El problema del viajante o vendedor viajero responde a la siguiente pregunta: dadas *n+1* ciudades (enumeradas de 0 a *n*) y las distancias entre cada par de ellas, ¿cuál es la ruta más corta posible que inicia en la ciudad 0, visita cada ciudad una vez y al finalizar regresa a la ciudad 0?

La solución óptima se puede obtener en tiempo $O(n!)$. Un algoritmo más eficiente puede hacerlo casi en tiempo $O(2^n)$. En la práctica es demasiado lento buscar la solución óptima si *n* es grande. La función *viajante* de más abajo es una heurística simple para encontrar una solución aproximada. Recibe como parámetros el número *n* de ciudades (además de la ciudad 0), la matriz *m* de distancias entre ciudades (*m[i][j]* es la distancia entre las ciudades *i* y *j*), un arreglo de salida *z* de tamaño *n+1*, en donde se almacenará la ruta más corta, y un número *nperm*. Esta función genera *nperm* permutaciones aleatorias de las ciudades 1 a *n*. Cada permutación corresponde a una ruta aleatoria partiendo de la ciudad 0, pasando por todas las otras ciudades y llegando a la ciudad 0 nuevamente. La función calcula la distancia recorrida para cada ruta, selecciona la ruta más corta (la que recorre la menor distancia), entregando en *z* cuál es esa ruta y retornando la distancia recorrida por *z*. No es la ruta óptima, pero mientras más grande es *nperm*, más se acercará al óptimo.

```
double viajante(int z[], int n, double **m, int nperm) {
  double min= DBL_MAX; // la menor distancia hasta el momento
  for (int i= 1; i<=nperm; i++) {
    int x[n+1];           // almacenará una ruta aleatoria
    gen_ruta_alea(x, n); // genera ruta x[0]=0, x[1], x[2], ..., x[n], x[0]=0
    // calcula la distancia al recorrer 0, x[1], ..., x[n], 0
    double d= dist(x, n, m);
    if (d<min) {     // si distancia es menor que la que se tenía hasta el momento
      min= d;        // d es la nueva menor distancia
      for (int j= 0; j<=n; j++)
        z[j]= x[j]; // guarda ruta que recorre la menor distancia en parámetro z
  } }
  return min;
}
```

Las funciones *viajante*, *gen_ruta_alea* y *dist* son dadas. Por ejemplo si *n* es 4, después de la llamada a *gen_ruta_alea(x, n)* el arreglo *x* podría ser 0, 4, 1, 3, 2. También podría ser 0, 3, 1, 4, 2, etc. Hay *n!* permutaciones posibles.

Programe la función *viajante_par* con la misma heurística pero generando las *nperm* rutas aleatorias en paralelo en *p* procesos pesados (creados con *fork*). Recibe los mismos parámetros que *viajante*, más el parámetro *p*. Note que **debe entregar la mejor ruta encontrada en el arreglo z**.

Metodología obligatoria: Lance *p* nuevos procesos pesados (hijos) invocando *p* veces *fork*. Cada hijo evalúa *nperm/p* rutas aleatorias invocando *viajante(..., nperm/p)*. Cuidado: recuerde que los procesos pesados no comparten la memoria. Cada hijo debe enviar su mejor ruta al padre por medio de un *pipe*. Use el proceso padre solo para crear a los hijos y para elegir la mejor solución entre las recibidas a través de los *p pipes* que conectan al padre con sus hijos. Si la mejor solución fue por ejemplo la del hijo 3, *viajante_par* debe retornar el valor *min* que calculó el hijo 3 y llevar una copia del arreglo *z* calculado por el hijo 3 al parámetro *z* de *viajante_par*. La forma de crear los procesos hijos y pipes es muy similar a la manera en que se hizo para encontrar un factor de un entero en la cátedra del jueves 16 de noviembre. La forma de enviar el arreglo *z* por el pipe es similar a la manera en que se envía el arreglo ordenado en el quicksort paralelo de la cátedra del martes 14 de noviembre.

Ud. encontrará el siguiente problema: todos los procesos hijo generarán exactamente los mismos subconjuntos llegando todos a la misma solución porque la función *gen_ruta_alea* entregará la misma ruta aleatoria en los 8 procesos hijos. Para lograr que cada proceso genere rutas distintas cambie la semilla para la función *random* en cada proceso hijo, antes de generar los subconjuntos aleatorios con:

*srandom(getUSecsOfDay()*getpid());*

Para evitar el warning de encabezado indefinido para *srandom* agregue esta primera línea en *viajante.c* (como está hecho en *test-viajante.c*):

#define _XOPEN_SOURCE 500

Se requiere que el incremento de velocidad (*speed up*) sea al menos un factor 1.5x. Cuando pruebe su tarea en su notebook asegúrese que posea al menos 2 cores y que esté configurado en modo máximo rendimiento. Si está configurado para ahorro de batería podría no lograr el *speed up* solicitado.

Instrucciones

Descargue *t7.zip* de U-cursos y descomprímalo. Ejecute el comando *make* sin parámetros en el directorio *T7* para recibir instrucciones acerca del archivo en donde debe programar su solución (*viajante.c*), cómo compilar y probar su solución y los requisitos que debe cumplir para aprobar la tarea.

Entrega

Ud. solo debe entregar por medio de U-cursos el archivo *viajante.zip* generado por el comando *make zip*. Contiene los archivos *viajante.c* y *resultados.txt*. A continuación es muy importante que descargue de U-cursos el mismo archivo que subió, luego descargue nuevamente los archivos adjuntos y vuelva a probar la tarea tal cual como la entregó. Esto es para evitar que Ud. reciba un 1.0 en su tarea porque entregó los archivos equivocados. Créame, sucede a menudo por ahorrarse esta verificación. Se descontará medio punto por día de atraso. No se consideran los días de receso, sábado, domingo o festivos.