier2 = RAND48_MULT2;
	static unsigned short additiver = RAND48_ADD;

	struct rusage ru;
	float t_inicial,t_final,timep;
	time_t t;
	struct tcset city,facility,comI,comj,mov,posibles;	//variables para conjuntos
	parejas perm[Max];					//permutacion
	char archivo[8],archivo1[12]="sal";
	unsigned long  iter_grasp,iter_mejor,			//contadores de iteraciones
			mejorcosto = 320000;			// mejor costo
	parejas mejorp[Max];					//mejor permutacion
	int i,j,
		best,						//mejor costo de permutacion
		total,						//costo de permutacion
		band,
		my_ide = m_get_myid();				//identificador de procesador
	tlista lflujo[maxlista],ldist[maxlista];		//listas de flujos y distancias
	costos listacosto[maxlista];				//lista de costos
	FILE *outfile;						//archivo de salida de procesador

	//tomar el tiempo inicial de ejecucion correspondiente al procesador
	getrusage(RUSAGE_SELF,&ru);
	t_inicial = (float)ru.ru_utime.tv_sec + (float)(ru.ru_utime.tv_usec)/1000000;

	//crear una secuencia de numeros seudoaleatorios con la semilla correspondiente al procesador
	mysrand48(seed[my_ide],&multiplier0,&multiplier1,&multiplier2,&additiver,internal_state);

	//cargar la lista de distancias y flujos de la matriz de datos
	carga_listas(ldist,lflujo);

	//ordenar los flujos de mayor a menor
	sortlflujo(lflujo,1,res);
	//ordenar los flujos con respecto a los indices
	ordenaij(lflujo,res);
	//ordenar las distancias de menor a mayor
```

```
sortldist(ldist,1,res);
//ordenar los flujos con respecto a los indices
ordenaij(ldist,res);
//calcular la lista de costos
mul_ldistlflujo(listacosto,ldist,lflujo,parametro1);
//inicializar los contadores de iteraciones
iter_grasp=iter_mejor=0;

//inicializar los conjuntos
initset(&city);
initset(&facility);
initset(&coml);
initset(&comj);

do
{
        //inicializar con los valores iniciales a los conjuntos
        init_conjunto(&city,&facility,&coml,&comj);

        //tomar un elemento de la lista de costos aleatoriamente y crear
        // los primeros dos elementos de la permutacion

randomiza(listacosto,perm,&city,&facility,&coml,&comj,parametro2,multiplier0,multiplier1,multiplier2,
                additiver,internal_state);
        //construir el resto de la permutacion

stage2(perm,&city,&facility,&coml,&comj,multiplier0,multiplier1,multiplier2,additiver,internal_state);

        //ordenar la permutacion
        sort_perm(perm);

        //calcular el costo de la permutacion inicial
        best=costo_total(perm);
        do
        {
                //calcular los vecinos y encontrar el menor de ellos
                total=two_exchange(perm, best,multiplier0,multiplier1,multiplier2,additiver,internal_state);
                if (total<best)
                {
                        band=true;
                        best=total;
                }
                else
                        band=false;
        }while (band!=false);//repetir hasta no encontrar un vecino menor
        //Γel costo del vecino es menor que el mejor encontrado
        if (best<mejorcosto)
        {
                //guardar la encontrada permutacion con costo menor
                copiap(perm,mejorp);
                //actualizar mejor costo encontrado
                mejorcosto=best;
                //actualizar mejor iteracion
                iter_mejor = iter_grasp;
        }
        iter_grasp++;
//repetir hasta no encontrar un costo menor en (iter_grasp - iter_mejor)
//iteraciones > max_iter o hasta encontrar el optimo
}while(((iter_grasp-itermejor)<max_iter)&&(optimo<mejorcosto));
//tomar el tiempo final de ejecucion correspondiente al procesador
getrusage(RUSAGE_SELF,&ru);
t_final = (float)ru.ru_utime.tv_sec + (float)(ru.ru_utime.tv_usec)/1000000;
```

```
	//calcular el tiempo total de ejecucion
	timep = (t_final- t_inicial);
	//guardar  mejor costo correspondiente al numero procesador
	mejor_total[my_ide] = mejorcosto;
	//guardar mejor tiempo correspondiente al numero procesador
	tproc[my_ide]=timep;
	//guardar numero de iteraciones correspondiente al numero procesador
	iter_total[my_ide] = iter_grasp;
	//guardar la mejor permutacion correspondiente al numero procesador
	for (i=1;i<=n;i++)
	{
		perm_total[my_ide][i] = mejorp[i].l;
	}

	//imprime cada uno de los datos al archivo de salida de procesador
	sprintf(archivo,"%d",my_ide);
	strcat(archivo1,archivo);
	outfile = fopen (archivo1,"w");
	fprintf (outfile,"my_id = %7d\n",my_ide);
	for (j=1;j<=n;j++)
	{
		fprintf (outfile,"%3d",perm_total[my_ide][j]);
	}
	fprintf (outfile,"\n");
	fprintf (outfile,"%7d",mejor_total[my_ide]);
	fprintf (outfile,"%7d",iter_total[my_ide]);
	fprintf (outfile,"%12.8f",tproc[my_ide]);
	fclose (outfile);
}

//subrutina vacia para crear hilos
void vacia(void)
{
		/*vacia routine to create threads */
}

void main(argc, argv)
int argc;
char *argv[];
{
	FILE *outfile;					//archivo de salida general
	struct rusage ru;
	float t_inicial,t_final,timep;
	time_t t;
	int i,j;

	if (argc != 7)
	{
		exit (1213);
	}

	carga_archivo(argv[1]);			//nombre del archivo de datos
	n = atoi(argv[2]);				//dimension de la matriz
	max_iter = atoi(argv[3]);			//maximo de iteraciones
	optimo = atoi(argv[4]);			//funcion objetivo
	Np = atoi(argv[5]);				//numero de procesadores
	outfile = fopen (argv[6],"w");		//archivo de salida general


	m_set_procs(Np);				//fijar el numero de proceasores a utilizar
	fprintf (outfile,"Np = %d\n",Np);
```

```
    // crear semillas de numeros seudoaleatorios para cada uno de los procesadores
    for (i=0;i<Np;i++)
            seed[i] = time(&t)*(i+1);

    //calcular parametros para cortar las listas de distancias y flujos
    res=(n*(n-1))/2;
    parametro1=(int)(res*.1)+1;
    parametro2=(int)(parametro1*.5)+1;
    //tomar el tiempo inicial de ejecucion
    getrusage(RUSAGE_SELF,&ru);
    t_inicial = (float)ru.ru_utime.tv_sec + (float)(ru.ru_utime.tv_usec)/1000000;

    if (Np > 1)
    {
            m_fork(vacia);              //crear un proceso inutil en cada procesador
            m_park_procs();             //suspende procesos hijos

            retval = schedctl (SCHEDMODE,SGS_GANG);
            if (retval == -1 )
            {
                    perror("schedctl");
                    exit(100);
            }

            m_rele_procs();             //reiniciar procesos
    }

    m_fork(grasp);              //ejecutar grasp en cada procesador
    m_kill_procs();             //matar los procesos creados

    //tomar el tiempo final de ejecucion
    getrusage(RUSAGE_SELF,&ru);
    t_final = (float)ru.ru_utime.tv_sec +(float)(ru.ru_utime.tv_usec)/1000000;

    //calcular el tiempo total de ejecucion
    timep = (t_final- t_inicial);

