Práctica 3

Implementación distribuida de un algoritmo de equilibrado dinámico de la carga usando MPI

3.1. El problema del viajante y su solución mediante un algoritmo de acotación y ramificación.

El *problema del viajante de comercio* (*traveling salesman problem — TSP*) puede ser representado como un grafo dirigido compuesto de un conjunto de vértices (ciudades) y arcos etiquetados (distancias entre ciudades). Una solución optimizada al *TSP* es un camino de coste mínimo en el cual todos los vértices son visitados exactamente una vez (véase la figura 3.1).

El problema se resuelve mediante un algoritmo de *acotación y ramificación* (*Branch-and-Bound — BB*) que, dinámicamente, construye un árbol de búsqueda cuya raíz es el problema inicial (véase la figura 3.2) y sus nodos hoja son caminos entre todas las ciudades (no necesariamente de coste óptimo).

En los algoritmos *BB* se definen las siguientes funciones y variables de interés (véase la figura 3.2):

- $c(P)$ — *Coste* de un nodo: es la distancia del camino completo representado por P, si P es un nodo solución. En otro caso, $c(P)$ es el coste de una solución de coste mínimo en el sub-árbol cuya raíz es P.

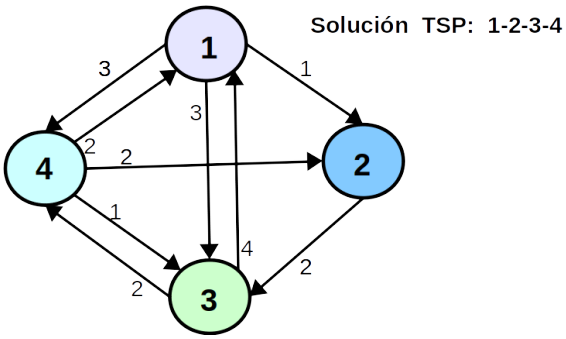

Figura 3.1: Grafo que representa un TSP con 4 ciudades

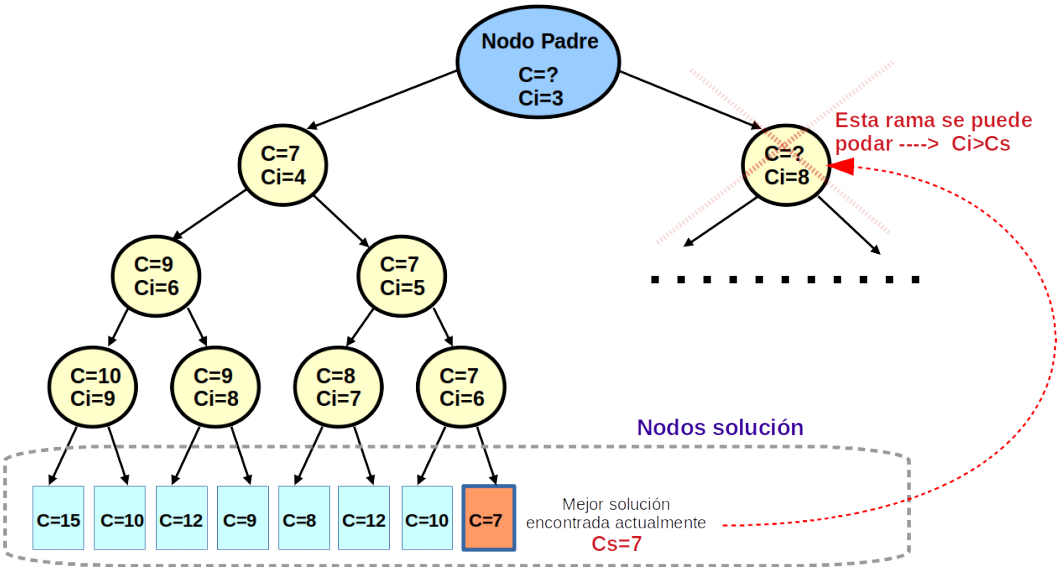

Figura 3.2: Árbol de búsqueda para un problema TSP

- $ci(P)$ — *Cota inferior*: es una cota inferior al coste de una solución al *TSP* para el (sub)problema P. Si P es un nodo solución, entonces $ci(P) = c(P)$.
- cs — *Cota superior*: es la longitud del camino completo más corto encontrado hasta el momento. Este valor debe ser una variable global del algoritmo y es utilizado para acotar el árbol de búsqueda.

