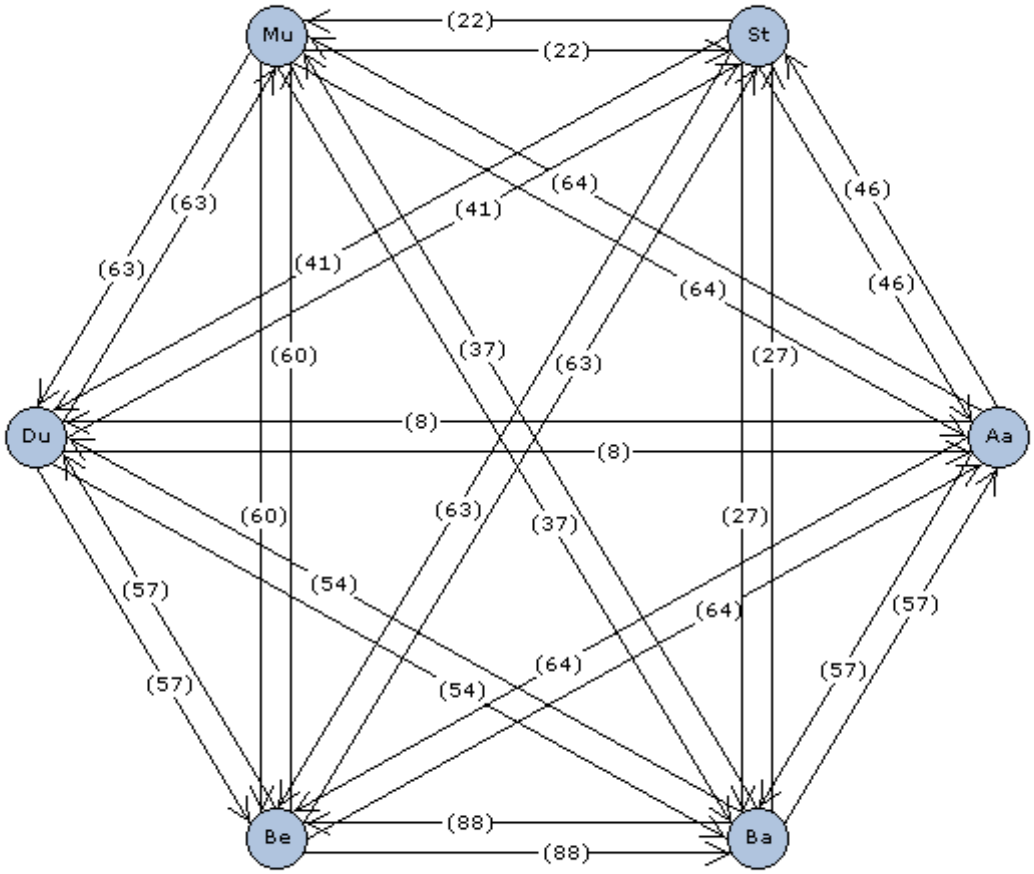
Trabajo Práctico N^{ro} 1 : Traveling Salesman Problem



Materia: Introducción a la Inteligencia Artificial – UTN FRBA

Profesor: Dr. Claudio Verrastro

Ayudante: Ing. Juan Carlos Gómez

Alumno: Gustavo Damián Gil

Fecha: 25-06-07

Índice de contenidos:

Enunciado del TSP y primeras conclusiones 3

Marco teórico y explicación del algoritmo del programa 4

Detalle del programa TSP12.C 5

Anexos:

¿Que es un Grafo? 8

ALGORITMO DE DIJKSTRA 9

Algoritmo de Floyd-Warshall (todos los caminos mínimos) 12

Bibliografía de estos algoritmos 13

Eficiencia computacional de algoritmos usando MATLAB 14

ENUNCIADO DEL TSP (Traveling Salesman Problem):

Una persona tiene que recorrer un cierto número de ciudades que están todas interconectadas con todas. Es decir, siempre se puede ir de una hacia otra en cualquier dirección. Otro dato que se tiene, es cuánto sale ir de una a otra. A los efectos prácticos, vamos a suponer que viajar desde la ciudad 1 hasta la ciudad 2, sale lo mismo que viajar desde 2 hasta 1.
El problema consiste en construir un itinerario que pase por todas las ciudades una sola vez, y que termine en el mismo lugar inicial, pero con la particularidad que sea el más barato, costo mínimo.

Después de trabajar en la resolución de éste problema, he podido constatar lo mencionado en clase referente a que el TSP, es un problema de complejidad NP-completo.
Esto significa que el número de posibles soluciones, crece exponencialmente con el número de nodos del grafo (ciudades), y rápidamente sobrepasa las capacidades de cálculo de las computadoras mas potentes.

En los anexos de éste informe, podrán encontrar información teórica con respecto a grafos, ya que es necesaria para comprender el funcionamiento del programa realizado en lenguaje C.
También encontrarán información recopilada de algoritmos de probada eficiencia como el Dijkstra o Floyd-Warshall, que resuelven la situación planteada en el TSP pero realizando algunas restricciones debido a algún conocimiento previo (heurística), que producen que deje de ser un problema de expansión combinatoria. Éstos algoritmos son empleados en software de uso corriente en labores de logística y distribución.
Finalmente encontraran información referente a la optimización de software, utilizando una herramienta de depuración del programa Matlab, ya que mi idea inicial era realizar mi algoritmo en dicha plataforma, pero me encontré limitado por mis escasos conocimientos de la misma, y radicalmente cuando me topé con la limitación de que dicho lenguaje no está pensado para trabajar con grafos, razón por la cuál debía realizar archivos DLL programados para tal finalidad.

