Metodología de la Programación Paralela
Curso 2023/24

Implementación de un Algoritmo Genético

En esta práctica se proporciona el código secuencial de un programa que implementa un algoritmo genético. Este programa se utilizará como punto de partida para el desarrollo de las diferentes versiones paralelas (OpenMP, MPI, Híbrida) que se propondrán en las siguientes sesiones de prácticas.

El principal objetivo es ir desarrollando y depurando el software paso a paso con el fin de obtener un código eficiente, bien estructurado y libre de errores. Para ello, se trabajará con las herramientas descritas en la sesión inicial.

La fecha de entrega de la práctica será el **6 de octubre a las 23:55**. Entregas posteriores no se calificarán. La entrega se realizará a través de la tarea habilitada en el Aula Virtual, adjuntando un fichero .zip o .tgz que incluya los ficheros de código fuente que han sido modificados y un documento en formato pdf que explique cómo se han resuelto los diferentes ejercicios propuestos.

Introducción

Los algoritmos genéticos son un tipo de metaheurísticas que se inspiran en la teoría de la evolución para encontrar buenas soluciones a problemas de optimización. Estos algoritmos se basan en hacer evolucionar una población de individuos sometiéndola a acciones aleatorias similares a las que se dan en la evolución biológica, como el cruce y la mutación genética. Al mismo tiempo, en cada paso de la evolución se lleva a cabo un proceso de selección en el que se escogen los individuos más aptos, es decir, aquellos que queremos que sobrevivan. Este tipo de metaheurísticas utilizan un esquema algorítmico como el que se indica a continuación:

```
Generar poblacion inicial
Calcular bondad de soluciones
repetir
      Seleccionar
      Cruzar
      Mutar
      Calcular bondad de soluciones
hasta que poblacion converja
```

donde:

1. **Población Inicial**: representa el conjunto de individuos, donde cada uno es una posible solución al problema que se quiere resolver.
2. **Bondad**: indica cuánto se parece un individuo al objetivo que se pretende conseguir. Un individuo será seleccionado para reproducirse si el resultado de su función de *fitness* es bueno.
3. **Seleccionar**: escoge los individuos más aptos para "transmitir sus genes" a la siguiente generación. Selecciona pares de individuos (ascendientes) según su bondad. Los individuos con mayor bondad tienen más probabilidad de ser seleccionados para reproducirse.
4. **Cruzar**: para cada par de individuos selecciona un punto (o puntos) aleatorio a partir del que se produce la mezcla y los nuevos descendientes se crean intercambiando los genes de los padres entre sí.

5. **Mutar**: produce mutaciones en los descendientes en función de una cierta probabilidad (normalmente baja). Esto permite, por un lado, que la población sea diversa y, por otro, evita que el algoritmo converja de forma prematura.

El algoritmo genético termina si la población converge, es decir, ya no se generan descendientes que sean significativamente diferentes de los obtenidos en la generación anterior. En cambio, si el algoritmo converge lentamente se suele limitar el número de generaciones.

Para más información acerca de este tipo de algoritmos metaheurísticos, se recomienda consultar https://cs.gmu.edu/~sean/book/metaheuristics/Essentials.pdf

Problema de Optimización

El Problema del Viajante de Comercio (TSP – Traveling Salesman Problem) en su formulación asimétrica, consiste en determinar un ciclo hamiltoniano de mínimo coste en un grafo dirigido $D = (N, A)$ donde existe un coste $c_{i,j}$ para cada arco $(i,j) \in A$. Recordemos que un ciclo hamiltoniano es un ciclo que visita todos los nodos del grafo exactamente una vez. En nuestro caso asumiremos que el grafo es completo, es decir, que contiene un arco para cada par ordenado de nodos. Uno de los posibles modelos para este problema tiene una variable para cada arco:

$$\forall (i,j) \in A, \quad x_{i,j} = \begin{cases} 1 & si\ (i,j)\ está\ en\ el\ ciclo; \\ 0 & en\ otro\ caso. \end{cases}$$

$$\min \sum_{(i,j)\in A} c_{i,j} x_{i,j}$$

$$s.a. \quad \sum_{i \in N} x_{i,v} = 1 \qquad \forall \quad v \in N$$

$$\sum_{j \in N} x_{v,j} = 1 \qquad \forall \quad v \in N$$

$$x_{i,j} \in \{0,1\} \qquad \forall \quad (i,j) \in A$$

La función objetivo es la suma de los costes de los arcos seleccionados. En cada nodo debe haber exactamente uno de los arcos seleccionados entrando al nodo (primera restricción) y exactamente uno de los arcos seleccionados saliendo del nodo (segunda restricción). Con estas condiciones visitamos cada nodo exactamente una vez. Debido a que vamos a usar un algoritmo metaheurístico para resolver el problema no es necesario introducir ninguna restricción adicional para prevenir la posible formación de subciclos.