3.1.1. Representación de los nodos y gestión del árbol de búsqueda

En principio, un problema TSP con N ciudades se representa como una matriz de adyacencia con $N \times N$ enteros (asumimos que las aristas del grafo son de tipo entero por simplicidad). El problema concreto a resolver se leerá de un archivo de texto. No obstante, una vez se dispone de dicha matriz que representa el problema a resolver, la información de un nodo de un árbol, incluyendo su cota inferior y las aristas que tiene fijadas, se puede representar usando únicamente un vector de $2N$ enteros.

Toda la gestión de los nodos del árbol y de la pila de nodos que usa el programa secuencial que se ha de paralelizar se puede realizar mediante funciones definidas en los archivos `libbb.h` y `libbb.cc`. A continuación se explica cómo se calcula la cota inferior de un nodo y cómo se expanden los nodos internos del árbol para realizar la búsqueda de soluciones. Estos contenidos se incluyen para describir cómo se realizan estos procedimientos, aunque la implementación paralela no requiere conocer estos detalles, ya que estos procedimientos se realizarán invocando funciones de `libbb.cc`.

Cálculo de la cota inferior

El cálculo de la cota inferior de un nodo del árbol de búsqueda se lleva a cabo obteniendo la *matriz de coste reducido*. Para ello introducimos las siguientes definiciones:

1. Una *fila* o *columna* de una matriz se dice *reducida* sii tiene al menos una entrada cero y las demás entradas son no negativas.

2. Una *matriz* se dice *reducida* sii todas sus filas y columnas están reducidas.

Si se resta un valor v a todas las entradas de una fila o columna de la matriz, el coste del (sub)problema representado por la matriz se reduce exactamente en v. Si P es un subproblema, entonces su cota inferior, $ci(P)$ es la suma de las cantidades restadas a sus filas y columnas para obtener la matriz de coste reducido. Un recorrido completo de costo mínimo continúa siéndolo tras la operación de sustracción. De esta forma, toda la información relativa a un subproblema está contenida en su matriz de coste reducido y en su valor de cota inferior.

Ejemplo. Supongamos que partimos de un problema A a partir del cual se obtiene una matriz reducida B:

$$A = \begin{pmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 1 & 4 & 7 & 16 & \infty \end{pmatrix} \qquad B = \begin{pmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{pmatrix}$$

La matriz B se obtiene sustrayendo 10,2,2,3,4 de las filas 1,2,3,4 y 5, así como 1 y 3 de las columnas 1 y 3. Al final el total sustraído es de 25. Esto significa que todos los recorridos completos del TSP para el grafo que representan las matrices A y B, tienen al menos 25 unidades ($ci(A) = 25$).

Construcción del árbol de búsqueda

Para un nodo del árbol de búsqueda A, se generan dos subproblemas independientes mediante la elección de un arco (i, j):

- El *hijo izquierdo* se obtiene mediante la inclusión del arco (i, j) en el camino de la solución, para lo que es necesario eliminar todos los demás árcos $(i, ?)$ y $(?, j)$ ($A[i, ?] = \infty$, $A[?, j] = \infty$). Si como consecuencia de la inclusión de (i, j) en el camino, i es la última ciudad a visitar y j es la inicial, entonces hemos encontrado un camino y la cota inferior del nodo solución obtenido es el coste de dicho camino ($ci(P) = c(P)$). Además, este valor pasa a ser una cota superior (cs) del coste de la solución al problema inicial, por lo que todos los nodos del árbol de búsqueda con $ci(P) \geq cs$ pueden ser descartados. Por otro lado, si todavía quedan ciudades por visitar, entonces el arco (j, i_0) (siendo i_0 la ciudad inicial) debe ser eliminado ($A[j, i_0] = \infty$) para evitar ciclos.

