Metodología de la Programación Paralela
Curso 2024/25

Implementación de un Algoritmo Evolutivo

En esta práctica se proporciona el código secuencial de un programa que implementa un algoritmo evolutivo. 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 **4 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.

Es recomendable, aunque no obligatorio, el uso de la herramienta *GitLab/GitHub* como repositorio software que permite ir almacenando los cambios realizados durante el desarrollo de la práctica.

Describir el sistema computacional utilizado con la ayuda de la herramienta: `lstopo -f -p --no-legend --no-io sistema.svg`.

Introducción

Los algoritmos evolutivos son un tipo de metaheurísticas que se inspiran en la naturaleza para encontrar buenas soluciones a problemas de optimización. Estos algoritmos hacen 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. Estas 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 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 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 de máxima diversidad (MDP – Maximum Diversity Problem) consiste en seleccionar un subconjunto *B* de *m* elementos a partir de un conjunto *A* de *n* elementos de tal forma que se maximice la suma de las distancias entre los elementos escogidos. La definición de distancia entre elementos depende del ámbito específico en el que se aplique el problema.

En nuestro caso, los elementos serán números enteros positivos y la distancia entre ellos valores reales que se generarán directamente para cada par de elementos de A. Cada subconjunto B de *m* candidatos se representará mediante un array de *m* elementos. Usando la terminología de algoritmos evolutivos, la codificación usada representa un individuo, y el conjunto de estos conforma una población. Cada individuo lleva asociado un valor de fitness *F* que coincide con la función objetivo *F.O.* seleccionada para este problema.

De esta forma, el MDP puede definirse como un problema binario cuadrático, donde la variable x_i ($i = 0, \dots, n-1$) toma el valor 1 si el elemento *i* es seleccionado y 0 en caso contrario. Por tanto, se podría formular de la siguiente manera:

Maximizar la F.O.: $\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} d_{ij} \cdot x_i \cdot x_j$

condicionado a: $\sum_{i=0}^{n-1} x_i = m\,, \quad x_i = \{0,1\}, \;\; 0 \leq i < n$

A continuación, se muestra una instancia reducida del problema, las estructuras de datos utilizadas y la solución obtenida. Partimos del conjunto A = {0, 1, 2, 3, 4} con *n=5* y la siguiente matriz de distancias:

$$D = \begin{pmatrix} 0 & 3.5 & 14.1 & 7.3 & 16.8 \\ 0 & 0 & 2.2 & 1.4 & 5.0 \\ 0 & 0 & 0 & 3.7 & 18.1 \\ 0 & 0 & 0 & 0 & 1.5 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

siendo d_{ij} un elemento de la matriz, y donde se ha tenido en cuenta que existe una simetría en las distancias entre elementos, con $d_{ij} = d_{ji}$, que no codificamos por simplicidad.

En este caso, por ejemplo, d_{02} = 14.1 representa la distancia entre el elemento 0 y el 2, y d_{13} = 1.4 la distancia entre el 1 y el 3.

Debido a que esta forma de almacenar los valores de distancia conlleva un uso ineficiente de memoria (pues sólo contiene información útil por encima de la diagonal principal), **en la práctica**, se ha optado por **representar la matriz de la siguiente forma compacta**:

$$D' = (3.5 \quad 14.1 \quad 7.3 \quad 16.8 \quad 2.2 \quad 1.4 \quad 5.0 \quad 3.7 \quad 18.1 \quad 1.5)$$

donde tenemos una codificación en forma de array unidimensional que contiene solo los valores distintos de cero ordenados como filas consecutivas, y con elementos d'_k con índices $k = f(i,j,n)$ dados por la expresión:

$$k = \frac{(n^2 - n)}{2} - \frac{(n-i)^2 - (n-i)}{2} + j - i - 1$$

Así, por ejemplo, el elemento d_{13} = 1.4 se corresponde con d'_5.

Entonces, para $m = 3$, los posibles subconjuntos B_i serían:

{0, 1, 2}, {1, 2, 4}, {2, 3, 4} … hasta $C_3^5 = 10$ combinaciones.

Y el subconjunto que maximiza las distancias sería B_{opt} = {0, 2, 4} con F_{opt} = 14.1+16.8+18.1 = 49.