    //Guardar los datos en el archivo de salida general
    fprintf (outfile,"Np = %d\n",Np);
    fprintf (outfile,"tiempo %12.8f\n",timep);
    for (i=0;i<Np;i++)
    {
            for (j=1;j<=n;j++)
                    fprintf (outfile,"%6d",perm_total[i][j]);
            fprintf (outfile,"%7d",mejor_total[i]);
            fprintf (outfile,"%7d",iter_total[i]);
            fprintf (outfile,"\n%12.8f",tproc[i]);
            fprintf (outfile,"\n");
    }
    fclose (outfile);
}
```

# Apéndice B

Las siguientes matrices están dadas en forma compacta y son las instancias de prueba utilizadas por nuestra implementación, fueron obtenidas de la pagina de QAPLIB

Nug5

| - | 1 | 1 | 2 | 3 |
|---|---|---|---|---|
| 5 | - | 2 | 1 | 2 |
| 2 | 3 | - | 1 | 2 |
| 4 | 0 | 0 | - | 1 |
| 1 | 2 | 0 | 5 | - |

Nug6

| - | 1 | 2 | 1 | 2 | 3 |
|---|---|---|---|---|---|
| 5 | - | 1 | 2 | 1 | 2 |
| 2 | 3 | - | 3 | 2 | 1 |
| 4 | 0 | 0 | - | 1 | 2 |
| 1 | 2 | 0 | 5 | - | 1 |
| 0 | 2 | 0 | 2 | 10 | - |

Nug7

| - | 1 | 2 | 3 | 2 | 3 | 4 |
|---|---|---|---|---|---|---|
| 5 | - | 1 | 2 | 1 | 2 | 3 |
| 2 | 3 | - | 1 | 2 | 1 | 2 |
| 4 | 0 | 1 | - | 3 | 2 | 1 |
| 1 | 2 | 0 | 5 | - | 1 | 2 |
| 0 | 2 | 2 | 2 | 10 | - | 1 |
| 0 | 2 | 5 | 2 | 0 | 5 | - |

Nug8

| - | 1 | 2 | 3 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|---|---|
| 5 | - | 1 | 2 | 2 | 1 | 2 | 3 |
| 2 | 3 | - | 1 | 3 | 2 | 1 | 2 |
| 4 | 0 | 0 | - | 4 | 3 | 2 | 1 |
| 1 | 2 | 0 | 5 | - | 1 | 2 | 3 |
| 0 | 2 | 0 | 2 | 10 | - | 1 | 2 |
| 0 | 2 | 0 | 2 | 0 | 5 | - | 1 |
| 6 | 0 | 5 | 10 | 0 | 1 | 10 | - |

Nug12

| - | 1 | 2 | 3 | 1 | 2 | 3 | 4 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | - | 1 | 2 | 2 | 1 | 2 | 3 | 3 | 2 | 3 | 4 |
| 2 | 3 | - | 1 | 3 | 2 | 1 | 2 | 4 | 3 | 2 | 3 |
| 4 | 0 | 0 | - | 4 | 3 | 2 | 1 | 5 | 4 | 3 | 2 |
| 1 | 2 | 0 | 5 | - | 1 | 2 | 3 | 1 | 2 | 3 | 4 |
| 0 | 2 | 0 | 2 | 10 | - | 1 | 2 | 2 | 1 | 2 | 3 |
| 0 | 2 | 0 | 2 | 0 | 5 | - | 1 | 3 | 2 | 1 | 2 |
| 6 | 0 | 5 | 10 | 0 | 1 | 10 | - | 4 | 3 | 2 | 1 |
| 2 | 4 | 5 | 0 | 0 | 1 | 5 | 0 | - | 1 | 2 | 3 |
| 1 | 5 | 2 | 0 | 5 | 5 | 2 | 0 | 0 | - | 1 | 2 |
| 1 | 0 | 2 | 5 | 1 | 4 | 3 | 5 | 10 | 5 | - | 1 |
| 1 | 0 | 2 | 5 | 1 | 0 | 3 | 0 | 10 | 0 | 2 | - |

Nug15

| - | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 10 | - | 1 | 2 | 3 | 2 | 1 | 2 | 3 | 4 | 3 | 2 | 3 | 4 | 5 |
| 0 | 1 | - | 1 | 2 | 3 | 2 | 1 | 2 | 3 | 4 | 3 | 2 | 3 | 4 |
| 5 | 3 | 10 | - | 1 | 4 | 3 | 2 | 1 | 2 | 5 | 4 | 3 | 2 | 3 |
| 1 | 2 | 2 | 1 | - | 5 | 4 | 3 | 2 | 1 | 6 | 5 | 4 | 3 | 2 |
| 0 | 2 | 0 | 1 | 3 | - | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 |
| 1 | 2 | 2 | 5 | 5 | 2 | - | 1 | 2 | 3 | 2 | 1 | 2 | 3 | 4 |
| 2 | 3 | 5 | 0 | 5 | 2 | 6 | - | 1 | 2 | 3 | 2 | 1 | 2 | 3 |
| 2 | 2 | 4 | 0 | 5 | 1 | 0 | 5 | - | 1 | 4 | 3 | 2 | 1 | 2 |
| 2 | 0 | 5 | 2 | 1 | 5 | 1 | 2 | 0 | - | 5 | 4 | 3 | 2 | 1 |
| 2 | 2 | 2 | 1 | 0 | 0 | 5 | 10 | 10 | 0 | - | 1 | 2 | 3 | 4 |
| 0 | 0 | 2 | 0 | 3 | 0 | 5 | 0 | 5 | 4 | 5 | - | 1 | 2 | 3 |
| 4 | 10 | 5 | 2 | 0 | 2 | 5 | 5 | 10 | 0 | 0 | 3 | - | 1 | 2 |
| 0 | 5 | 5 | 5 | 5 | 5 | 1 | 0 | 0 | 0 | 5 | 3 | 10 | - | 1 |
| 0 | 0 | 5 | 0 | 5 | 10 | 0 | 0 | 2 | 5 | 0 | 0 | 2 | 4 | - |

Nug20

| - | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 | 2 | 3 | 4 | 5 | 6 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | - | 1 | 2 | 3 | 2 | 1 | 2 | 3 | 4 | 3 | 2 | 3 | 4 | 5 | 4 | 3 | 4 | 5 | 6 |
| 5 | 3 | - | 1 | 2 | 3 | 2 | 1 | 2 | 3 | 4 | 3 | 2 | 3 | 4 | 5 | 4 | 3 | 4 | 5 |
| 0 | 10 | 2 | - | 1 | 4 | 3 | 2 | 1 | 2 | 5 | 4 | 3 | 2 | 3 | 6 | 5 | 4 | 3 | 4 |
| 5 | 5 | 0 | 1 | - | 5 | 4 | 3 | 2 | 1 | 6 | 5 | 4 | 3 | 2 | 7 | 6 | 5 | 4 | 3 |
| 2 | 1 | 5 | 0 | 5 | - | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 | 2 | 3 | 4 | 5 | 6 |
| 10 | 5 | 2 | 5 | 6 | 5 | - | 1 | 2 | 3 | 2 | 1 | 2 | 3 | 4 | 3 | 2 | 3 | 4 | 5 |
| 3 | 1 | 4 | 2 | 5 | 2 | 0 | - | 1 | 2 | 3 | 2 | 1 | 2 | 3 | 4 | 3 | 2 | 3 | 4 |
| 1 | 2 | 4 | 1 | 2 | 1 | 0 | 1 | - | 1 | 4 | 3 | 2 | 1 | 2 | 5 | 4 | 3 | 2 | 3 |
| 5 | 4 | 5 | 0 | 5 | 6 | 0 | 1 | 2 | - | 5 | 4 | 3 | 2 | 1 | 6 | 5 | 4 | 3 | 2 |
| 5 | 2 | 0 | 10 | 2 | 0 | 5 | 10 | 0 | 5 | - | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 |
| 5 | 5 | 0 | 2 | 0 | 0 | 10 | 10 | 3 | 5 | 5 | - | 1 | 2 | 3 | 2 | 1 | 2 | 3 | 4 |
| 0 | 0 | 0 | 2 | 5 | 10 | 2 | 2 | 5 | 0 | 2 | 2 | - | 1 | 2 | 3 | 2 | 1 | 2 | 3 |
| 0 | 10 | 5 | 0 | 1 | 0 | 2 | 0 | 5 | 5 | 5 | 10 | 2 | - | 1 | 4 | 3 | 2 | 1 | 2 |
| 5 | 10 | 1 | 2 | 1 | 2 | 5 | 10 | 0 | 1 | 1 | 5 | 2 | 5 | - | 5 | 4 | 3 | 2 | 1 |
| 4 | 3 | 0 | 1 | 1 | 0 | 1 | 2 | 5 | 0 | 10 | 0 | 1 | 5 | 3 | - | 1 | 2 | 3 | 4 |
| 4 | 0 | 0 | 5 | 5 | 1 | 2 | 5 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | - | 1 | 2 | 3 |
| 0 | 5 | 5 | 2 | 2 | 0 | 1 | 2 | 0 | 5 | 2 | 1 | 0 | 5 | 5 | 0 | 5 | - | 1 | 2 |
| 0 | 10 | 0 | 5 | 5 | 1 | 0 | 2 | 0 | 5 | 2 | 2 | 0 | 5 | 10 | 2 | 2 | 1 | - | 1 |
| 1 | 5 | 0 | 5 | 1 | 5 | 10 | 10 | 2 | 2 | 5 | 5 | 5 | 0 | 10 | 0 | 0 | 1 | 6 | - |

Nug21

| - | 1 | 2 | 3 | 4 | 5 | 6 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | - | 1 | 2 | 3 | 4 | 5 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 3 | 2 | 3 | 4 | 5 | 6 | 7 |
| 2 | 4 | - | 1 | 2 | 3 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 4 | 3 | 2 | 3 | 4 | 5 | 6 |
| 0 | 0 | 3 | - | 1 | 2 | 3 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 4 | 3 | 2 | 3 | 4 | 5 |
| 0 | 10 | 4 | 0 | - | 1 | 2 | 5 | 4 | 3 | 2 | 1 | 2 | 3 | 6 | 5 | 4 | 3 | 2 | 3 | 4 |
| 2 | 4 | 0 | 0 | 5 | - | 1 | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 7 | 6 | 5 | 4 | 3 | 2 | 3 |
| 10 | 0 | 5 | 0 | 2 | 1 | - | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 8 | 7 | 6 | 5 | 4 | 3 | 2 |
| 5 | 0 | 5 | 2 | 0 | 2 | 10 | - | 1 | 2 | 3 | 4 | 5 | 6 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 2 | 5 | 2 | 0 | 2 | 10 | 1 | - | 1 | 2 | 3 | 4 | 5 | 2 | 1 | 2 | 3 | 4 | 5 | 6 |
| 5 | 2 | 1 | 0 | 0 | 1 | 5 | 3 | 10 | - | 1 | 2 | 3 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 |
| 2 | 1 | 4 | 6 | 0 | 4 | 10 | 5 | 2 | 5 | - | 1 | 2 | 3 | 4 | 3 | 2 | 1 | 2 | 3 | 4 |
| 5 | 0 | 1 | 0 | 2 | 10 | 10 | 0 | 1 | 5 | 0 | - | 1 | 2 | 5 | 4 | 3 | 2 | 1 | 2 | 3 |
| 0 | 5 | 0 | 2 | 0 | 10 | 6 | 0 | 5 | 6 | 0 | 5 | - | 1 | 6 | 5 | 4 | 3 | 2 | 1 | 2 |
| 0 | 0 | 4 | 5 | 0 | 2 | 0 | 0 | 2 | 0 | 1 | 5 | 2 | - | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| 2 | 0 | 0 | 2 | 0 | 5 | 0 | 2 | 0 | 1 | 2 | 2 | 0 | 2 | - | 1 | 2 | 3 | 4 | 5 | 6 |
| 0 | 0 | 4 | 5 | 0 | 5 | 10 | 4 | 3 | 5 | 1 | 0 | 4 | 1 | 4 | - | 1 | 2 | 3 | 4 | 5 |
| 5 | 0 | 0 | 1 | 2 | 0 | 2 | 5 | 0 | 5 | 0 | 0 | 2 | 0 | 5 | 0 | - | 1 | 2 | 3 | 4 |
| 6 | 2 | 6 | 1 | 1 | 5 | 1 | 2 | 2 | 0 | 2 | 0 | 2 | 5 | 1 | 3 | 2 | - | 1 | 2 | 3 |
| 3 | 0 | 3 | 1 | 0 | 0 | 10 | 10 | 0 | 5 | 0 | 0 | 1 | 3 | 0 | 0 | 2 | 5 | - | 1 | 2 |