Marco teórico y explicación del algoritmo del programa:

El comportamiento de la lógica del algoritmo, está basado en el análisis de una función de evaluación que tiene la siguiente forma: $\mathbf{f_n=h_n+g_n}$
Siendo **n** el estado parcial de la solución en el que nos encontramos (en nuestro caso las ciudades que llevamos recorridas, por ejemplo: n=1 3 7 , significa que el recorrido realizado hasta éste estado es ciudad 1, cuidad 3 y ciudad 7).
La variable **g**: es el costo inter-ciudad acumulado hasta el estado actual, por ejemplo: si nuestros costos no euclidianos puestos en juego son, g13=12unidades, g17=5unidades y g37=9unidades, nuestro costo acumulado en el estado 1 3 7 resulta:

g137=g13+g37
g137=12+9=21unidades

La variable **h**: es el costo inter-ciudad estimado para terminar el recorrido desde nuestro estado actual n. Éste es un costo estimado ya que no sabemos cuál es el camino que implica menor costo para terminar el itinerario desde nuestro estado actual.
Finalmente la variable **f**: es la suma de la otras dos, y resulta ser un estimador del costo que me insumiría realizar el itinerario completo desde el estado actual (ciudades recorridas hasta el momento).

Como podemos apreciar, nuestra función de evaluación **f** tiene una parte que no se encuentra definida con exactitud (la parte de **h**), es por eso que resulta ser un estimador. Matemáticamente, se ha demostrado que si mi función de evaluación es minorante (estima que me costará menos encontrar la solución, de lo que realmente me costará), voy a encontrar la solución de menor costo. Por otra parte, si la función de evaluación es mayorante (sobreestima el costo para encontrar la solución), puedo arribar a una solución que no sea la de menor costo, y no cumpliríamos el cometido del enunciado.
Sabiendo esto, podemos plantear que nos conviene diseñar una función de estimación minorante (como la realizada en el algoritmo del programa TSP12.C que se adjunta), pero debemos tener presente que cuanto mas alejada sea la estimación de la realidad, mas proceso de cálculo y análisis serían requeridos, por lo que resulta lo mas conveniente que la función de estimación **h** sea lo mas certera posible.

Como posible mejora al algoritmo plateado (no fue implementada por razones de tiempo), se me ocurrió la posibilidad de utilizar diferentes maneras de calcular una **h** durante el proceso de búsqueda, ya que inicialmente, cuando no hemos recorrido muchos estados, es lo mismo que decir que el nivel de profundidad no es muy grande, el valor de **h** es el que mas peso tiene en la función **f** con respecto al **g**, y a partir de cierto nivel de profundidad, el valor de **g** comienza a pesar mas que el de **h**.

En nuestro algoritmo, se optó por realizar de una única manera la estimación de **h**, que resultó ser una buena relación de compromiso como estimador, ya que es mejor que la propuesta presentada en clase, de buscar el menor costo inter-ciudad no empleado hasta el momento, y multiplicarlo por la cantidad de ciudades o estados que faltan recorrer para terminar el itinerario. Esto lo puedo afirmar porque requiere menos tiempo de cálculo (ya que consiste en realizar una suma de valores pre calculados una única vez) y se aproxima mas a el valor de **h*** que es la estimación perfecta.

Antes de comenzar, comentaré que resulta indistinto tener conocimiento de cuál es la ciudad por la cuál iniciar nuestro itinerario, ya que debemos recorrer todas las ciudades, motivo por el cuál nosotros comenzaremos siempre por la ciudad 1, y la otra cuestión interesante es que el algoritmo calculará el recorrido de menor costo, pero no garantizará que sea el único, en principio si los costos inter-ciudad son iguales y bidireccionales, como lo estamos planteando, entonces tendremos siempre un número par de soluciones (una en un sentido y otra en el otro).

Detalle del programa TSP12.C:

Si bien, el programa se encuentra debidamente comentado para comprender su funcionamiento con la simple lectura del mismo, intentaré describir brevemente las tres porciones primordiales del algoritmo que realizan la parte importante de resolución del problema. Estas partes son: el *main*, la función *insert_abiertos* y la función *extract_abiertos*, de las cuales mencionaré las funciones que realizan, sin dar detalles de su funcionamiento para no desviarnos de lo interesante.

El programa se ejecuta teniendo los archivos de entrada en el mismo path dónde se encuentra el ejecutable, haciendo: *TSP12[ENTER]* o *TSP12 IN_xx.TXT[ENTER]*

Secuencia de *main* :

1 - Importamos el archivo de entrada.
2 - Adecuamos nuestra matriz de costos inter-ciudad, para adecuarlas al algoritmo.
3 - Calculamos los costos *minimos_iniciales*.
4 - Creamos el primer nodo de la lista.
5 - Verificamos si el primer nodo de la lista es el último por su nivel de profundidad.
6 - Si no lo es, genero los sucesores, y si se puede los inserto en la lista.
7 - Una vez generados todos los sucesores, desecho al nodo padre y retorno al paso 5.