Para más información acerca de este tipo de problemas, se recomienda consultar la página web http://grafo.etsii.urjc.es/optsicom/mdp

Programa

El código secuencial proporcionado lleva a cabo las siguientes tareas:

1. Genera un fichero con los valores en formato compacto de la matriz D de distancias con números reales aleatorios entre cada par de elementos del conjunto A de enteros.
2. Genera de forma aleatoria los posibles subconjuntos B_i que representan a la población inicial.
3. Aplica el algoritmo evolutivo un número de iteraciones hasta alcanzar soluciones relativamente buenas maximizando los valores de fitness.

El programa recibe como entrada los siguientes parámetros: el tamaño del conjunto *A* (*n*), el tamaño de cada subconjunto *B* (*m*), el número de generaciones (*n_gen*) del algoritmo evolutivo y el tamaño de la población de individuos (*tam_pob*).

Si se habilita la macro `PRINT` del fichero `/src/mh.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, se proporciona el script *run.sh* y un *Makefile* en el directorio `/src/`. Los comandos `make sec` y `make test_sec` permiten realizar ambas operaciones. Los valores de los parámetros de entrada a utilizar para ejecutar el programa se encuentran en el fichero `/input/igen/in.txt`. Puedes modificar/añadir los parámetros que consideres oportunos.

Además, desde `/input/igen/` podemos compilar y ejecutar el fichero `igen.cpp` para generar (en `/input/`) otros ficheros de entrada (instancias del problema) con datos aleatorios de distancias correspondientes a cada tamaño de matriz establecido, y cuyo formato es [$i \quad j \quad d_{i,j}$], es decir, tres columnas, con el elemento origen *i*, el destino *j*, y la distancia $d_{i,j}$ entre ambos en formato doble precisión.

Ejercicios

Se realizarán los siguientes ejercicios:

1. Detectar los errores de memoria presentes en el código mediante el uso de la herramienta *Valgrind*. 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 `mh.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 a la constante definida para tal propósito en el fichero `mh.c`, facilitándose así la experimentación con este parámetro que pasa a ser variable. 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 cinco columnas: *n, m, n_gen, tam_pob, m_rate* codificando, por ejemplo, los valores 500 200 100 50 0,15.

4. Cambiar el criterio de convergencia del algoritmo evolutivo de forma que se realice en función del *fitness*. El algoritmo finalizará cuando el valor de *fitness* no mejore el obtenido en la iteración anterior (o las 5, o 10, etc., iteraciones anteriores si el mejor fitness no cambia demasiado entre iteraciones) 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.

Observaciones

- La herramienta *Valgrind* solo se utilizará para depurar el código en esta Práctica 0, pero no en las siguientes prácticas.

- Si alguna función o subrutina puede implementarse de varias formas: con bucles *while*, *for*, etc., tener en cuenta cuál de ellas es preferible para una posterior paralelización.

- Una vez realizados los experimentos, para las siguientes prácticas, se puede volver a fijar el parámetro de mutación MUTATION_RATE con un valor razonable, con lo que los parámetros de entrada al programa volverán a ser cuatro.

- El cambio en el criterio de convergencia solo es para el ejercicio 5, posteriormente se seguirá con el criterio habitual de parada por número fijo de iteraciones.

Otros aspectos generares sobre las prácticas

Se deben tener en cuenta los siguientes puntos a la hora de realizar las prácticas:

- Se debe aplicar el algoritmo sobre la misma instancia de problema y ejecutar sobre la misma máquina para: (a) comparar la bondad de los resultados de las distintas versiones con un mismo modelo de programación y entre modelos: secuencial (Práctica 0), OpenMP (Práctica 1), MPI (Práctica 2) e híbrido (Práctica 3); y (b) para hacer los cálculos de *speed-up* y eficiencia, y poder comparar tiempos (Prácticas 1, 2 y 3).

- Hay que indicar siempre en la memoria el número de núcleos (>= 6 cores físicos) del sistema computacional donde se han llevado a cabo los experimentos.

- Comentar el código suficientemente, sobre todo en lo referente a paralelismo.

- Para tomar tiempos en los experimentos, se recomienda ejecutar los programas desde la terminal de comandos de Linux directamente, y se deben cerrar otras aplicaciones o no interactuar de forma innecesaria con ellas durante las ejecuciones.