| 0 | 1 | 2 | 1 | 0 | 0 | 1 | 6 | 0 | 2 | 0 | 2 | 0 | 10 | 1 | 2 | 0 | 1 | 0 | - | 1 |
| 1 | 6 | 5 | 2 | 2 | 0 | 5 | 0 | 4 | 3 | 0 | 0 | 6 | 0 | 0 | 2 | 0 | 2 | 5 | 5 | - |

Nug22

| - | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | - | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 2 | 4 | - | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 | 0 | 3 | - | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | 10 | 4 | 0 | - | 1 | 2 | 3 | 4 | 5 | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 2 | 4 | 0 | 0 | 5 | - | 1 | 2 | 3 | 4 | 5 | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 6 |
| 10 | 0 | 5 | 0 | 2 | 1 | - | 1 | 2 | 3 | 4 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 |
| 5 | 0 | 5 | 2 | 0 | 2 | 10 | - | 1 | 2 | 3 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 3 | 4 |
| 0 | 2 | 5 | 2 | 0 | 2 | 10 | 1 | - | 1 | 2 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 3 |
| 5 | 2 | 1 | 0 | 0 | 1 | 5 | 3 | 10 | - | 1 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 2 |
| 2 | 1 | 4 | 6 | 0 | 4 | 10 | 5 | 2 | 5 | - | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| 5 | 0 | 1 | 0 | 2 | 10 | 10 | 0 | 1 | 5 | 0 | - | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 0 | 5 | 0 | 2 | 0 | 10 | 6 | 0 | 5 | 6 | 0 | 5 | - | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 | 0 | 4 | 5 | 0 | 2 | 0 | 0 | 2 | 0 | 1 | 5 | 2 | - | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 2 | 0 | 0 | 2 | 0 | 5 | 0 | 2 | 0 | 1 | 2 | 2 | 0 | 2 | - | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 0 | 4 | 5 | 0 | 5 | 10 | 4 | 3 | 5 | 1 | 0 | 4 | 1 | 4 | - | 1 | 2 | 3 | 4 | 5 | 6 |
| 5 | 0 | 0 | 1 | 2 | 0 | 2 | 5 | 0 | 5 | 0 | 0 | 2 | 0 | 5 | 0 | - | 1 | 2 | 3 | 4 | 5 |
| 6 | 2 | 6 | 1 | 1 | 5 | 1 | 2 | 2 | 0 | 2 | 0 | 2 | 5 | 1 | 3 | 2 | - | 1 | 2 | 3 | 4 |
| 3 | 0 | 3 | 1 | 0 | 0 | 10 | 10 | 0 | 5 | 0 | 0 | 1 | 3 | 0 | 0 | 2 | 5 | - | 1 | 2 | 3 |
| 0 | 1 | 2 | 1 | 0 | 0 | 1 | 6 | 0 | 2 | 0 | 2 | 0 | 10 | 1 | 2 | 0 | 1 | 0 | - | 1 | 2 |
| 1 | 6 | 5 | 2 | 2 | 0 | 5 | 0 | 4 | 3 | 0 | 0 | 6 | 0 | 0 | 2 | 0 | 2 | 5 | 5 | - | 1 |
| 10 | 1 | 5 | 2 | 0 | 10 | 5 | 5 | 0 | 5 | 6 | 4 | 2 | 0 | 5 | 0 | 0 | 10 | 5 | 2 | 4 | - |

Nug24

| - | 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 5 | 6 | 2 | 3 | 4 | 5 | 6 | 7 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | - | 1 | 2 | 3 | 4 | 2 | 1 | 2 | 3 | 4 | 5 | 3 | 2 | 3 | 4 | 5 | 6 | 4 | 3 | 4 | 5 | 6 | 7 |
| 2 | 4 | - | 1 | 2 | 3 | 3 | 2 | 1 | 2 | 3 | 4 | 4 | 3 | 2 | 3 | 4 | 5 | 5 | 4 | 3 | 4 | 5 | 6 |
| 0 | 0 | 3 | - | 1 | 2 | 4 | 3 | 2 | 1 | 2 | 3 | 5 | 4 | 3 | 2 | 3 | 4 | 6 | 5 | 4 | 3 | 4 | 5 |
| 0 | 10 | 4 | 0 | - | 1 | 5 | 4 | 3 | 2 | 1 | 2 | 6 | 5 | 4 | 3 | 2 | 3 | 7 | 6 | 5 | 4 | 3 | 4 |
| 2 | 4 | 0 | 0 | 5 | - | 6 | 5 | 4 | 3 | 2 | 1 | 7 | 6 | 5 | 4 | 3 | 2 | 8 | 7 | 6 | 5 | 4 | 3 |
| 10 | 0 | 5 | 0 | 2 | 1 | - | 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 5 | 6 | 2 | 3 | 4 | 5 | 6 | 7 |
| 5 | 0 | 5 | 2 | 0 | 2 | 10 | - | 1 | 2 | 3 | 4 | 2 | 1 | 2 | 3 | 4 | 5 | 3 | 2 | 3 | 4 | 5 | 6 |
| 0 | 2 | 5 | 2 | 0 | 2 | 10 | 1 | - | 1 | 2 | 3 | 3 | 2 | 1 | 2 | 3 | 4 | 4 | 3 | 2 | 3 | 4 | 5 |
| 5 | 2 | 1 | 0 | 0 | 1 | 5 | 3 | 10 | - | 1 | 2 | 4 | 3 | 2 | 1 | 2 | 3 | 5 | 4 | 3 | 2 | 3 | 4 |
| 2 | 1 | 4 | 6 | 0 | 4 | 10 | 5 | 2 | 5 | - | 1 | 5 | 4 | 3 | 2 | 1 | 2 | 6 | 5 | 4 | 3 | 2 | 3 |
| 5 | 0 | 1 | 0 | 2 | 10 | 10 | 0 | 1 | 5 | 0 | - | 6 | 5 | 4 | 3 | 2 | 1 | 7 | 6 | 5 | 4 | 3 | 2 |
| 0 | 5 | 0 | 2 | 0 | 10 | 6 | 0 | 5 | 6 | 0 | 5 | - | 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 5 | 6 |
| 0 | 0 | 4 | 5 | 0 | 2 | 0 | 0 | 2 | 0 | 1 | 5 | 2 | - | 1 | 2 | 3 | 4 | 2 | 1 | 2 | 3 | 4 | 5 |
| 2 | 0 | 0 | 2 | 0 | 5 | 0 | 2 | 0 | 1 | 2 | 2 | 0 | 2 | - | 1 | 2 | 3 | 3 | 2 | 1 | 2 | 3 | 4 |
| 0 | 0 | 4 | 5 | 0 | 5 | 10 | 4 | 3 | 5 | 1 | 0 | 4 | 1 | 4 | - | 1 | 2 | 4 | 3 | 2 | 1 | 2 | 3 |
| 5 | 0 | 0 | 1 | 2 | 0 | 2 | 5 | 0 | 5 | 0 | 0 | 2 | 0 | 5 | 0 | - | 1 | 5 | 4 | 3 | 2 | 1 | 2 |
| 6 | 2 | 6 | 1 | 1 | 5 | 1 | 2 | 2 | 0 | 2 | 0 | 2 | 5 | 1 | 3 | 2 | - | 6 | 5 | 4 | 3 | 2 | 1 |
| 3 | 0 | 3 | 1 | 0 | 0 | 10 | 10 | 0 | 5 | 0 | 0 | 1 | 3 | 0 | 0 | 2 | 5 | - | 1 | 2 | 3 | 4 | 5 |
| 0 | 1 | 2 | 1 | 0 | 0 | 1 | 6 | 0 | 2 | 0 | 2 | 0 | 10 | 1 | 2 | 0 | 1 | 0 | - | 1 | 2 | 3 | 4 |
| 1 | 6 | 5 | 2 | 2 | 0 | 5 | 0 | 4 | 3 | 0 | 0 | 6 | 0 | 0 | 2 | 0 | 2 | 5 | 5 | - | 1 | 2 | 3 |
| 10 | 1 | 5 | 2 | 0 | 10 | 5 | 5 | 0 | 5 | 6 | 4 | 2 | 0 | 5 | 0 | 0 | 10 | 5 | 2 | 4 | - | 1 | 2 |
| 0 | 0 | 2 | 4 | 5 | 0 | 2 | 5 | 5 | 0 | 6 | 5 | 1 | 4 | 0 | 2 | 6 | 10 | 1 | 1 | 0 | 5 | - | 1 |
| 10 | 1 | 1 | 0 | 1 | 0 | 3 | 2 | 2 | 5 | 0 | 10 | 5 | 2 | 2 | 0 | 5 | 4 | 0 | 3 | 1 | 0 | 0 | - |

Nug25

| - | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 | 2 | 3 | 4 | 5 | 6 | 3 | 4 | 5 | 6 | 7 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | - | 1 | 2 | 3 | 2 | 1 | 2 | 3 | 4 | 3 | 2 | 3 | 4 | 5 | 4 | 3 | 4 | 5 | 6 | 5 | 4 | 5 | 6 | 7 |
| 2 | 4 | - | 1 | 2 | 3 | 2 | 1 | 2 | 3 | 4 | 3 | 2 | 3 | 4 | 5 | 4 | 3 | 4 | 5 | 6 | 5 | 4 | 5 | 6 |
| 0 | 0 | 3 | - | 1 | 4 | 3 | 2 | 1 | 2 | 5 | 4 | 3 | 2 | 3 | 6 | 5 | 4 | 3 | 4 | 7 | 6 | 5 | 4 | 5 |
| 0 | 10 | 4 | 0 | - | 5 | 4 | 3 | 2 | 1 | 6 | 5 | 4 | 3 | 2 | 7 | 6 | 5 | 4 | 3 | 8 | 7 | 6 | 5 | 4 |
| 10 | 0 | 5 | 0 | 2 | - | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 | 2 | 3 | 4 | 5 | 6 | 3 | 4 | 5 | 6 | 7 |
| 5 | 0 | 5 | 2 | 0 | 10 | - | 1 | 2 | 3 | 2 | 1 | 2 | 3 | 4 | 3 | 2 | 3 | 4 | 5 | 4 | 3 | 4 | 5 | 6 |
| 0 | 2 | 5 | 2 | 0 | 10 | 1 | - | 1 | 2 | 3 | 2 | 1 | 2 | 3 | 4 | 3 | 2 | 3 | 4 | 5 | 4 | 3 | 4 | 5 |
| 5 | 2 | 1 | 0 | 0 | 5 | 3 | 10 | - | 1 | 4 | 3 | 2 | 1 | 2 | 5 | 4 | 3 | 2 | 3 | 6 | 5 | 4 | 3 | 4 |
| 2 | 1 | 4 | 6 | 0 | 10 | 5 | 2 | 5 | - | 5 | 4 | 3 | 2 | 1 | 6 | 5 | 4 | 3 | 2 | 7 | 6 | 5 | 4 | 3 |
| 0 | 5 | 0 | 2 | 0 | 6 | 0 | 5 | 6 | 0 | - | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 | 2 | 3 | 4 | 5 | 6 |
| 0 | 0 | 4 | 5 | 0 | 0 | 0 | 2 | 0 | 1 | 2 | - | 1 | 2 | 3 | 2 | 1 | 2 | 3 | 4 | 3 | 2 | 3 | 4 | 5 |
| 2 | 0 | 0 | 2 | 0 | 0 | 2 | 0 | 1 | 2 | 0 | 2 | - | 1 | 2 | 3 | 2 | 1 | 2 | 3 | 4 | 3 | 2 | 3 | 4 |
| 0 | 0 | 4 | 5 | 0 | 10 | 4 | 3 | 5 | 1 | 4 | 1 | 4 | - | 1 | 4 | 3 | 2 | 1 | 2 | 5 | 4 | 3 | 2 | 3 |
| 5 | 0 | 0 | 1 | 2 | 2 | 5 | 0 | 5 | 0 | 2 | 0 | 5 | 0 | - | 5 | 4 | 3 | 2 | 1 | 6 | 5 | 4 | 3 | 2 |
| 3 | 0 | 3 | 1 | 0 | 10 | 10 | 0 | 5 | 0 | 1 | 3 | 0 | 0 | 2 | - | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 |
| 0 | 1 | 2 | 1 | 0 | 1 | 6 | 0 | 2 | 0 | 0 | 10 | 1 | 2 | 0 | 0 | - | 1 | 2 | 3 | 2 | 1 | 2 | 3 | 4 |
| 1 | 6 | 5 | 2 | 2 | 5 | 0 | 4 | 3 | 0 | 6 | 0 | 0 | 2 | 0 | 5 | 5 | - | 1 | 2 | 3 | 2 | 1 | 2 | 3 |
| 10 | 1 | 5 | 2 | 0 | 5 | 5 | 0 | 5 | 6 | 2 | 0 | 5 | 0 | 0 | 5 | 2 | 4 | - | 1 | 4 | 3 | 2 | 1 | 2 |
| 0 | 0 | 2 | 4 | 5 | 2 | 5 | 5 | 0 | 6 | 1 | 4 | 0 | 2 | 6 | 1 | 1 | 0 | 5 | - | 5 | 4 | 3 | 2 | 1 |