El programa *main* comienza importando un archivo de texto **(1)** con el formato IN_xx.TXT, siendo xx una numeración que identifica al archivo de entrada, usando *load_matrix*, y nuestro programa responderá la solución en otro archivo de texto con el formato OUTxxGDG.TXT, manteniendo la numeración correspondiente al archivo de entrada. Además opté por presentar los resultados también en pantalla, por si se produciera algún error al crear el archivo de salida.
Del archivo de entrada, se obtiene la cantidad de ciudades del itinerario, que llamaremos N, y los costos inter-ciudad que son los elementos de la matriz de costos, cuyo índice es el número de ciudad.
Con ésta información generamos una matriz como la siguiente, en el caso de ser 5 ciudades:

	1	**2**	**3**	**4**	**5**
1	0	5	15	17	7
2	5	0	6	19	20
3	15	6	0	7	5
4	17	19	7	0	21
5	7	20	5	21	0

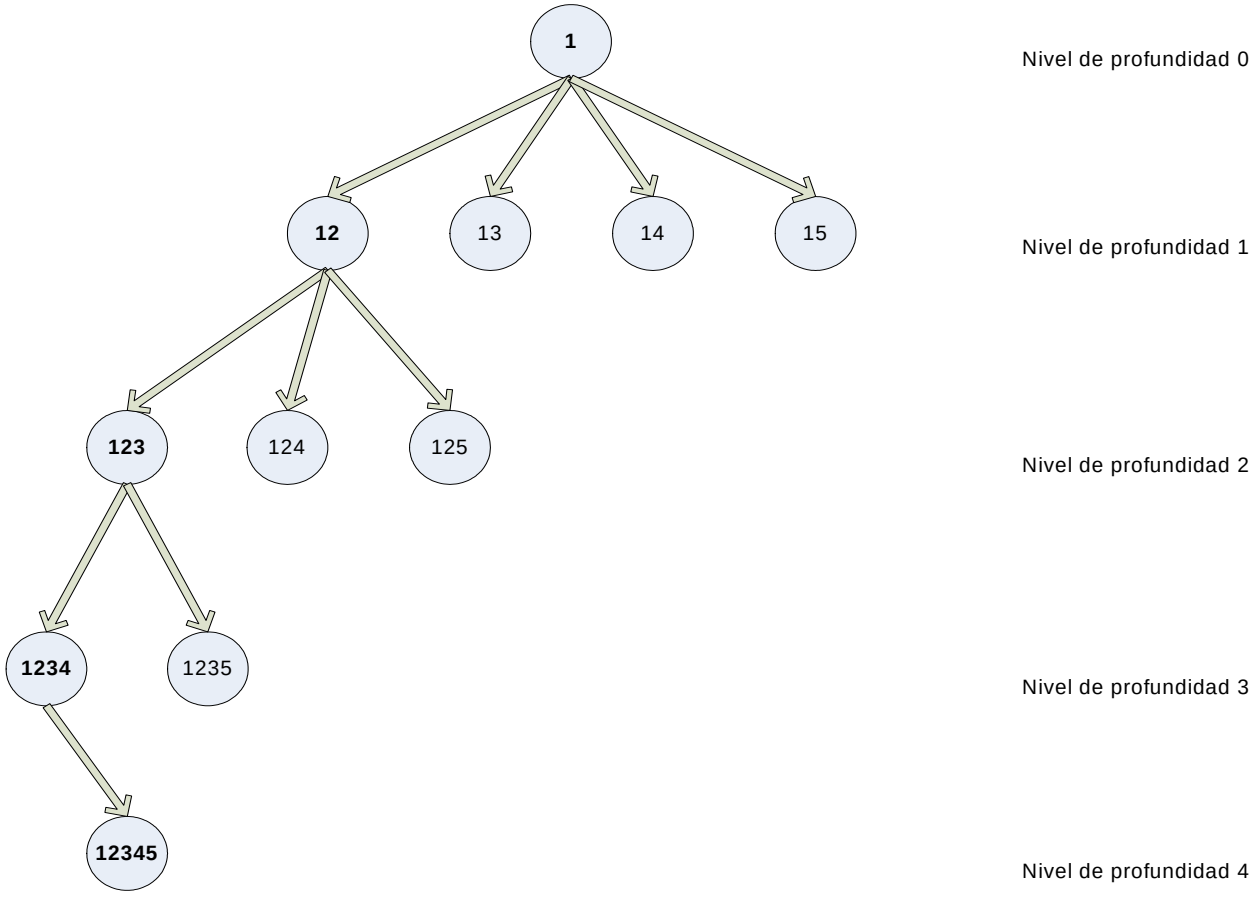
Luego por cuestiones del algoritmo, inventaremos un costo infinito en cada estado (el máximo valor que puede tomar la variable int) **(2)**, para que no se quiera quedar en una ciudad o estado para siempre, y a su vez calculamos el mínimo costo inter-ciudad de cada nivel de profundidad usando *minimos_iniciales* una única vez. Con éstos valores calcularemos en cada nodo la estimación $\mathbf{h_n}$ **(3)**.