- El *hijo derecho* se obtiene mediante la exclusión del arco (i, j) ($A[i, j] = \infty$) en el camino de la solución.

Nos interesa obtener árboles derechos que puedan ser acotados pronto, para así reducir el espacio de búsqueda. Para ello utilizamos una heurística que elija un arco (i, j) que produzca un hijo derecho con un valor de cota inferior tan grande como sea posible. Si elegimos un arco con coste positivo, el valor de cota inferior del hijo derecho no se incrementará ya que su matriz de coste seguirá siendo reducida al eliminar la entrada (i, j). Si elegimos un arco con coste nulo, al eliminar dicha entrada quizás tengamos que restar alguna cantidad a las entradas de la fila i y de la columna j para obtener la matriz de coste reducido. El valor de cota inferior para el hijo derecho se incrementará en $\text{mín}_{k \neq j}\{M[i, k]\} + \text{mín}_{k \neq i}\{M[k, j]\}$, ya que ésto es lo que se necesita sustraer de la fila i y de la columna j para introducir un 0 en ambas. Esta heurística permite maximizar el valor de cota inferior para el subárbol derecho.

3.2. El programa secuencial.

Para abordar esta práctica, se partirá de un programa secuencial que implementa el algoritmo Branch-and-Bound para resolver problemas TSP de diferente tamaño.

La estructura del algoritmo paralelo va a ser muy similar al algoritmo secuencial ya que cada proceso ejecuta fundamentalmente la iteración principal del algoritmo secuencial, pero incluyendo llamadas a funciones auxiliares para repartir la carga (y detectar el fin) y para mantener actualizada la cota superior. Por ello, es muy importante entender bien la estructura del algoritmo secuencial, antes de abordar la implementación de su versión paralela y distribuida. Esto no significa que se tengan que conocer los detalles de cómo se gestiona el árbol de búsqueda y la pila de nodos (solo se necesita conocer algunos detalles sobre cómo se representan los nodos en la pila para poder enviar listas de nodos de un proceso a otro) pero sí es importante que se entienda cómo se explora el árbol en el ciclo principal del algoritmo.

Los archivos proporcionados son los siguientes:

- `bbseq.cc`: Implementación del algoritmo de Branch-and-Bound secuencial que puede servir como plantilla para implementar el algoritmo paralelo (bbpar.cc).
- `libbb.h`: archivo de cabecera para las funciones del algoritmo de Branch-and-Bound, de manejo de la pila de nodos. gestión del árbol de búsqueda y de la memoria dinámica.
- `libbb.cc`: Implementación de las funciones declaradas en `libbb.h`.
- `tspXX.X`: Archivos de ejemplo con problemas TSP. Se incluyen archivos de hasta 40 ciudades.

3.3. El algoritmo paralelo.

Tras un estudio de los aspectos de *descomposición* y *asignación*, llegamos a un diseño de programa paralelo con las siguientes características:

- El árbol de búsqueda es explorado en paralelo por parte de varios procesos (uno por procesador). Dichos procesos ejecutan el algoritmo *BB* sobre nodos diferentes del árbol.
- No existe ningún proceso *maestro* centralizado encargado de repartir nodos del árbol de búsqueda entre varios procesos *esclavos* que ejecutarían el algoritmo *BB*. En su lugar, se implementará un esquema de equilibrado de carga totalmente distribuido. El equilibrado de carga es realizado por el conjunto de procesos paralelos que mantienen una partición del árbol de búsqueda. Cada uno de los procesos se comunica con el resto de los procesos:
  1. para pedir nuevos nodos (cuando su parte del árbol de búsqueda haya sido ya explorada),
  2. para ofrecer parte de sus nodos (cuando reciba peticiones de otros procesos).
- La detección de la situación de fin se lleva a cabo mediante un algoritmo distribuido de paso de testigo en anillo. El programa paralelo terminará cuando ya no queden más nodos por explorar, es decir, la situación de fin se establecera cuando el mecanismo de equilibrado de carga no pueda suministrar más nodos a los procesos. Por ello, el algoritmo de detección de fin deberá integrarse con el mecanismo equilibrado de carga.