| 2 | 2 | 0 | 2 | 0 | 5 | 5 | 0 | 2 | 4 | 5 | 0 | 0 | 5 | 3 | 5 | 1 | 0 | 4 | 4 | - | 1 | 2 | 3 | 4 |
| 1 | 2 | 0 | 0 | 2 | 0 | 0 | 5 | 10 | 5 | 0 | 0 | 0 | 0 | 5 | 2 | 5 | 0 | 4 | 4 | 1 | - | 1 | 2 | 3 |
| 1 | 5 | 3 | 2 | 1 | 2 | 5 | 2 | 10 | 3 | 0 | 4 | 5 | 5 | 0 | 1 | 6 | 0 | 5 | 1 | 0 | 0 | - | 1 | 2 |
| 1 | 1 | 1 | 2 | 0 | 0 | 5 | 2 | 1 | 2 | 1 | 2 | 1 | 2 | 0 | 2 | 5 | 5 | 0 | 0 | 10 | 0 | 0 | - | 1 |
| 0 | 10 | 0 | 5 | 2 | 1 | 0 | 5 | 5 | 2 | 5 | 5 | 1 | 5 | 5 | 10 | 5 | 0 | 2 | 2 | 1 | 0 | 0 | 2 | - |

Nug30

| - | 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 5 | 6 | 2 | 3 | 4 | 5 | 6 | 7 | 3 | 4 | 5 | 6 | 7 | 8 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | - | 1 | 2 | 3 | 4 | 2 | 1 | 2 | 3 | 4 | 5 | 3 | 2 | 3 | 4 | 5 | 6 | 4 | 3 | 4 | 5 | 6 | 7 | 5 | 4 | 5 | 6 | 7 | 8 |
| 2 | 4 | - | 1 | 2 | 3 | 3 | 2 | 1 | 2 | 3 | 4 | 4 | 3 | 2 | 3 | 4 | 5 | 5 | 4 | 3 | 4 | 5 | 6 | 6 | 5 | 4 | 5 | 6 | 7 |
| 0 | 0 | 3 | - | 1 | 2 | 4 | 3 | 2 | 1 | 2 | 3 | 5 | 4 | 3 | 2 | 3 | 4 | 6 | 5 | 4 | 3 | 4 | 5 | 7 | 6 | 5 | 4 | 5 | 6 |
| 0 | 10 | 4 | 0 | - | 1 | 5 | 4 | 3 | 2 | 1 | 2 | 6 | 5 | 4 | 3 | 2 | 3 | 7 | 6 | 5 | 4 | 3 | 4 | 8 | 7 | 6 | 5 | 4 | 5 |
| 2 | 4 | 0 | 0 | 5 | - | 6 | 5 | 4 | 3 | 2 | 1 | 7 | 6 | 5 | 4 | 3 | 2 | 8 | 7 | 6 | 5 | 4 | 3 | 9 | 8 | 7 | 6 | 5 | 4 |
| 10 | 0 | 5 | 0 | 2 | 1 | - | 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 5 | 6 | 2 | 3 | 4 | 5 | 6 | 7 | 3 | 4 | 5 | 6 | 7 | 8 |
| 5 | 0 | 5 | 2 | 0 | 2 | 10 | - | 1 | 2 | 3 | 4 | 2 | 1 | 2 | 3 | 4 | 5 | 3 | 2 | 3 | 4 | 5 | 6 | 4 | 3 | 4 | 5 | 6 | 7 |
| 0 | 2 | 5 | 2 | 0 | 2 | 10 | 1 | - | 1 | 2 | 3 | 3 | 2 | 1 | 2 | 3 | 4 | 4 | 3 | 2 | 3 | 4 | 5 | 5 | 4 | 3 | 4 | 5 | 6 |
| 5 | 2 | 1 | 0 | 0 | 1 | 5 | 3 | 10 | - | 1 | 2 | 4 | 3 | 2 | 1 | 2 | 3 | 5 | 4 | 3 | 2 | 3 | 4 | 6 | 5 | 4 | 3 | 4 | 5 |
| 2 | 1 | 4 | 6 | 0 | 4 | 10 | 5 | 2 | 5 | - | 1 | 5 | 4 | 3 | 2 | 1 | 2 | 6 | 5 | 4 | 3 | 2 | 3 | 7 | 6 | 5 | 4 | 3 | 4 |
| 5 | 0 | 1 | 0 | 2 | 10 | 10 | 0 | 1 | 5 | 0 | - | 6 | 5 | 4 | 3 | 2 | 1 | 7 | 6 | 5 | 4 | 3 | 2 | 8 | 7 | 6 | 5 | 4 | 3 |
| 0 | 5 | 0 | 2 | 0 | 10 | 6 | 0 | 5 | 6 | 0 | 5 | - | 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 5 | 6 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 0 | 4 | 5 | 0 | 2 | 0 | 0 | 2 | 0 | 1 | 5 | 2 | - | 1 | 2 | 3 | 4 | 2 | 1 | 2 | 3 | 4 | 5 | 3 | 2 | 3 | 4 | 5 | 6 |
| 2 | 0 | 0 | 2 | 0 | 5 | 0 | 2 | 0 | 1 | 2 | 2 | 0 | 2 | - | 1 | 2 | 3 | 3 | 2 | 1 | 2 | 3 | 4 | 4 | 3 | 2 | 3 | 4 | 5 |
| 0 | 0 | 4 | 5 | 0 | 5 | 10 | 4 | 3 | 5 | 1 | 0 | 4 | 1 | 4 | - | 1 | 2 | 4 | 3 | 2 | 1 | 2 | 3 | 5 | 4 | 3 | 2 | 3 | 4 |
| 5 | 0 | 0 | 1 | 2 | 0 | 2 | 5 | 0 | 5 | 0 | 0 | 2 | 0 | 5 | 0 | - | 1 | 5 | 4 | 3 | 2 | 1 | 2 | 6 | 5 | 4 | 3 | 2 | 3 |
| 6 | 2 | 6 | 1 | 1 | 5 | 1 | 2 | 2 | 0 | 2 | 0 | 2 | 5 | 1 | 3 | 2 | - | 6 | 5 | 4 | 3 | 2 | 1 | 7 | 6 | 5 | 4 | 3 | 2 |
| 3 | 0 | 3 | 1 | 0 | 0 | 10 | 10 | 0 | 5 | 0 | 0 | 1 | 3 | 0 | 0 | 2 | 5 | - | 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 5 | 6 |
| 0 | 1 | 2 | 1 | 0 | 0 | 1 | 6 | 0 | 2 | 0 | 2 | 0 | 10 | 1 | 2 | 0 | 1 | 0 | - | 1 | 2 | 3 | 4 | 2 | 1 | 2 | 3 | 4 | 5 |
| 1 | 6 | 5 | 2 | 2 | 0 | 5 | 0 | 4 | 3 | 0 | 0 | 6 | 0 | 0 | 2 | 0 | 2 | 5 | 5 | - | 1 | 2 | 3 | 3 | 2 | 1 | 2 | 3 | 4 |
| 10 | 1 | 5 | 2 | 0 | 10 | 5 | 5 | 0 | 5 | 6 | 4 | 2 | 0 | 5 | 0 | 0 | 10 | 5 | 2 | 4 | - | 1 | 2 | 4 | 3 | 2 | 1 | 2 | 3 |
| 0 | 0 | 2 | 4 | 5 | 0 | 2 | 5 | 5 | 0 | 6 | 5 | 1 | 4 | 0 | 2 | 6 | 10 | 1 | 1 | 0 | 5 | - | 1 | 5 | 4 | 3 | 2 | 1 | 2 |
| 10 | 1 | 1 | 0 | 1 | 0 | 3 | 2 | 2 | 5 | 0 | 10 | 5 | 2 | 2 | 0 | 5 | 4 | 0 | 3 | 1 | 0 | 0 | - | 6 | 5 | 4 | 3 | 2 | 1 |
| 2 | 2 | 0 | 2 | 0 | 0 | 5 | 5 | 0 | 2 | 4 | 1 | 5 | 0 | 0 | 5 | 3 | 0 | 5 | 1 | 0 | 4 | 4 | 5 | - | 1 | 2 | 3 | 4 | 5 |
| 1 | 2 | 0 | 0 | 2 | 4 | 0 | 0 | 5 | 10 | 5 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 2 | 5 | 0 | 4 | 4 | 5 | 1 | - | 1 | 2 | 3 | 4 |
| 1 | 5 | 3 | 2 | 1 | 0 | 2 | 5 | 2 | 10 | 3 | 0 | 0 | 4 | 5 | 5 | 0 | 5 | 1 | 6 | 0 | 5 | 1 | 0 | 0 | 0 | - | 1 | 2 | 3 |
| 1 | 1 | 1 | 2 | 0 | 10 | 0 | 5 | 2 | 1 | 2 | 0 | 1 | 2 | 1 | 2 | 0 | 0 | 2 | 5 | 5 | 0 | 0 | 1 | 10 | 0 | 0 | - | 1 | 2 |
| 0 | 10 | 0 | 5 | 2 | 1 | 1 | 0 | 5 | 5 | 2 | 0 | 5 | 5 | 1 | 5 | 5 | 0 | 10 | 5 | 0 | 2 | 2 | 0 | 1 | 0 | 0 | 2 | - | 1 |
| 1 | 5 | 2 | 5 | 1 | 1 | 3 | 2 | 2 | 2 | 10 | 1 | 5 | 5 | 0 | 10 | 1 | 0 | 10 | 3 | 0 | 5 | 2 | 0 | 0 | 0 | 10 | 2 | 2 | - |

# Sko42

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| - | 1 | 2 | 3 | 4 | 5 | 6 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| 2 | - | 1 | 2 | 3 | 4 | 5 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 3 | 2 | 3 | 4 | 5 | 6 | 7 | 4 | 3 | 4 | 5 | 6 | 7 | 8 | 5 | 4 | 5 | 6 | 7 | 8 | 9 | 6 | 5 | 6 | 7 | 8 | 9 | 10 |
| 10 | 1 | - | 1 | 2 | 3 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 4 | 3 | 2 | 3 | 4 | 5 | 6 | 5 | 4 | 3 | 4 | 5 | 6 | 7 | 6 | 5 | 4 | 5 | 6 | 7 | 8 | 7 | 6 | 5 | 6 | 7 | 8 | 9 |
| 5 | 0 | 0 | - | 1 | 2 | 3 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 4 | 3 | 2 | 3 | 4 | 5 | 6 | 5 | 4 | 3 | 4 | 5 | 6 | 7 | 6 | 5 | 4 | 5 | 6 | 7 | 8 | 7 | 6 | 5 | 6 | 7 | 8 |
| 4 | 5 | 1 | 0 | - | 1 | 2 | 5 | 4 | 3 | 2 | 1 | 2 | 3 | 6 | 5 | 4 | 3 | 2 | 3 | 4 | 7 | 6 | 5 | 4 | 3 | 4 | 5 | 8 | 7 | 6 | 5 | 4 | 5 | 6 | 9 | 8 | 7 | 6 | 5 | 6 | 7 |
| 1 | 1 | 5 | 1 | 1 | - | 1 | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 7 | 6 | 5 | 4 | 3 | 2 | 3 | 8 | 7 | 6 | 5 | 4 | 3 | 4 | 9 | 8 | 7 | 6 | 5 | 4 | 5 | 10 | 9 | 8 | 7 | 6 | 5 | 6 |
| 5 | 0 | 10 | 0 | 6 | 0 | - | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 11 | 10 | 9 | 8 | 7 | 6 | 5 |
| 6 | 0 | 0 | 0 | 1 | 5 | 0 | - | 1 | 2 | 3 | 4 | 5 | 6 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 5 | 0 | 0 | 10 | 1 | 1 | 5 | 0 | - | 1 | 2 | 3 | 4 | 5 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 3 | 2 | 3 | 4 | 5 | 6 | 7 | 4 | 3 | 4 | 5 | 6 | 7 | 8 | 5 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 | 2 | 1 | 2 | 0 | 5 | 4 | 1 | 5 | - | 1 | 2 | 3 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 4 | 3 | 2 | 3 | 4 | 5 | 6 | 5 | 4 | 3 | 4 | 5 | 6 | 7 | 6 | 5 | 4 | 5 | 6 | 7 | 8 |
| 2 | 1 | 0 | 1 | 0 | 1 | 10 | 5 | 0 | 1 | - | 1 | 2 | 3 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 4 | 3 | 2 | 3 | 4 | 5 | 6 | 5 | 4 | 3 | 4 | 5 | 6 | 7 | 6 | 5 | 4 | 5 | 6 | 7 |
| 0 | 0 | 5 | 2 | 6 | 0 | 10 | 2 | 5 | 2 | 5 | - | 1 | 2 | 5 | 4 | 3 | 2 | 1 | 2 | 3 | 6 | 5 | 4 | 3 | 2 | 3 | 4 | 7 | 6 | 5 | 4 | 3 | 4 | 5 | 8 | 7 | 6 | 5 | 4 | 5 | 6 |
| 0 | 5 | 0 | 5 | 0 | 0 | 2 | 0 | 4 | 5 | 2 | 1 | - | 1 | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 7 | 6 | 5 | 4 | 3 | 2 | 3 | 8 | 7 | 6 | 5 | 4 | 3 | 4 | 9 | 8 | 7 | 6 | 5 | 4 | 5 |
| 1 | 10 | 3 | 5 | 0 | 0 | 0 | 0 | 2 | 3 | 5 | 0 | 0 | - | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 10 | 9 | 8 | 7 | 6 | 5 | 4 |