	1	**2**	**3**	**4**	**5**
1	inf	5	15	17	7
2	5	inf	6	19	20
3	15	6	inf	7	5
4	17	19	7	inf	21
5	7	20	5	21	inf

minimos_iniciales
5
5
5
7
5

A continuación mostramos un diagrama de estados, que ilustra en forma esquemática como sería el comportamiento del programa. El contenido de cada globo es el estado actual (las ciudades por las que transcurrí), si bien no está graficado, cada nodo contiene el valor de **f**, el nivel de profundidad en el que nos encontramos y un puntero al nodo próximo.

Lo interesante del algoritmo (su inteligencia), es que en vez de ir analizando todas las posibilidades, que sería ir descomponiendo todos los niveles de cada nodo, el algoritmo calcula y evalúa todos los nodos posibles desde su estado actual, y se abrirá camino por el nodo que sea el de menor costo (el de menor **f**).



En éste esquema, se ha encontrado como camino mas corto, la secuencia 1 2 3 4 5 y retornado a 1 como camino de menor costo. Aquí se hace evidente lo antes mencionado que también es solución el camino 1 5 4 3 2 1.

Continuando con la función *main* podemos decir que luego de generar los *mínimos_iniciales*, creamos un primer nodo **(4)**, llamado *nod_nulo* que nos servirá como terminador de la lista de nodos que iremos abriendo a medida que evoluciona el algoritmo, en ésta lista quedarán ordenados los estados posibles en función de su valor de **f** a medida que se vallan insertando.
Como siguiente paso **(5)**, verificamos si el primer nodo de la lista es el último por su nivel de profundidad. Si no lo es, genero los sucesores si se puede **(6)**, con *nod_sucesor* y los inserto en la lista abierta.
Una vez generados todos los sucesores, desecho al nodo padre **(7)**, *nod_actual* y retorno al paso 5.

Ahora mencionaremos la función *insert_abiertos* : ésta función inserta en la lista de nodos *abiertos*, el nodo que está analizando de acuerdo a su valor de **f**, tener presente que si éste nodo es de menor costo que el primer elemento de la lista, el comienzo de la lista cambia para apuntar al nodo actual, ya que la lista de nodos abiertos está ordenada por orden decreciente de valor de **f**.
Secuencia de *insert_abiertos* :

1- Le pasamos a la función el puntero al nodo abierto, al que deseo insertar y el nulo.
2- Compara y lo inserta cuando corresponda.
3- Retorna el puntero a abiertos que es el comienzo de la lista de nodos.

Finalmente comentamos la tercer parte del algoritmo, que es la función *extract_abiertos*: a la cuál le pasamos el puntero al nodo *abiertos*, el nodo *nulo*, y *punteros_lista*.
Por *punteros_lista* la función retornará dos punteros en forma implícita, ya que una función en C, no puede retornar mas de un parámetro. Resumiendo, recibe el viejo puntero a *abiertos* y retorna el nuevo puntero a abiertos, y el nodo extraído que pasa a ser el nodo actual, a través de la estructura *punteros_lista*.
Secuencia de *extract_abiertos* :

1 - Extraigo el primer nodo.
2 - Modifico el puntero abiertos que es el comienzo de la lista a el sedo componente.

¿Que es un Grafo?

Es un conjunto de objetos llamados *vértices* o *nodos*, unidos por enlaces llamados *aristas* o *costos*.

Todos los grafos constan de al menos un conjunto de nodos (denotado por ***V***) y un conjunto de aristas (denotado por ***E***). Dependiendo del tipo de grafo, es como se define a las aristas.

Grafo no dirigido:

Un grafo no dirigido ***G*** es un par ordenado ***G=(V,E)*** dónde:

- ***V*** es un conjunto cualquiera (llamado de *vértices* o *nodos*)
- $E \subseteq \{x \in \mathcal{P}(V) : |x| = 2\}$ es un conjunto de *pares no ordenados* de vértices (llamado de *aristas* o *lados*)

Un par no ordenado es un conjunto de la forma $\{a, b\}$ dónde $a \neq b$. Representa la arista que tiene comienzo en ***a*** y finaliza en ***b***.

Grafo dirigido:

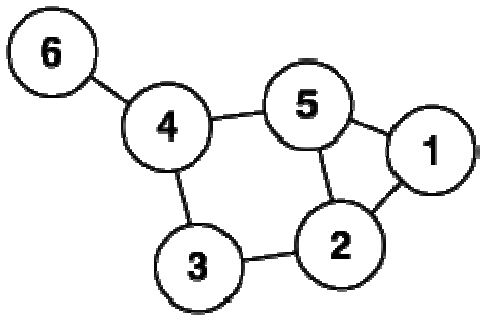
Un grafo dirigido ***G*** es un par ordenado ***G=(V,E)*** dónde:

- ***V*** es un conjunto cualquiera (llamado de *vértices* o *nodos*)
- $E \subseteq \{(a, b) \in V \times V : a \neq b\}$ es un conjunto de *pares ordenados* de vértices (llamado de *arcos* o *aristas*)

Propiedades de los grafos

Dos aristas de un grafo son llamadas **adyacentes** si tienen un vértice en común. De forma similar, dos vértices son llamados **adyacentes** si una arista los une. Se dice que una arista es **incidente** a un vértice si ésta lo une a otro.
En un grafo ponderado, a cada arista se le asocia un valor, como costo, peso, longitud, etc. según sea lo que modele.
Normalmente los vértices y las aristas de un grafo, por su naturaleza como elementos de un conjunto, son distinguibles. Este tipo de grafos son llamados etiquetados, también pueden sólo etiquetarse los vértices o áristas.

