

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.

Programa la función *viajante_par* con la misma heurística pero generando las $nperm$ rutas aleatorias usando p threads que se ejecutan en paralelo. Recibe los mismos parámetros que *viajante*, más el parámetro p .

Metodología obligatoria: Lance p nuevos threads invocando p veces *pthread_create*. Cada thread evalúa $nperm/p$ rutas aleatorias invocando *viajante*(..., $nperm/p$). Defina una estructura con campos para todos los

parámetros que recibirá la función *viajante* (versión secuencial) y otro campo para el valor retornado. Cuidado: no puede compartir el arreglo z entre todos los threads. Debe crear un arreglo z independiente para cada thread. Use el thread principal solo para crear los threads y para elegir la mejor solución entre las mejores encontradas por los p threads. Si la mejor solución fue por ejemplo la del thread 3, *viajante_par* debe retornar el valor *min* que calculó el thread 3 y copiar el arreglo z calculado por el thread 3 en el parámetro z de *viajante_par*. La forma de crear los threads es muy similar a la manera en que se crearon los threads para resolver en paralelo el problema de la búsqueda de un factor en la clase auxiliar del viernes 14 de mayo.

Se requiere que el incremento de velocidad (*speed up*) sea al menos un factor 1.7x. Cuando pruebe su tarea en su notebook asegúrese de que posea al menos 2 cores, que esté configurado en modo alto rendimiento y que no estén corriendo otros procesos intensivos en uso de CPU al mismo tiempo. De otro modo podría no lograr el *speed up* solicitado.

Instrucciones

Baje *t5.zip* de U-cursos y descomprímalo. El directorio *T5* contiene los archivos *test-t5.c*, *Makefile* y *t5.h* (con los encabezados requeridos). Ud. debe crear el archivo *t5.c* y programar ahí la función *viajante_par*. Hay una plantilla para su solución en el archivo *t5.c.plantilla*.

Pruebe su tarea bajo Debian 10 de 64 bits con los comandos:

- *make test-g*
- *make test-O*
- *make test-valgrind*

Copie la salida de todos estos comandos y péguela en el archivo *resultados.txt*. Ud. necesita cumplir el requisito de un speed up de 1.7x solo al ejecutar con *make test-O*. No se preocupe si en las otras ejecuciones no cumple ese requisito. La ejecución con *make test-valgrind* no debe reportar errores de manejo de memoria o fugas de memoria. Se descontará medio punto si alguna de las compilaciones reporta algún warning.

Entrega

Ud. solo debe entregar los archivos *t5.c* y *resultados.txt* en el formato *.zip* por medio de U-cursos. Se descontará medio punto por día de atraso. No se consideran los días de vacaciones, sábado, domingo o festivos.