| 2 | 0 | 2 | 5 | 2 | 1 | 2 | 2 | 10 | 2 | 0 | 0 | 0 | 10 | - | 1 | 2 | 3 | 4 | 5 | 6 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 5 | 1 | 5 | 5 | 2 | 0 | 5 | 2 | 5 | 5 | 2 | 0 | 0 | 2 | 0 | - | 1 | 2 | 3 | 4 | 5 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 3 | 2 | 3 | 4 | 5 | 6 | 7 | 4 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | 0 | 1 | 0 | 2 | 0 | 10 | 0 | 0 | 5 | 1 | 10 | 3 | 5 | 6 | 2 | - | 1 | 2 | 3 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 4 | 3 | 2 | 3 | 4 | 5 | 6 | 5 | 4 | 3 | 4 | 5 | 6 | 7 |
| 5 | 10 | 5 | 2 | 3 | 0 | 0 | 2 | 2 | 5 | 2 | 2 | 0 | 1 | 0 | 0 | 0 | - | 1 | 2 | 3 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 4 | 3 | 2 | 3 | 4 | 5 | 6 | 5 | 4 | 3 | 4 | 5 | 6 |
| 0 | 10 | 0 | 2 | 5 | 1 | 4 | 0 | 2 | 10 | 0 | 4 | 5 | 5 | 1 | 5 | 0 | 5 | - | 1 | 2 | 5 | 4 | 3 | 2 | 1 | 2 | 3 | 6 | 5 | 4 | 3 | 2 | 3 | 4 | 7 | 6 | 5 | 4 | 3 | 4 | 5 |
| 2 | 0 | 1 | 6 | 0 | 0 | 5 | 0 | 1 | 0 | 2 | 5 | 6 | 5 | 2 | 5 | 5 | 5 | 1 | - | 1 | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 7 | 6 | 5 | 4 | 3 | 2 | 3 | 8 | 7 | 6 | 5 | 4 | 3 | 4 |
| 5 | 2 | 1 | 5 | 2 | 0 | 6 | 4 | 2 | 0 | 0 | 0 | 0 | 2 | 0 | 2 | 1 | 0 | 0 | 3 | - | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 9 | 8 | 7 | 6 | 5 | 4 | 3 |
| 5 | 1 | 1 | 5 | 2 | 1 | 4 | 0 | 0 | 1 | 5 | 0 | 5 | 2 | 2 | 0 | 1 | 0 | 0 | 1 | 0 | - | 1 | 2 | 3 | 4 | 5 | 6 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | 1 | 5 | 0 | 2 | 2 | 0 | 0 | 0 | 2 | 2 | 0 | 2 | 0 | 0 | 4 | 0 | 2 | 0 | 2 | 2 | 5 | - | 1 | 2 | 3 | 4 | 5 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 3 | 2 | 3 | 4 | 5 | 6 | 7 |
| 10 | 0 | 1 | 5 | 0 | 5 | 2 | 2 | 6 | 4 | 5 | 0 | 1 | 0 | 0 | 0 | 3 | 2 | 3 | 0 | 0 | 5 | 2 | - | 1 | 2 | 3 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 4 | 3 | 2 | 3 | 4 | 5 | 6 |
| 0 | 0 | 2 | 2 | 5 | 5 | 1 | 5 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 5 | 6 | 2 | 2 | 4 | 2 | 0 | 2 | - | 1 | 2 | 3 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 4 | 3 | 2 | 3 | 4 | 5 |
| 1 | 3 | 2 | 1 | 5 | 1 | 2 | 3 | 5 | 0 | 4 | 10 | 5 | 2 | 5 | 0 | 0 | 10 | 10 | 10 | 4 | 5 | 2 | 10 | 1 | - | 1 | 2 | 5 | 4 | 3 | 2 | 1 | 2 | 3 | 6 | 5 | 4 | 3 | 2 | 3 | 4 |
| 4 | 5 | 0 | 0 | 5 | 2 | 1 | 2 | 0 | 5 | 0 | 5 | 2 | 0 | 5 | 0 | 2 | 0 | 4 | 10 | 1 | 2 | 0 | 0 | 4 | 2 | - | 1 | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 7 | 6 | 5 | 4 | 3 | 2 | 3 |
| 2 | 10 | 1 | 0 | 0 | 1 | 2 | 2 | 0 | 0 | 1 | 1 | 1 | 5 | 2 | 0 | 1 | 2 | 5 | 5 | 1 | 10 | 2 | 2 | 0 | 10 | 2 | - | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 8 | 7 | 6 | 5 | 4 | 3 | 2 |
| 0 | 0 | 1 | 1 | 1 | 10 | 2 | 5 | 0 | 3 | 10 | 1 | 0 | 2 | 5 | 2 | 5 | 0 | 5 | 2 | 0 | 0 | 0 | 0 | 5 | 5 | 0 | 3 | - | 1 | 2 | 3 | 4 | 5 | 6 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 5 | 5 | 2 | 5 | 0 | 0 | 2 | 0 | 2 | 3 | 5 | 0 | 4 | 0 | 2 | 1 | 2 | 2 | 1 | 5 | 2 | 5 | 0 | 4 | 0 | 2 | 2 | 0 | - | 1 | 2 | 3 | 4 | 5 | 2 | 1 | 2 | 3 | 4 | 5 | 6 |
| 0 | 0 | 2 | 3 | 0 | 0 | 2 | 2 | 5 | 5 | 5 | 5 | 10 | 5 | 5 | 0 | 5 | 2 | 5 | 2 | 1 | 0 | 2 | 5 | 1 | 1 | 3 | 5 | 0 | 1 | - | 1 | 2 | 3 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 |
| 2 | 2 | 6 | 2 | 4 | 0 | 3 | 2 | 10 | 0 | 5 | 2 | 5 | 10 | 1 | 0 | 6 | 10 | 1 | 1 | 5 | 2 | 0 | 3 | 0 | 5 | 0 | 0 | 0 | 0 | 2 | - | 1 | 2 | 3 | 4 | 3 | 2 | 1 | 2 | 3 | 4 |
| 0 | 1 | 1 | 10 | 2 | 2 | 2 | 5 | 10 | 0 | 0 | 1 | 5 | 4 | 5 | 5 | 0 | 2 | 1 | 0 | 10 | 4 | 2 | 5 | 2 | 0 | 5 | 10 | 3 | 5 | 0 | 2 | - | 1 | 2 | 5 | 4 | 3 | 2 | 1 | 2 | 3 |
| 5 | 1 | 1 | 5 | 2 | 5 | 5 | 0 | 3 | 1 | 5 | 5 | 2 | 0 | 1 | 2 | 3 | 0 | 2 | 10 | 1 | 0 | 1 | 2 | 2 | 0 | 0 | 2 | 6 | 4 | 5 | 4 | 2 | - | 1 | 6 | 5 | 4 | 3 | 2 | 1 | 2 |
| 5 | 3 | 0 | 4 | 1 | 0 | 0 | 2 | 5 | 5 | 2 | 10 | 2 | 10 | 5 | 0 | 5 | 5 | 5 | 2 | 0 | 2 | 5 | 0 | 1 | 10 | 3 | 0 | 5 | 10 | 2 | 1 | 5 | 2 | - | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| 0 | 10 | 10 | 0 | 5 | 10 | 0 | 5 | 1 | 2 | 0 | 5 | 5 | 2 | 0 | 5 | 0 | 1 | 0 | 5 | 0 | 5 | 10 | 10 | 1 | 2 | 2 | 5 | 0 | 5 | 0 | 2 | 0 | 1 | 5 | - | 1 | 2 | 3 | 4 | 5 | 6 |
| 10 | 5 | 5 | 0 | 6 | 5 | 0 | 2 | 2 | 0 | 0 | 3 | 1 | 10 | 5 | 1 | 5 | 0 | 0 | 0 | 5 | 5 | 6 | 10 | 0 | 5 | 0 | 5 | 5 | 1 | 2 | 0 | 0 | 0 | 2 | 0 | - | 1 | 2 | 3 | 4 | 5 |
| 10 | 0 | 10 | 2 | 2 | 4 | 0 | 2 | 0 | 1 | 0 | 5 | 1 | 1 | 5 | 0 | 0 | 6 | 10 | 5 | 10 | 2 | 1 | 3 | 4 | 0 | 2 | 2 | 2 | 0 | 0 | 0 | 0 | 10 | 0 | 2 | 10 | - | 1 | 2 | 3 | 4 |
| 0 | 10 | 5 | 10 | 2 | 5 | 1 | 0 | 0 | 5 | 5 | 5 | 0 | 2 | 2 | 5 | 0 | 3 | 3 | 0 | 10 | 2 | 5 | 0 | 0 | 0 | 10 | 1 | 1 | 10 | 10 | 2 | 2 | 0 | 2 | 5 | 10 | 2 | - | 1 | 2 | 3 |
| 0 | 0 | 5 | 3 | 0 | 5 | 1 | 5 | 10 | 2 | 0 | 1 | 1 | 5 | 10 | 0 | 5 | 4 | 2 | 4 | 4 | 2 | 2 | 2 | 3 | 0 | 0 | 0 | 5 | 5 | 3 | 2 | 2 | 0 | 1 | 0 | 5 | 10 | 1 | - | 1 | 2 |
| 0 | 1 | 2 | 0 | 5 | 2 | 6 | 1 | 1 | 0 | 0 | 5 | 0 | 4 | 5 | 2 | 5 | 5 | 1 | 0 | 5 | 5 | 0 | 1 | 5 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 3 | 0 | 0 | 0 | 2 | - | 1 |
| 2 | 4 | 1 | 5 | 5 | 5 | 1 | 3 | 6 | 10 | 5 | 5 | 0 | 1 | 5 | 0 | 10 | 5 | 10 | 2 | 2 | 0 | 0 | 0 | 5 | 1 | 2 | 0 | 2 | 3 | 0 | 6 | 0 | 10 | 0 | 5 | 1 | 0 | 4 | 1 | 5 | - |

Sko64

| - | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | - | 1 | 2 | 3 | 4 | 5 | 6 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 3 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 4 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 5 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 6 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 7 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 8 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| 0 | 0 | - | 1 | 2 | 3 | 4 | 5 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 4 | 3 | 2 | 3 | 4 | 5 | 6 | 7 | 5 | 4 | 3 | 4 | 5 | 6 | 7 | 8 | 6 | 5 | 4 | 5 | 6 | 7 | 8 | 9 | 7 | 6 | 5 | 6 | 7 | 8 | 9 | 10 | 8 | 7 | 6 | 7 | 8 | 9 | 10 | 11 | 9 | 8 | 7 | 8 | 9 | 10 | 11 | 12 |
| 0 | 2 | 1 | - | 1 | 2 | 3 | 4 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 5 | 4 | 3 | 2 | 3 | 4 | 5 | 6 | 6 | 5 | 4 | 3 | 4 | 5 | 6 | 7 | 7 | 6 | 5 | 4 | 5 | 6 | 7 | 8 | 8 | 7 | 6 | 5 | 6 | 7 | 8 | 9 | 9 | 8 | 7 | 6 | 7 | 8 | 9 | 10 | 10 | 9 | 8 | 7 | 8 | 9 | 10 | 11 |
| 6 | 0 | 0 | 0 | - | 1 | 2 | 3 | 5 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 6 | 5 | 4 | 3 | 2 | 3 | 4 | 5 | 7 | 6 | 5 | 4 | 3 | 4 | 5 | 6 | 8 | 7 | 6 | 5 | 4 | 5 | 6 | 7 | 9 | 8 | 7 | 6 | 5 | 6 | 7 | 8 | 10 | 9 | 8 | 7 | 6 | 7 | 8 | 9 | 11 | 10 | 9 | 8 | 7 | 8 | 9 | 10 |
| 0 | 5 | 5 | 2 | 10 | - | 1 | 2 | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 3 | 7 | 6 | 5 | 4 | 3 | 2 | 3 | 4 | 8 | 7 | 6 | 5 | 4 | 3 | 4 | 5 | 9 | 8 | 7 | 6 | 5 | 4 | 5 | 6 | 10 | 9 | 8 | 7 | 6 | 5 | 6 | 7 | 11 | 10 | 9 | 8 | 7 | 6 | 7 | 8 | 12 | 11 | 10 | 9 | 8 | 7 | 8 | 9 |