Ejemplo



La imagen es una representación del siguiente grafo:

- ***V***={1,2,3,4,5,6}
- ***E***={{1,2},{1,5},{2,3},{2,5},{3,4},{4,5},{4,6}}

El hecho que el vértice 1 sea adyacente con el vértice 2 puede ser denotado como 1 ~ 2.

ALGORITMO DE DIJKSTRA

El algoritmo de Dijkstra determina la ruta más corta desde un nodo origen ***s*** hacia los demás nodos.

Explicación del algoritmo

Dado un grafo G, el algoritmo Dijkstra encontrará desde un nodo origen fijado, el camino mínimo desde dicho nodo a cualquier otro dentro del conjunto V de vértices de dicho grafo, en forma eficiente.
La idea central subyacente en el algoritmo de Dijkstra es que cada subcamino del camino mínimo será a su vez un subcamino de coste mínimo.
El algoritmo de Dijkstra evita gran número de sumas y comparaciones gracias a que almacena información computacional de unas etapas o pasos a otras. Esto lo conseguimos mediante el procedimiento de etiquetado gracias al cual conseguiremos que el algoritmo quede reducido a una complejidad del orden de O (n^2).
Las distancias o costos, se almacenan en un vector D. Básicamente, el algoritmo toma en la i-ésima iteración al nodo que tiene la menor distancia ***Vi***, y analiza si es posible disminuir la distancia de sus nodos adyacentes.
Para hacerlo, verifica si la distancia hasta ***Vi + w(i,j)*** , es menor a la distancia actual en el nodo adyacente, siendo ***w(i,j)*** : el costo para ir de i al nodo adyacente.

A continuación se muestra el pseudocódigo del Algoritmo:

```
int[] Dijkstra(Grafo g, Nodo s)
  Construir un arreglo D[] con una celda por cada nodo en g;
  Hacer D[s] = 0 donde s es el nodo inicial;
  Hacer D[u] = infinito para todos los nodos u excepto el inicial;
  Construir una cola de prioridad mínima PQ que contenga cada nodo n, con D[n]
como su prioridad;
  while PQ no sea vacia hacer:
    Vi = P.removeMinElement();
```

```
    para cada vertice Vj todavía en PQ adyacente a Vi hacer:
      if D[Vi] + w(Vi, Vj) < D[Vj] entonces:
        D[Vj] = D[Vi] + w(Vi, Vj);
  Regresar D;
```

Ejemplo de implementación en C

A continuación se muestra un programa en el que se usa el algoritmo de Dijkstra para calcular la menor distancia desde cada uno de los nodos hacia los demás nodos.

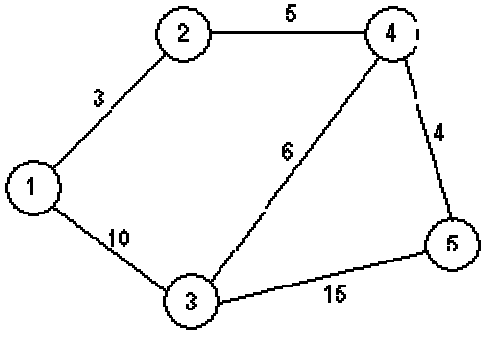
```
#include <cstdio>
#include <queue>
#include <cstring>

using namespace std;

#define INF 999
#define MAXN 100

// los nodos en el grafo van de 1 a n
int C[MAXN][MAXN], // matriz de adyacencia, 0 si no estan conectados
                   // costo para ir de i a j si estan conectados
    n,             // numero de nodos
    e;             // numero de aristas

void iniGrafo()
{
  for (int i=1; <=n; i++)
    memset(&C[i][1], 0, n * sizeof(int));
}

void insertanodo(int a, int b, int c)
{
  C[a][b] = C[b][a] = c;
}

int D[MAXN];          /* distancias minima desde s al nodo i */
int padre[MAXN];      /* ruta hacia el nodo i desde s */
int permanente[MAXN]; /* verdadero al tener la menor ruta al nodo i */

void dijkstra(int s)
{
  priority_queue< pair<int,int> > pq;
  pair <int,int> nodotmp;
  int Vi, Vj;

  // inicializacion
  for (int i=1; i<=n; i++)
    D[i] = INF,
    padre[i] = -1,
    permanente[i] = false;
  // inicializacion del nodo inicial
  D[s] = 0;
  pq.push( pair <int,int> (D[s], s) );
  // calculamos las distancias
  while( !pq.empty() )
  {
    nodotmp = pq.top();  pq.pop();
    Vi = nodotmp.second;
    if ( !permanente[Vi] )
    {
      permanente[Vi] = true;
      for (Vj = 1; Vj <= n; Vj++)
```

```
        if ( !permanente[Vj] && C[Vi][Vj] > 0 && D[Vi] + C[Vi][Vj] < D[Vj] )
          D[Vj] = D[Vi] + C[Vi][Vj],
          padre[Vj] = Vi,
          pq.push( pair <int,int> (-D[Vj], Vj) );
    }
  }
}

void imprimeGrafo(int n)
{
  for (int i=1; i<=n; i++)
    printf("%d(%d)  ", i, D[i]);
  printf("\n");
}

main()
{
  int a, b, c;

  // leemos el numero de nodos y aristas
  scanf("%d%d", &n, &e);
  // inicializamos el grafo
  iniGrafo();
  // leemos las aristas
  for (int i=0; i<e; i++)
    scanf("%d%d%d", &a, &b, &c),
    insertanodo(a, b, c);
  // usamos dijkstra para calcular la menor distancia
  // desde el i-esimo nodo hacia los demas
  for (int i=1; i<=n; i++)
    dijkstra( i ),
    printf("La menor distancia desde el nodo %d"
           " hacia los otros nodos es:\n", i),
    imprimeGrafo( n ),
    printf("\n");
}
```