- La difusión de nuevos valores globales de cota superior entre los diferentes procesos se lleva cabo también mediante un esquema descentralizado. El objetivo es simular un par de operaciones asíncronas de lectura y escritura de valores de cota superior.

Nuestra implementación del algoritmo de BB paralelo consistirá en un único proceso por procesador. La utilización de varios *comunicadores* y de variantes asíncronas de las primitivas de paso de mensajes nos van a permitir implementar los diferentes requerimientos de comunicación de forma modular.

Para orientaros sobre el funcionamiento del algoritmo paralelo podéis visitar la página:

`https://lsi.ugr.es/jmantas/ppr/practicas/practicas.php?prac=prac03`

que presenta animaciones que ilustran gráficamente su comportamiento.

3.4. El programa principal.

De forma resumida, se puede considerar que un proceso con rango `id` ejecutará el siguiente algoritmo:

```
main ()  {
    U = INFINITO;           // inicializa cota superior

    if (id == 0) Leer_Problema_Inicial (&nodo);

    ... Difusión matriz del problema inical del proceso 0 al resto


    if (id != 0) {
        Equilibrar_Carga (&pila, &fin);
        if (! fin) Pop (&pila, &nodo);
    }

    while (! fin)  {                              // ciclo del Branch&Bound
        Ramifica  (&nodo, &nodo_izq, &nodo_dch);

        if (Solucion (&nodo_dch)) {
            if (ci (nodo_dch) < U)  U = ci (nodo_dch);     // actualiza c.s.
        }
        else {                    //  no es un nodo hoja
           if (ci (nodo_dch) < U)  Push (&pila, &nodo_dch);
        }

        if (Solucion(&nodo_izq)) {
            if (ci (nodo_izq) < U)  U = ci (nodo_izq);    // actualiza c.s.
        }
        else {                    // no es nodo hoja
            if (ci (nodo_izq) < U)  Push (&pila, &nodo_izq);
        }
```

```
        Difusion_Cota_Superior (&U);
        if (hay_nueva_cota_superior)   Acotar (&pila, U);

        Equilibrado_Carga  (&pila, &fin);
        if (! fin)  Pop (&pila, &nodo);
    }
}
```

Inicialmente sólo el proceso 0 lee la matriz del problema, por lo que se deberá difundir esta matriz para que la tengan todos los procesos.

Asumimos que el proceso 0 es el que expande el problema inicial. El resto de procesos llaman al procedimiento de equilibrado de carga, lo que provocará que tengan que realizar peticiones de trabajo al proceso 0. Según se vayan generando nuevos nodos, las peticiones de trabajo se irán atendiendo y los procesos obtendrán nodos para expandir.

En el ciclo principal del algoritmo, la expansión de un nodo genera dos nuevos nodos, `nodo_izq` y `nodo_dch`. Si un nodo generado es una solución al problema se comprueba si su coste mejora la cota superior actual, `U`, en cuyo caso se actualiza ésta. Si el nodo generado no es una solución y su cota inferior es menor que la cota superior actual, se almacena para su posterior expansión.

Cada proceso mantiene localmente una *pila* de nodos para expandir. La elección de una estructura de datos tipo pila permite que el árbol de búsqueda sea explorado por cada proceso de acuerdo con un esquema *primero en profundidad*, para así encontrar soluciones lo antes posible. Obsérvese que en el proceso, el almacenamiento del nodo derecho se lleva a cabo antes que el del nodo izquierdo, con el fin de expandir primero el nodo izquierdo y llegar antes a una solución prometedora.