Para el problema que nos ocupa, los vértices representan las ciudades por las que pasa el viajante conectados entre sí por arcos o carreteras. Se trata de encontrar el *tour* que cumple con las condiciones anteriores, y que codificaremos mediante un array de enteros representando el sucesor y el predecesor de cada ciudad. Usando la terminología de algoritmos evolutivos, la codificación usada representa un individuo cuyo conjunto conforma una población. Cada individuo lleva asociado un valor de fitness que coincide con la función objetivo seleccionada para este problema.

Por ejemplo, el tour (0, 4, 2, 1, 5, 3, 0) quedaría codificado como [0 4 2 1 5 3] donde, por simplicidad, no incluiremos el nodo final por coincidir con el origen.

Programa

El código secuencial proporcionado lleva a cabo las siguientes tareas:

1. Genera una matriz de números reales aleatorios, D, con costes $c_{i,j}$ para cada arco $(i,j) \in A$ que se corresponden con las distancias entre ciudades. Toda la funcionalidad de lectura y escritura de datos del problema se encuentra en el fichero `/src/io.c`.
2. Genera de forma aleatoria los posibles *tours* que representan a la población inicial.
3. Aplica el algoritmo genético un número de iteraciones hasta alcanzar soluciones relativamente buenas minimizando los valores de fitness.

El programa recibe como entrada los siguientes parámetros: el número de ciudades (*n*), el número de generaciones (*n_gen*) del algoritmo genético y el tamaño de la población de individuos (*tam_pob*).

Si se habilita la macro `PRINT` del fichero `/src/ga.c`, el programa imprimirá el mejor valor de *fitness* obtenido en cada momento, permitiendo de esta forma analizar cómo evoluciona el algoritmo.

Para compilar y ejecutar el programa en los ordenadores del laboratorio se proporciona el script ***run.sh*** y un *Makefile* en el directorio `src`. Los comandos `make sec` y `make test_sec` permiten realizar sendas operaciones, respectivamente. Los valores de los parámetros de entrada a utilizar para ejecutar el programa se encuentran en el fichero `/input/in.txt`. Puedes modificar/añadir los parámetros que consideres oportunos.

Ejercicios

Se realizarán los siguientes ejercicios:

1. Detectar los errores de memoria presentes en el código mediante el uso de la herramienta Valgrind (solo la usaremos en esta Práctica 0). Mostrar el informe devuelto por Valgrind e indicar cómo se han solucionado los errores.

2. Implementar las funciones `cruzar()`, `mutar()` y `fitness()` contenidas en el fichero `ga.c`. Los cambios realizados en el código deben ir acompañados de comentarios. En la memoria se explicará cómo se ha implementado cada una de las funciones, justificando las decisiones tomadas. El código mostrado en la memoria debe coincidir con el que se entregue.

3. Introduce un nuevo parámetro que represente la tasa de mutación, *m_rate*, que será leído del fichero de entrada `in.txt`, y que sustituirá la misma variable definida en la función `mutar` del algoritmo, facilitándose así la experimentación con este parámetro. Habrá que modificar también los ficheros *run.sh* y *Makefile* así como los argumentos de algunas funciones del programa. Entonces, en el fichero de entrada habrá cuatro columnas: *n, n_gen, tam_pob, m_rate* codificando, por ejemplo, los valores 100 150 100 0.15.

4. Cambiar el criterio de convergencia del algoritmo evolutivo de forma que se realice en función del valor de *fitness*. El algoritmo finalizará cuando el valor de *fitness* no mejore el obtenido en la iteración anterior en un valor porcentual establecido. Prueba con valores como 2%, 5%, 10% u otros que estimes oportunos. Los resultados obtenidos con cada uno deben reflejar tanto el tiempo empleado en obtener la solución como valor final de fitness alcanzado.

   El cambio en el criterio de convergencia solo es para este ejercicio 4, posteriormente se seguirá con el criterio habitual de parada por número de iteraciones.