Como entrada para el programa anterior se introdujo la siguiente información:

```
5 6
1 2 3
1 3 10
2 4 5
3 4 6
3 5 15
4 5 4
```

correspondiente al siguiente grafo:



La salida del programa fue, en la salida el primer número es el nodo y el valor entre paréntesis es la distancia:

```
La menor distancia desde el nodo 1 hacia los otros nodos es:
1(0)  2(3)  3(10)  4(8)  5(12)

La menor distancia desde el nodo 2 hacia los otros nodos es:
1(3) 2(0) 3(11) 4(5) 5(9)

La menor distancia desde el nodo 3 hacia los otros nodos es:
1(10) 2(11) 3(0) 4(6) 5(10)

La menor distancia desde el nodo 4 hacia los otros nodos es:
1(8) 2(5) 3(6) 4(0) 5(4)

La menor distancia desde el nodo 5 hacia los otros nodos es:
1(12) 2(9) 3(10) 4(4) 5(0)
```

Algoritmo de Floyd-Warshall (todos los caminos mínimos)

El problema que intenta resolver este algoritmo es el de encontrar el camino más corto entre todos los pares de nodos o vértices de un grafo. Esto es semejante a construir una tabla con todas las distancias mínimas entre pares de ciudades de un mapa, indicando además la ruta a seguir para ir de la primera ciudad a la segunda.

El algoritmo de Floyd-Warshall ('All-Pairs-Shortest-Path' - Todos los caminos mínimos) ideado por Floyd en 1962 basándose en un teorema de Warshall también de 1962, usa la metodología de Programación Dinámica para resolver el problema. Éste puede resolver el problema con pesos negativos y tiempos de ejecución iguales a $O(V^3)$; sin embargo, para ciclos de peso negativo el algoritmo tiene problemas.

A continuación se muestra el pseudocódigo del Algoritmo:

```
Para k = '0' hasta n hacer
{
    Para i = '0' hasta n hacer
    {
        Para j = '0' hasta n hacer
        {
        A[i,j] = mínimo(A[i,j],A[i,k] + A[k,j])
        }
    }
}

// Ak[i,j] significa: el costo del camino más corto que va de i a j y que no
// pasa  por algún vértice mayor que k.
```

Bibliografía:

Dijkstra

http://es.wikipedia.org/wiki/Algoritmo_de_Dijkstra

http://mictlan.utm.mx/temario/graf_dijkstra.htm

FloydWarshall

http://es.wikipedia.org/wiki/Algoritmo_de_Floyd-Warshall

http://personales.upv.es/arodrigu/grafos/FloydWarshall.htm

http://www.algorithmist.com/index.php/Floyd-Warshall's_Algorithm.c

http://ants.dif.um.es/asignaturas/redes/redes/tema3/floyd.htm

http://es.wikipedia.org/wiki/Implementaci%C3%B3n_del_algoritmo_de_Floyd_en_Java

Eficiencia computacional de algoritmos usando MATLAB

El concepto de la eficiencia computacional radica en saber cómo se ha empleado el tiempo de la CPU en la ejecución de un determinado programa. Para evaluarla, es necesario poder medirla, una vez medida resultaría una herramienta o un indicador muy útil para determinar los cuellos de botella de un programa, es decir las funciones y las líneas de código que más veces se llaman y que se llevan la mayor parte del tiempo de ejecución.
Es evidente que, cuando se trata de mejorar la eficiencia de un programa, es más importante tratar de mejorar una función que se lleva el 60% del tiempo total, que otra que sólo se lleva el 2%.

Para conocer el tiempo de ejecución de un algoritmo contamos con varias posibilidades. La primera de ellas consiste usar la función **`clock`**, que devuelve la hora actual con una precisión de centésimas de segundo.

```
clock
ans =
1.0e+003 *
1.9970 0.0050 0.0050 0.0160 0.0070 0.0534
```

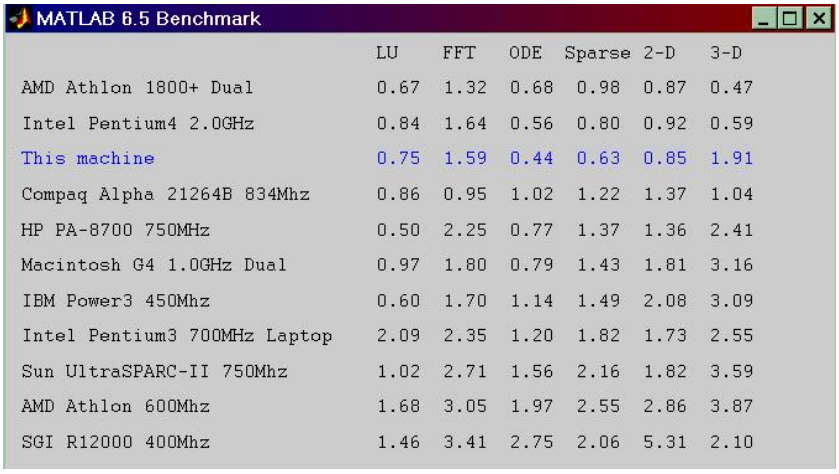
`etime(t2, t1)` devuelve el tiempo en segundos transcurrido entre dos tiempos **`t2`** y **`t1`**.
Por ejemplo, se puede calcular el tiempo consumido en calcular la inversa de una matriz **`M 100×100`** del siguiente modo:

```
M = rand(1000);
t = clock;
inv(M);
time = etime(clock, t)
time =
2.0730
```