| 4 | 1 | 5 | 4 | 0 | 0 | - | 1 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 3 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 4 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 5 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 6 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 7 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 8 |
| 4 | 10 | 2 | 3 | 2 | 0 | 2 | - | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 |
| 4 | 5 | 2 | 1 | 2 | 2 | 1 | 0 | - | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| 0 | 0 | 0 | 0 | 1 | 4 | 0 | 2 | 5 | - | 1 | 2 | 3 | 4 | 5 | 6 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 3 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 4 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 5 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 6 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 7 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 2 | 0 | 0 | 2 | 0 | 5 | 5 | 6 | 2 | 5 | - | 1 | 2 | 3 | 4 | 5 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 4 | 3 | 2 | 3 | 4 | 5 | 6 | 7 | 5 | 4 | 3 | 4 | 5 | 6 | 7 | 8 | 6 | 5 | 4 | 5 | 6 | 7 | 8 | 9 | 7 | 6 | 5 | 6 | 7 | 8 | 9 | 10 | 8 | 7 | 6 | 7 | 8 | 9 | 10 | 11 |
| 0 | 0 | 0 | 6 | 5 | 5 | 0 | 0 | 0 | 0 | 5 | - | 1 | 2 | 3 | 4 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 5 | 4 | 3 | 2 | 3 | 4 | 5 | 6 | 6 | 5 | 4 | 3 | 4 | 5 | 6 | 7 | 7 | 6 | 5 | 4 | 5 | 6 | 7 | 8 | 8 | 7 | 6 | 5 | 6 | 7 | 8 | 9 | 9 | 8 | 7 | 6 | 7 | 8 | 9 | 10 |
| 5 | 5 | 1 | 5 | 0 | 5 | 0 | 2 | 5 | 3 | 2 | 4 | - | 1 | 2 | 3 | 5 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 6 | 5 | 4 | 3 | 2 | 3 | 4 | 5 | 7 | 6 | 5 | 4 | 3 | 4 | 5 | 6 | 8 | 7 | 6 | 5 | 4 | 5 | 6 | 7 | 9 | 8 | 7 | 6 | 5 | 6 | 7 | 8 | 10 | 9 | 8 | 7 | 6 | 7 | 8 | 9 |
| 5 | 0 | 0 | 1 | 2 | 0 | 2 | 5 | 5 | 2 | 1 | 5 | 2 | - | 1 | 2 | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 3 | 7 | 6 | 5 | 4 | 3 | 2 | 3 | 4 | 8 | 7 | 6 | 5 | 4 | 3 | 4 | 5 | 9 | 8 | 7 | 6 | 5 | 4 | 5 | 6 | 10 | 9 | 8 | 7 | 6 | 5 | 6 | 7 | 11 | 10 | 9 | 8 | 7 | 6 | 7 | 8 |
| 2 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 2 | 0 | - | 1 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 3 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 4 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 5 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 6 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 7 |
| 2 | 0 | 5 | 5 | 10 | 2 | 10 | 0 | 1 | 5 | 0 | 5 | 2 | 2 | 6 | - | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 |
| 5 | 6 | 2 | 2 | 4 | 2 | 0 | 2 | 3 | 2 | 1 | 0 | 2 | 5 | 3 | 6 | - | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 4 | 2 | 0 | 2 | 4 | 5 | 0 | 0 | 5 | 2 | 1 | 2 | 0 | 5 | 0 | 5 | 2 | - | 1 | 2 | 3 | 4 | 5 | 6 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 3 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 4 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 5 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 6 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| 5 | 5 | 1 | 10 | 2 | 2 | 0 | 10 | 2 | 5 | 4 | 0 | 0 | 0 | 2 | 4 | 6 | 1 | - | 1 | 2 | 3 | 4 | 5 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 4 | 3 | 2 | 3 | 4 | 5 | 6 | 7 | 5 | 4 | 3 | 4 | 5 | 6 | 7 | 8 | 6 | 5 | 4 | 5 | 6 | 7 | 8 | 9 | 7 | 6 | 5 | 6 | 7 | 8 | 9 | 10 |
| 4 | 3 | 0 | 5 | 5 | 2 | 5 | 0 | 0 | 2 | 0 | 2 | 3 | 5 | 0 | 4 | 0 | 2 | 1 | - | 1 | 2 | 3 | 4 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 5 | 4 | 3 | 2 | 3 | 4 | 5 | 6 | 6 | 5 | 4 | 3 | 4 | 5 | 6 | 7 | 7 | 6 | 5 | 4 | 5 | 6 | 7 | 8 | 8 | 7 | 6 | 5 | 6 | 7 | 8 | 9 |
| 1 | 0 | 2 | 5 | 1 | 1 | 3 | 5 | 0 | 1 | 2 | 0 | 1 | 10 | 5 | 5 | 10 | 0 | 0 | 0 | - | 1 | 2 | 3 | 5 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 6 | 5 | 4 | 3 | 2 | 3 | 4 | 5 | 7 | 6 | 5 | 4 | 3 | 4 | 5 | 6 | 8 | 7 | 6 | 5 | 4 | 5 | 6 | 7 | 9 | 8 | 7 | 6 | 5 | 6 | 7 | 8 |
| 0 | 1 | 1 | 10 | 2 | 2 | 2 | 5 | 10 | 0 | 0 | 1 | 5 | 4 | 5 | 5 | 0 | 2 | 1 | 0 | 10 | - | 1 | 2 | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 3 | 7 | 6 | 5 | 4 | 3 | 2 | 3 | 4 | 8 | 7 | 6 | 5 | 4 | 3 | 4 | 5 | 9 | 8 | 7 | 6 | 5 | 4 | 5 | 6 | 10 | 9 | 8 | 7 | 6 | 5 | 6 | 7 |
| 1 | 2 | 2 | 0 | 0 | 2 | 6 | 4 | 5 | 4 | 2 | 10 | 5 | 0 | 5 | 10 | 0 | 0 | 0 | 6 | 5 | 3 | - | 1 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 3 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 4 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 5 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 6 |
| 10 | 0 | 5 | 10 | 0 | 5 | 1 | 2 | 0 | 5 | 5 | 2 | 0 | 5 | 0 | 1 | 0 | 5 | 0 | 5 | 10 | 10 | 1 | - | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 |
| 0 | 5 | 0 | 5 | 5 | 1 | 2 | 0 | 0 | 0 | 2 | 0 | 5 | 2 | 2 | 2 | 10 | 1 | 10 | 0 | 10 | 2 | 2 | 4 | - | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| 2 | 5 | 1 | 0 | 0 | 5 | 5 | 5 | 0 | 2 | 2 | 5 | 0 | 3 | 3 | 0 | 10 | 2 | 5 | 0 | 0 | 0 | 10 | 1 | 1 | - | 1 | 2 | 3 | 4 | 5 | 6 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 3 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 4 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 5 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 0 | 0 | 5 | 5 | 3 | 2 | 2 | 0 | 1 | 0 | 5 | 10 | 1 | 0 | 5 | 2 | 0 | 1 | 2 | 0 | 5 | 2 | 6 | 1 | 1 | 0 | - | 1 | 2 | 3 | 4 | 5 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 4 | 3 | 2 | 3 | 4 | 5 | 6 | 7 | 5 | 4 | 3 | 4 | 5 | 6 | 7 | 8 | 6 | 5 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 3 | 6 | 10 | 5 | 5 | 0 | 1 | 5 | 0 | 10 | 5 | 10 | 2 | 2 | 0 | 0 | 0 | 5 | 1 | 2 | 0 | 2 | 3 | 0 | 6 | 0 | - | 1 | 2 | 3 | 4 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 5 | 4 | 3 | 2 | 3 | 4 | 5 | 6 | 6 | 5 | 4 | 3 | 4 | 5 | 6 | 7 | 7 | 6 | 5 | 4 | 5 | 6 | 7 | 8 |
| 2 | 3 | 0 | 1 | 10 | 1 | 2 | 0 | 10 | 5 | 0 | 5 | 2 | 2 | 0 | 4 | 4 | 3 | 5 | 0 | 0 | 5 | 5 | 5 | 2 | 2 | 2 | 0 | - | 1 | 2 | 3 | 5 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 6 | 5 | 4 | 3 | 2 | 3 | 4 | 5 | 7 | 6 | 5 | 4 | 3 | 4 | 5 | 6 | 8 | 7 | 6 | 5 | 4 | 5 | 6 | 7 |
| 1 | 3 | 0 | 2 | 6 | 0 | 0 | 2 | 0 | 0 | 5 | 5 | 1 | 5 | 2 | 0 | 5 | 4 | 1 | 1 | 0 | 5 | 3 | 2 | 0 | 0 | 5 | 0 | 2 | - | 1 | 2 | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 3 | 7 | 6 | 5 | 4 | 3 | 2 | 3 | 4 | 8 | 7 | 6 | 5 | 4 | 3 | 4 | 5 | 9 | 8 | 7 | 6 | 5 | 4 | 5 | 6 |
| 1 | 10 | 10 | 5 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 10 | 0 | 2 | 5 | 1 | 5 | 0 | 1 | 0 | 5 | 2 | 1 | 4 | 5 | 5 | 0 | 0 | 0 | - | 1 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 3 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 4 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 5 |
| 2 | 2 | 5 | 2 | 10 | 2 | 5 | 5 | 1 | 0 | 10 | 0 | 2 | 0 | 0 | 0 | 4 | 5 | 4 | 2 | 0 | 1 | 5 | 1 | 0 | 5 | 0 | 6 | 2 | 3 | 0 | - | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 |