Cuando un proceso produce un nuevo valor de cota superior, dicho valor deberá ser difundido para que sea conocido por los demás procesos. Por ello, cada proceso deberá comprobar si algún otro proceso generó un nuevo valor de cota superior. El procedimiento `Difundir_Cota_Superior` implementa estas cuestiones.

Si se generó un nuevo valor de cota superior (local o remotamente) podremos eliminar de la pila aquellos nodos con valor de cota inferior mayor que la nueva cota superior (procedimiento `Acotar`).

Antes de extraer un nuevo nodo de la pila (procedimiento `Pop`), la llamada a `Equilibrar_Carga` permitirá:

- pedir nodos a otros procesos si la pila está vacía,
- ceder nodos a otros procesos si se reciben peticiones y hay nodos para dar, o
- detectar la situación de fin.

3.5. Equilibrado de carga.

Cuando la pila de un proceso (con identificador id está vacía, se envía un mensaje de *petición de trabajo*, indicando el identificador del proceso, id, como el origen de la petición. Para ello, el proceso id mandará este mensaje al siguiente proceso $(id + 1) \mod P$ en el anillo.

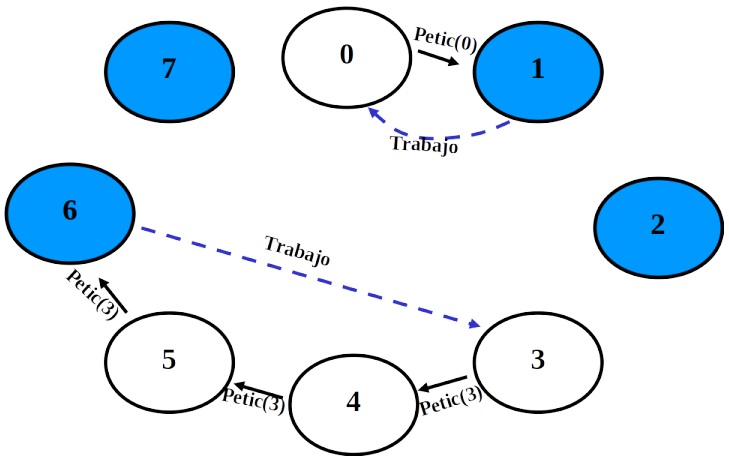

Figura 3.3: Funcionamiento algoritmo de equilibrado de carga en anillo

A continuación el proceso entra un bucle, del que saldrá cuando haya conseguido trabajo para seguir ejecutándose, o bien cuando, al fallar sus peticiones, se detecte la situación de fin. El proceso espera recibir un mensaje de respuesta. Si recibe un mensaje de trabajo, la petición tuvo éxito, almacenándose los nodos recibidos en la pila local. Si recibe un mensaje de petición (desde el proceso anterior, $(id - 1 + P) \mod P$), esta petición no podrá ser atendida al estar vacía la pila del proceso, por lo que dicho mensaje es reenviado al siguiente proceso, $(id+1) \mod P$. Además, si el origen de la petición es el mismo proceso, la petición habrá viajado por todo el anillo sin poder ser atendida, por lo que podríamos encontrarnos en una situación de fin (véase sección 3.6).