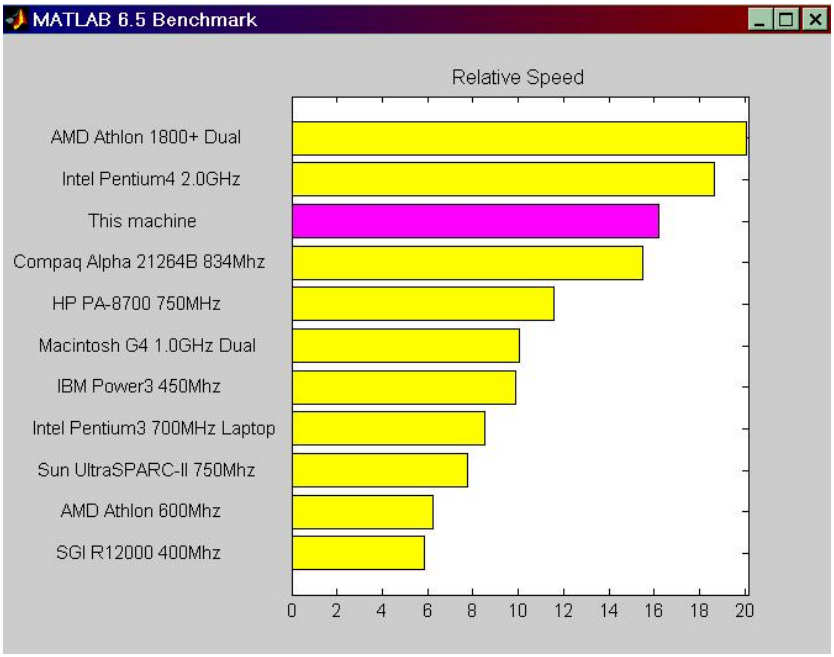
Una última posibilidad es usar las instrucciones **`tic`** y **`toc`**.
El modo de proceder es ejecutar la orden **`tic`** antes del algoritmo y la orden **`toc`** después.

```
M = rand(1000);
tic, inv(M);
toc
elapsed time =
2.0630
```

Para conocer el nivel de desempeño actual de nuestra PC se puede ejecutar el comando **`bench`** de Matlab, las siguientes gráficas corresponden al análisis de mi computadora:

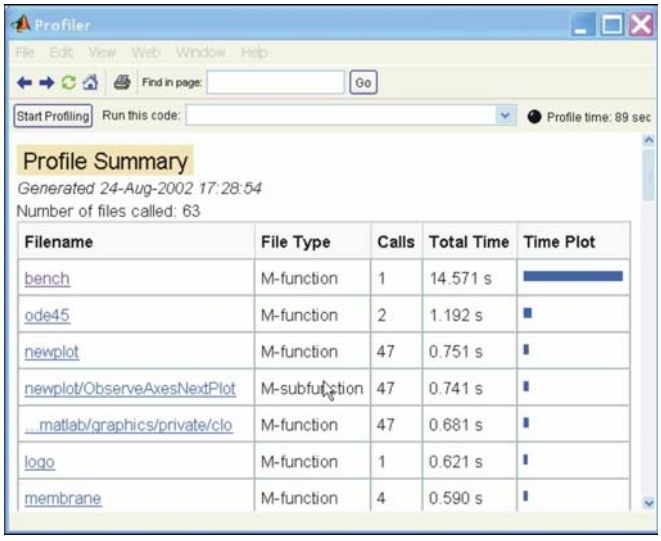

MATLAB 6.5 Benchmark

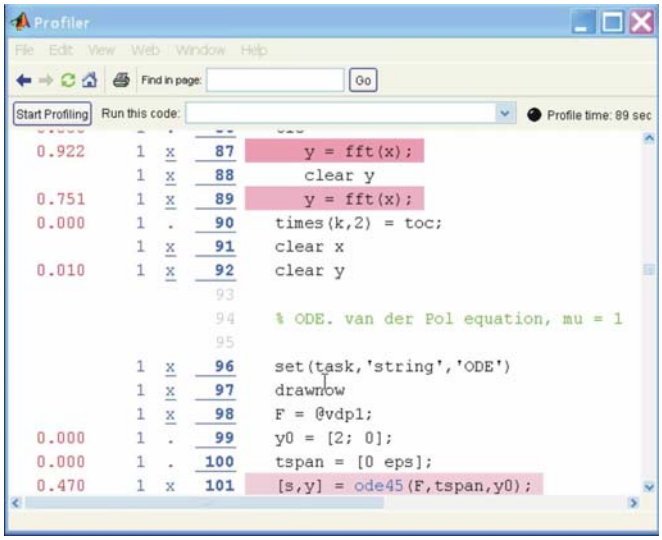
	LU	FFT	ODE	Sparse	2-D	3-D
AMD Athlon 1800+ Dual	0.67	1.32	0.68	0.98	0.87	0.47
Intel Pentium4 2.0GHz	0.84	1.64	0.56	0.80	0.92	0.59
This machine	0.75	1.59	0.44	0.63	0.85	1.91
Compaq Alpha 21264B 834Mhz	0.86	0.95	1.02	1.22	1.37	1.04
HP PA-8700 750MHz	0.50	2.25	0.77	1.37	1.36	2.41
Macintosh G4 1.0GHz Dual	0.97	1.80	0.79	1.43	1.81	3.16
IBM Power3 450Mhz	0.60	1.70	1.14	1.49	2.08	3.09
Intel Pentium3 700MHz Laptop	2.09	2.35	1.20	1.82	1.73	2.55
Sun UltraSPARC-II 750Mhz	1.02	2.71	1.56	2.16	1.82	3.59
AMD Athlon 600Mhz	1.68	3.05	1.97	2.55	2.86	3.87
SGI R12000 400Mhz	1.46	3.41	2.75	2.06	5.31	2.10



El profiler de Matlab

En las últimas versiones 6.X se ha incorporado una herramienta de análisis de eficiencia de algoritmos que cada vez tiene mas funcionalidades, el **`profiler`**. Para activar el profiling, que es el análisis del "costo de procesamiento" de cada línea de código de un programa, basta escribir **`profile on`** y para terminar **`profile off`**.
Matlab activará el **`profiler`** e ira almacenando la información a la que se puede acceder con **`profile report`**. Los informes detallados son bastante precisos, y son una gran ayuda en la optimización de los programas.





En los informes mostrados arriba, se puede observar que el programa analizado emplea varios archivos tipo m, como ser: bench, ode45, newplot, y otros.

La columna Calls, indica la cantidad de invocaciones que recibió cada archivo, y un gráfico de barras con el tiempo total empleado por el archivo invocado.
Ésta información nos expresa de manera simple cuáles son los archivos mas invocados, y los que mas tiempo demoran. Justamente los que debemos inicialmente revisar, para optimizar.

En la segunda pantalla, aparece el programa, con la cantidad de tiempo empleada en cada línea de código, lo que nos ayuda a ver que instrucciones son las que nos conviene emplear, y de que manera.
Otras formas de arrancar el profiler consisten en ***View → Profiler***, o tipear **`profile viewer`** en la consola de comandos, y pulsar el botón ***Start Profiling***. Inmediatamente aparece una marca verde y comienza a correr el tiempo.

Al terminar la ejecución se muestra el resumen de resultados visualizados previamente, del que podemos profundizar diciendo que las sentencias que han llevado más tiempo de la CPU, aparecen coloreadas en un tono rosa de intensidad creciente.
Otra información de la vista del código, es la columna ***acc:*** un punto (.) indica que esa línea fue acelerada automáticamente por ésta herramienta, mientras que (x) indica lo contrario.
En nuestro ejemplo, la instrucción **`times`** pudo ser optimizada automáticamente, pero la tranformada rápida de Fourier (Fast Fourier Transform) **`fft`**, ya no puede optimizarse mas.

Típicamente para analizar con el ***profiler***, se realiza usando la función **`profile`** intercalada en el código fuente (estas líneas forman parte de un fichero ****.m***):

```
profile on -detail operator;
sol=ode15s(@tiropar4,tspan2,y0,options,1,0.001);
profile report;
```

La primer línea activa el ***profiler*** a la vez que define el ***grado de detalle*** que se desea. La segunda línea es una llamada a la función ***ode15s***, que a su vez llama a muchas otras funciones y la tercera línea detiene el ***profiler*** y le pide un informe en HTML con los resultados calculados.

Actualmente existen tres posibles grados de detalle respecto a la información que se le pide al ***profiler***:

Mmex	determina el tiempo utilizado por funciones y sub-funciones definidas en ficheros ****.m*** y ****.mex***. Ésta es la opción por defecto.
builtin	como el anterior incluyendo las ***funciones intrínsecas*** de MATLAB.
operator	como ***builtin*** pero incluyendo también el tiempo empleado por los ***operadores*** tales como la suma + y el producto *.

Otros posibles comandos relacionados con el ***profiler*** de MATLAB son los siguientes:
profile viewer abre la ventana del ***profiler***
profile on activa el ***profiler*** poniendo a cero los contadores
profile on –detail level como el anterior, pero con el grado de detalle indicado
profile on –history activa el ***profiler*** con información sobre el orden de las llamadas
profile off desactiva el ***profiler*** sin poner a cero los contadores
profile resume vuelve a activar el ***profiler*** sin poner a cero los contadores
profile clear pone a cero los contadores
profile report detiene el ***profiler***, genera páginas HTML con los resultados y los muestra en un browser
profile report basename genera un informe consistente en varios ficheros HTML en el directorio actual; los nombre de los ficheros están basados en el

nombre ***basename***, que debe darse sin extensión
profile plot detiene el ***profiler*** y representa gráficamente los resultados en un diagrama de barras correspondientes a las funciones más usadas