| 1 | 10 | 0 | 1 | 3 | 0 | 5 | 5 | 1 | 2 | 5 | 2 | 2 | 0 | 0 | 10 | 3 | 0 | 0 | 5 | 5 | 5 | 5 | 5 | 0 | 5 | 5 | 0 | 2 | 10 | 2 | 0 | - | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 5 | 1 | 10 | 5 | 0 | 1 | 5 | 0 | 0 | 0 | 2 | 10 | 5 | 5 | 4 | 0 | 4 | 2 | 1 | 1 | 2 | 0 | 0 | 1 | 3 | 5 | 0 | 5 | 0 | 0 | 0 | 0 | 5 | - | 1 | 2 | 3 | 4 | 5 | 6 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 3 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 4 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 6 | 5 | 1 | 2 | 5 | 2 | 5 | 0 | 5 | 1 | 0 | 0 | 5 | 2 | 2 | 1 | 10 | 10 | 2 | 0 | 10 | 2 | 5 | 0 | 2 | 5 | 1 | 2 | 0 | 0 | 5 | 0 | 0 | 0 | - | 1 | 2 | 3 | 4 | 5 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 4 | 3 | 2 | 3 | 4 | 5 | 6 | 7 | 5 | 4 | 3 | 4 | 5 | 6 | 7 | 8 |
| 2 | 0 | 3 | 2 | 4 | 1 | 2 | 3 | 1 | 0 | 1 | 5 | 2 | 10 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 5 | 1 | 0 | 0 | 2 | 6 | 5 | 4 | 0 | 2 | 0 | 0 | 5 | 0 | - | 1 | 2 | 3 | 4 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 5 | 4 | 3 | 2 | 3 | 4 | 5 | 6 | 6 | 5 | 4 | 3 | 4 | 5 | 6 | 7 |
| 10 | 2 | 1 | 5 | 5 | 5 | 4 | 1 | 3 | 2 | 2 | 0 | 1 | 0 | 2 | 2 | 5 | 0 | 0 | 1 | 1 | 10 | 5 | 2 | 5 | 3 | 1 | 3 | 3 | 0 | 0 | 0 | 5 | 10 | 0 | 0 | - | 1 | 2 | 3 | 5 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 6 | 5 | 4 | 3 | 2 | 3 | 4 | 5 | 7 | 6 | 5 | 4 | 3 | 4 | 5 | 6 |
| 1 | 3 | 0 | 0 | 0 | 5 | 0 | 2 | 0 | 10 | 5 | 2 | 3 | 0 | 10 | 2 | 10 | 2 | 4 | 2 | 1 | 0 | 0 | 5 | 2 | 0 | 5 | 1 | 0 | 2 | 2 | 10 | 3 | 5 | 0 | 0 | 5 | - | 1 | 2 | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 3 | 7 | 6 | 5 | 4 | 3 | 2 | 3 | 4 | 8 | 7 | 6 | 5 | 4 | 3 | 4 | 5 |
| 5 | 0 | 5 | 5 | 10 | 0 | 10 | 2 | 0 | 3 | 4 | 5 | 0 | 1 | 0 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | 0 | 5 | 1 | 5 | 4 | 5 | 0 | 4 | 0 | 0 | 5 | 5 | 0 | 4 | 5 | 10 | - | 1 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 3 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 4 |
| 0 | 0 | 6 | 4 | 5 | 4 | 2 | 2 | 2 | 0 | 10 | 0 | 0 | 5 | 0 | 10 | 2 | 5 | 3 | 4 | 0 | 2 | 10 | 2 | 2 | 6 | 5 | 10 | 2 | 10 | 0 | 3 | 1 | 2 | 1 | 3 | 1 | 2 | 10 | - | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 |
| 5 | 0 | 5 | 5 | 10 | 1 | 5 | 2 | 0 | 5 | 10 | 5 | 10 | 0 | 10 | 5 | 5 | 3 | 5 | 0 | 5 | 0 | 2 | 2 | 0 | 2 | 0 | 1 | 0 | 5 | 0 | 5 | 1 | 10 | 5 | 5 | 0 | 0 | 1 | 0 | - | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 2 | 1 | 5 | 10 | 1 | 2 | 0 | 2 | 10 | 2 | 0 | 5 | 10 | 0 | 0 | 5 | 0 | 1 | 1 | 10 | 4 | 2 | 0 | 2 | 10 | 5 | 0 | 5 | 0 | 0 | 5 | 2 | 2 | 0 | 0 | 0 | 5 | 0 | 0 | 5 | 6 | - | 1 | 2 | 3 | 4 | 5 | 6 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 3 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 5 | 0 | 10 | 0 | 0 | 5 | 1 | 0 | 0 | 6 | 10 | 0 | 0 | 10 | 2 | 2 | 2 | 5 | 6 | 0 | 1 | 5 | 0 | 1 | 0 | 0 | 10 | 5 | 1 | 6 | 0 | 0 | 5 | 0 | 0 | 1 | 0 | 0 | 2 | 5 | 2 | 6 | - | 1 | 2 | 3 | 4 | 5 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 4 | 3 | 2 | 3 | 4 | 5 | 6 | 7 |
| 10 | 10 | 5 | 5 | 0 | 0 | 1 | 5 | 0 | 0 | 1 | 5 | 0 | 5 | 1 | 10 | 5 | 2 | 2 | 5 | 1 | 0 | 3 | 5 | 5 | 6 | 0 | 1 | 2 | 3 | 2 | 2 | 5 | 2 | 2 | 0 | 3 | 5 | 2 | 2 | 0 | 1 | 0 | - | 1 | 2 | 3 | 4 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 5 | 4 | 3 | 2 | 3 | 4 | 5 | 6 |
| 10 | 1 | 2 | 0 | 2 | 1 | 10 | 0 | 0 | 2 | 0 | 0 | 5 | 2 | 0 | 1 | 1 | 2 | 1 | 10 | 5 | 1 | 0 | 2 | 10 | 2 | 1 | 0 | 1 | 6 | 0 | 2 | 5 | 6 | 10 | 3 | 2 | 4 | 0 | 10 | 2 | 0 | 5 | 2 | - | 1 | 2 | 3 | 5 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 6 | 5 | 4 | 3 | 2 | 3 | 4 | 5 |
| 3 | 1 | 5 | 0 | 5 | 3 | 0 | 0 | 1 | 0 | 2 | 1 | 5 | 0 | 4 | 10 | 0 | 2 | 0 | 10 | 0 | 0 | 2 | 5 | 0 | 5 | 5 | 2 | 5 | 0 | 0 | 10 | 1 | 0 | 0 | 10 | 0 | 2 | 0 | 5 | 0 | 2 | 2 | 5 | 1 | - | 1 | 2 | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 3 | 7 | 6 | 5 | 4 | 3 | 2 | 3 | 4 |
| 5 | 6 | 1 | 10 | 5 | 10 | 0 | 5 | 0 | 5 | 0 | 0 | 5 | 0 | 0 | 0 | 4 | 6 | 2 | 2 | 1 | 0 | 1 | 2 | 0 | 0 | 0 | 1 | 2 | 5 | 0 | 5 | 0 | 2 | 1 | 0 | 0 | 5 | 5 | 1 | 2 | 0 | 10 | 0 | 3 | 0 | - | 1 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 3 |
| 3 | 2 | 2 | 0 | 3 | 5 | 0 | 5 | 0 | 0 | 4 | 0 | 1 | 4 | 0 | 1 | 1 | 0 | 10 | 10 | 3 | 10 | 0 | 5 | 2 | 1 | 4 | 5 | 5 | 5 | 1 | 5 | 0 | 2 | 5 | 1 | 2 | 2 | 4 | 10 | 0 | 2 | 2 | 2 | 3 | 1 | 0 | - | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 |
| 0 | 5 | 2 | 5 | 1 | 1 | 2 | 0 | 1 | 5 | 2 | 5 | 10 | 1 | 0 | 10 | 5 | 0 | 5 | 5 | 10 | 0 | 1 | 5 | 0 | 2 | 1 | 0 | 5 | 2 | 2 | 5 | 1 | 2 | 0 | 5 | 1 | 5 | 0 | 0 | 2 | 0 | 4 | 5 | 1 | 1 | 5 | 0 | - | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 5 | 5 | 0 | 1 | 0 | 4 | 3 | 2 | 2 | 0 | 2 | 2 | 1 | 5 | 2 | 0 | 5 | 1 | 2 | 1 | 1 | 0 | 5 | 2 | 3 | 5 | 2 | 0 | 2 | 0 | 0 | 1 | 5 | 2 | 5 | 0 | 10 | 1 | 0 | 1 | 1 | 5 | 10 | 5 | 0 | 2 | 5 | 1 | 0 | - | 1 | 2 | 3 | 4 | 5 | 6 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 1 | 5 | 1 | 2 | 2 | 0 | 10 | 0 | 0 | 5 | 1 | 2 | 0 | 2 | 5 | 0 | 0 | 2 | 5 | 2 | 5 | 5 | 0 | 1 | 0 | 0 | 4 | 2 | 0 | 10 | 0 | 0 | 0 | 0 | 1 | 2 | 6 | 5 | 5 | 1 | 0 | 6 | 5 | 5 | 10 | 10 | 0 | 1 | 5 | - | 1 | 2 | 3 | 4 | 5 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 6 |
| 3 | 2 | 2 | 5 | 5 | 5 | 2 | 2 | 10 | 0 | 4 | 5 | 6 | 10 | 0 | 2 | 2 | 5 | 0 | 0 | 5 | 1 | 10 | 5 | 0 | 0 | 5 | 2 | 4 | 0 | 2 | 1 | 0 | 1 | 3 | 5 | 0 | 10 | 2 | 1 | 0 | 5 | 0 | 2 | 5 | 5 | 1 | 1 | 6 | 2 | 5 | - | 1 | 2 | 3 | 4 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 |
| 1 | 0 | 1 | 4 | 2 | 5 | 2 | 5 | 1 | 10 | 10 | 0 | 1 | 1 | 6 | 10 | 4 | 0 | 5 | 0 | 5 | 0 | 1 | 4 | 5 | 3 | 5 | 5 | 2 | 0 | 0 | 5 | 2 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6 | 0 | 5 | 10 | 2 | 4 | 2 | 1 | 2 | 0 | 0 | - | 1 | 2 | 3 | 5 | 4 | 3 | 2 | 1 | 2 | 3 | 4 |
| 2 | 5 | 5 | 0 | 6 | 4 | 6 | 0 | 0 | 5 | 10 | 0 | 1 | 0 | 10 | 0 | 5 | 2 | 0 | 5 | 1 | 4 | 2 | 5 | 5 | 1 | 0 | 5 | 5 | 5 | 5 | 10 | 0 | 10 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 3 | 0 | 1 | 0 | 5 | 5 | 3 | 1 | 5 | 2 | 0 | - | 1 | 2 | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 3 |
| 4 | 5 | 5 | 0 | 2 | 2 | 10 | 0 | 5 | 5 | 0 | 0 | 2 | 4 | 5 | 5 | 0 | 0 | 3 | 5 | 5 | 2 | 5 | 0 | 2 | 0 | 2 | 5 | 10 | 0 | 0 | 0 | 5 | 0 | 6 | 2 | 5 | 10 | 10 | 2 | 2 | 3 | 10 | 4 | 0 | 5 | 5 | 10 | 0 | 0 | 5 | 0 | 2 | 0 | - | 1 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 2 |
| 5 | 3 | 0 | 10 | 0 | 0 | 0 | 1 | 0 | 1 | 2 | 0 | 2 | 2 | 2 | 4 | 10 | 5 | 0 | 0 | 2 | 0 | 2 | 3 | 5 | 0 | 5 | 0 | 5 | 5 | 5 | 0 | 6 | 0 | 2 | 0 | 2 | 0 | 5 | 0 | 2 | 0 | 2 | 5 | 10 | 5 | 0 | 0 | 5 | 3 | 5 | 2 | 1 | 5 | 0 | - | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| 0 | 10 | 5 | 1 | 0 | 1 | 10 | 3 | 2 | 2 | 4 | 0 | 0 | 0 | 5 | 3 | 1 | 2 | 0 | 0 | 0 | 4 | 2 | 5 | 5 | 0 | 1 | 0 | 1 | 5 | 10 | 3 | 1 | 0 | 2 | 2 | 5 | 0 | 2 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 2 | 1 | 0 | 0 | 0 | 5 | 10 | 10 | 0 | 2 | - | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 1 | 0 | 2 | 5 | 2 | 2 | 10 | 10 | 10 | 0 | 0 | 10 | 5 | 0 | 5 | 1 | 2 | 0 | 2 | 0 | 5 | 0 | 5 | 10 | 10 | 5 | 0 | 5 | 2 | 0 | 0 | 0 | 4 | 2 | 4 | 5 | 0 | 1 | 3 | 1 | 0 | 0 | 10 | 6 | 0 | 5 | 5 | 2 | 6 | 0 | 5 | 1 | 4 | 5 | 5 | 2 | - | 1 | 2 | 3 | 4 | 5 | 6 |