Si no se llegó a la situación de fin, el proceso sondea mensajes de petición de trabajo de otros procesos. Si no hay mensajes pendientes el procedimiento de equilibrado de carga acaba y el proceso puede continuar ejecutando el algoritmo de BB. Mientras haya peticiones de trabajo, el proceso intenta dividir su pila local para ceder nodos al proceso solicitante. Si la pila no tiene nodos suficientes para ceder, reenvía las peticiones al proceso siguiente (podemos decidir no ceder nodos si el tamaño de la pila está por debajo de un umbral, por ejemplo: la función que divide la pila falla si no hay al menos dos nodos en la pila).

```
Equilibrado_Carga (tPila *pila, bool *fin)  {
    if (Vacia (pila))  {  // el proceso no tiene trabajo: pide a otros procesos
        Enviar peticion de trabajo al proceso (id+1)%P;
        while (Vacia (pila)  &&  ! fin) {
            Esperar mensaje de otro proceso;
            switch (tipo de mensaje) {
                case PETIC :                    // peticion de trabajo
                    Recibir mensaje de peticion de trabajo;
                    if (solicitante == id) {      // peticion devuelta
                        Reenviar peticion de trabajo al proceso (id+1)%P;
                        Iniciar deteccion de posible situacion de fin;
                    }
                    else         // peticion de otro proceso: la retransmite al siguie
                        Pasar peticion de trabajo al proceso (id+1)%P;
                    break;
```

```
                case NODOS :            // resultado de una peticion de trabajo
                    Recibir nodos del proceso donante;
                    Almacenar nodos recibidos en la pila;
                    break;
            }
        }
    }
    if (! fin)  {  // el proceso tiene nodos para trabajar
        Sondear si hay mensajes pendientes de otros procesos;
        while (hay mensajes)  { // atiende peticiones mientras haya mensajes
            Recibir mensaje de peticion de trabajo;
            if (hay suficientes nodos en la pila para ceder)
                Enviar nodos al proceso solicitante;
            else
                Pasar peticion de trabajo al proceso (id+1)%P;
            Sondear si hay mensajes pendientes de otros procesos;
        }
    }
}
```

3.6. Detección de fin.

El algoritmo de BB acabará cuando el espacio de búsqueda haya sido explorado, es decir, cuando no queden más nodos para ramificar. El programa debe acabar cuando todos los procesos agoten sus pilas locales y no puedan obtener nodos de otros procesos para seguir trabajando.

3.6.1. Algoritmo de terminación de Dijkstra.

Supongamos un conjunto de procesos organizados en anillo. Un proceso puede estar en dos estados: *activo* y *pasivo*. Cuando el proceso P_0 pasa al estado pasivo envía un testigo al anterior proceso del anillo: P_{n-1}. Cuando un proceso recibe el testigo, si su estado es pasivo, lo enviará al proceso anterior, o, si su estado es activo, lo mantendrá hasta que pase al estado pasivo. Cuando el proceso P_0 reciba el testigo de nuevo, sabrá que todos los demás procesos han terminado.

Este esquema no funciona si un proceso en estado pasivo puede ser reactivado por un mensaje de trabajo enviado por otro proceso (véase la figura 3.4). Supongamos que el proceso P_i envía un mensaje de trabajo al proceso P_j (en estado pasivo), para después pasar al estado pasivo. Cuando P_i reciba el testigo, lo pasará al proceso P_{i-1}. Si $i > j$, el mensaje de trabajo llegará al proceso P_j antes que el testigo. Dicho mensaje de trabajo reactivará al proceso P_j, por lo que al recibir el testigo, lo retendrá. Sin embargo, si $i < j$, y el testigo visitó al proceso P_j antes de recibir el mensaje de trabajo, P_j podrá ser reactivado y P_i recibirá el testigo, creyendo que todos los procesos P_k con $k > i$ están pasivos (entre ellos P_j). En este caso se podría producir una situación de fin errónea.