| 2 | 10 | 0 | 0 | 0 | 0 | 10 | 0 | 1 | 2 | 5 | 0 | 10 | 5 | 0 | 4 | 5 | 0 | 2 | 1 | 10 | 0 | 2 | 10 | 0 | 2 | 2 | 0 | 1 | 5 | 1 | 1 | 2 | 2 | 5 | 1 | 3 | 10 | 1 | 2 | 0 | 1 | 1 | 5 | 2 | 0 | 0 | 0 | 0 | 6 | 2 | 5 | 2 | 6 | 0 | 1 | 0 | 0 | - | 1 | 2 | 3 | 4 | 5 |
| 0 | 0 | 5 | 0 | 4 | 1 | 5 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 5 | 10 | 1 | 6 | 3 | 0 | 0 | 4 | 5 | 2 | 0 | 5 | 10 | 1 | 0 | 4 | 0 | 5 | 5 | 5 | 6 | 2 | 0 | 10 | 0 | 1 | 2 | 0 | 5 | 5 | 0 | 10 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 5 | 2 | 2 | 3 | - | 1 | 2 | 3 | 4 |
| 0 | 2 | 10 | 5 | 2 | 2 | 0 | 2 | 5 | 0 | 0 | 2 | 0 | 1 | 5 | 2 | 2 | 0 | 0 | 2 | 5 | 5 | 0 | 10 | 0 | 0 | 5 | 1 | 0 | 5 | 1 | 1 | 0 | 0 | 0 | 5 | 2 | 0 | 2 | 0 | 5 | 0 | 1 | 5 | 3 | 5 | 1 | 1 | 2 | 0 | 5 | 10 | 1 | 3 | 4 | 0 | 10 | 10 | 2 | 1 | - | 1 | 2 | 3 |
| 3 | 0 | 5 | 5 | 5 | 10 | 5 | 5 | 2 | 1 | 1 | 2 | 0 | 2 | 2 | 6 | 5 | 2 | 5 | 5 | 5 | 10 | 1 | 5 | 5 | 2 | 0 | 10 | 1 | 2 | 0 | 2 | 2 | 1 | 2 | 2 | 0 | 3 | 4 | 5 | 0 | 0 | 1 | 2 | 0 | 1 | 1 | 1 | 0 | 1 | 2 | 1 | 3 | 3 | 2 | 0 | 1 | 0 | 10 | 0 | 0 | - | 1 | 2 |
| 5 | 10 | 0 | 2 | 1 | 5 | 5 | 4 | 0 | 10 | 0 | 1 | 0 | 1 | 0 | 2 | 1 | 10 | 5 | 4 | 4 | 5 | 0 | 1 | 3 | 5 | 2 | 1 | 0 | 3 | 5 | 0 | 0 | 6 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 5 | 2 | 1 | 1 | 5 | 0 | 0 | 10 | 2 | 5 | 5 | 0 | 0 | 5 | 1 | 2 | 5 | 1 | 10 | 0 | 2 | - | 1 |
| 1 | 3 | 0 | 10 | 5 | 5 | 1 | 2 | 0 | 2 | 1 | 0 | 10 | 0 | 1 | 5 | 5 | 0 | 5 | 0 | 4 | 0 | 2 | 2 | 0 | 5 | 0 | 0 | 5 | 10 | 0 | 2 | 0 | 0 | 0 | 10 | 0 | 4 | 5 | 0 | 1 | 10 | 0 | 10 | 10 | 2 | 2 | 0 | 1 | 0 | 5 | 5 | 2 | 1 | 10 | 2 | 3 | 0 | 0 | 0 | 0 | 5 | 5 | - |

Sko81

# BIBLIGRAFÍA


[1] R.E. Burkard, S.E. Karisch ,and F. Rendl, **QAPLIB - A Quadratic Assignment Problem**, Library, *internet*, http://www.imm.dtu.dk/~sk/qaplib/ins.html.

[2] K. M. Chandy, S.Taylor. **An introduction parallel programming**. *Janes and Bartlet publshers Boston*, 1992.

[3] E.G. Coffman,J.K. Lenstra y H.H.G. Rinnooy. **Handbooks in operations research and management science.** *Vol. 3 computing ORSA.*

[4] S.A. Cook . **The complexity of theorem-proving procedures**. *Proceedings of the third annual ACM symposium on the theory of computing*. Asociation for Computing Machinery,1971.

[5] A.D. Díaz, F. Glober, H. M.Ghaziri, J.L. Gonzalez, P. Moscato, F.T. Tseng. **Optimización Heurística y Redes Neuronales en Dirección de Operaciones e Ingeniería**. Editorial Paraninfo,1996.

[6] T.A. Feo and J.L. Gonzalez Velarde.**The intermodal Trailer Assignment Problem.** *Transportation Science*, vol. 29: pp330-341,1995.

[7] T.A. Feo and M.G.C. Resende. **A probabilstic Heuristic for a Computacionally Dificult Set Covering Problem**. *Operations Research Letters*, vol. 8, pp 67 - 71,1989.

[8] T.A. Feo and M.G.C. Resende. **Greedy Randomized Adaptative Search Procedures**. *Journal of Global Optimization*, vol 6, pp109-133, 1995.

[9] P. Festa and M.G. C. Resende. **GRASP: An annotated Bibliography**. To appear in *Ensays and Surveys on Metaheuristics*. P. Hansen and C.C. Riveiro, eds., Kluwer Academic Publihers, 2000.

[10] I. Flores de la Mota. **Apuntes de Programación Entera**. *Departamento de Ingeniería de Sistemas, DEPFI*, 1993.

[11] J. Galve, J. C. González, A. Sánchez, J. A. Velásquez. **Algorítmica; Diseño y análisis de algoritmos funcionales e imperativos.** *Addison-Wesley Iberoamericana*, 1993.

[12] P.J. Hatcher, M.J. Quinn. **Data-Parallel Programming on MIMD Computers**. *Scientific and Enginering Computation*. Juanus Kowalik, editor,1991.

[13] T.C. Koopmans,and M.J. Beckmann. **Assignment problems and the location of economic activities.** *Econometrica*, vol 25: pp 53-76,1957.

[14] E.L. Lawler. **The Quadratic Assignment Problem**. *Managnament Sci.*, vol. 9, pp 586-599,1963.

[15] J.K. Lenstra, A.H.G. Rinnooy Kan, P. Emde Boas. **An Appraisal of computational complexity for operations research**. *European Journal of Operational Research*,vol 11, 1982

[16] T. G. Lewis, H. El-Rewini. **Introdiction to parallel computing.** *Prentice Hall*,1992

[17] Y. Li, P.M. Pardalos, M.G.C. Resende. **A Greedy Randomized Adaptative Search Procedure for the Quadratic Assignment Problem**. In P.M. Pardalos and H. Wolkowicz, editors, *Quadratic assignment and related problems*, vol. 16 of DIMACS *Series on Discrete Mathematics and Theoretical Computer Sience*, pp 237-261. American Mathematical Society, 1994.

[18] C.E. Nugent, T.E. Vollman, and J. Ruml. **An Experimental Comparison of Tecniques for the Assignment of Facilities to locations.** *Journal of Operations Research,* vol. 16 : pp 150-173, 1969.

[19] P.M. Pardalos and Crouse. **A parallel algorithm for the quadratic assignment problem.** *In Proceceedings of the supercomputing 1989 Conference,* pp 351-360, ACM Press,1989.

[20] P.M. Pardalos, L.S. Pitssoulis, and M.G.C. Resende. **A Parallel GRASP implemetation for the Quadratic Assignment problem.** In A. Ferreira and J. Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems – Irregular'94,* pp 111-130. Klower Academic Publishers, 1995.

[21] J. Prawda. **Métodos y Modelos de Investigación de Operaciones.** *LIMUSA.*1ª *Edición,*1991.

[22] R.L. Rardin. **Optimization in Operatins Research.** *Prentice Hall*.1998.

[23] C. R. Reeves. **Modern Heuritic Tecniques for Combinatorial Problems.** *John Wiley & Sons, N. Y.*, 1993.

[24] M.G.C. Resende. **Greedy Randomized Adaptative Search Procedure.** Technical report, AT&T Labs Research, FlorhamPark, NJ 07932 USA,1998.

[25] M.G.R. Resende, K.G. Ramakrishnan, Z. Drenzner. **Computing lower bounds for the quadratic assignment problem. with an interior point algorithm for linear programming.** *Operations Research,* vol 43,N° 5, september-october1995.

[26] M. G. C. Resende, P. M. Pardalos, Yong Li. **Algorithm 754: Fortran subroutines for aproximate solution of dense quadratic assignment problem using GRASP.** *ACM Transactions on mathematical software,* vol 22, num. 1, pp 104-118, march 1996.

[27] C. Roucairol. **A parallel branch and baund algorithm for the quadratic assignment problem.** *Discrete Applied Mathematics,* vol. 18, pp 211-225, 1987.

[28] S. Sahni and T. Gonzalez. **P-complete aproximations problems.** *J. Asssoc. Comp. Machine.* vol. 23 , pp 555-565, 1976.

[29] J. Skorin-Kapov. **Tabu Apllied to the Quadratic Assignment Problem.***ORSA Journal on Computing 2* ,vol..1, pp 33- 45, 1990.