Para evitar este escenario, se propone la siguiente modificación: cada proceso, además de un estado, tiene un color: *blanco* o *negro* (inicialmente blanco). El testigo, también tiene un

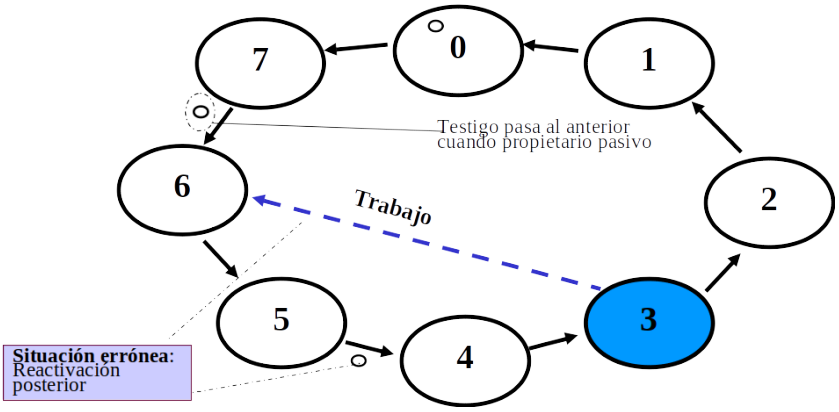

Figura 3.4: Funcionamiento anómalo en la detección de fin en anillo

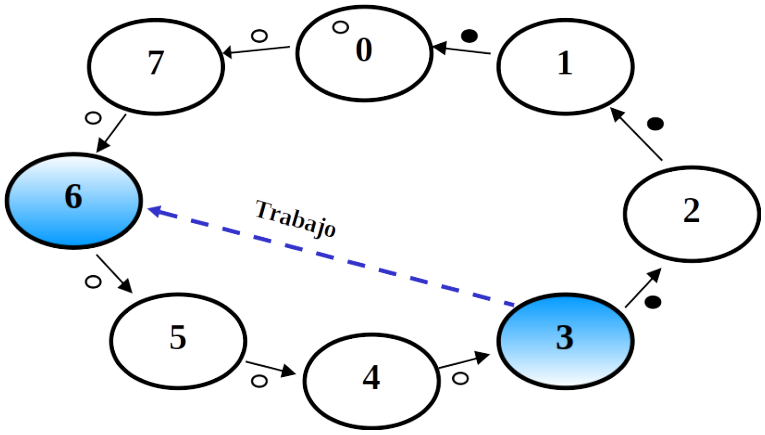

Figura 3.5: Funcionamiento del algotirmo de Dijkstra para la detección de fin en anillo

color (inicialmente blanco).

Inicialmente el proceso P_0 posee el testigo de detección de fin. Cuando el proceso P_0 pasa al estado pasivo, envía un testigo blanco al proceso P_{n-1}. Cuando un proceso P_i envía un mensaje de trabajo a otro proceso P_j, siendo $i < j$, cambia su color a negro. Un proceso pasivo con color blanco reenviará el testigo sin cambiar el color de éste. Un proceso pasivo con color negro reenviará el testigo con color negro; tras ello, el proceso cambiará su color a blanco. El algoritmo terminará cuando el proceso P_0, siendo su color blanco, reciba un testigo blanco. Si el proceso P_0 recibe un testigo negro, volverá a enviarlo con color blanco. Si el mensaje de trabajo que provocó que el testigo fuera cambiado a negro reactivó a algún proceso, el testigo será retenido por el proceso activo hasta que pase al estado pasivo. La figura 3.5 ilustra este procedimiento.

```
P (id)
    . . .
    Esperar evento;
    switch (tipo_evento) {
        case  MENSAJE_TRABAJO :
            estado = ACTIVO;
```

```
        case  MENSAJE_PETICION :
            if  (hay trabajo para ceder)  {
                j = origen(PETICION);
                Enviar TRABAJO a P(j);
                if  (id < j)  mi_color = NEGRO;
            }
        case  MENSAJE_TOKEN :
            token_presente = TRUE;
            if  (estado == PASIVO)  {
                if  (id==0   &&  mi_color == BLANCO  &&  color(TOKEN) == BLANCO)
                    TERMINACION DETECTADA;
                else {
                    if  (id==0)
                        color(TOKEN) = BLANCO;
                    else if  (mi_color == NEGRO)
                        color(TOKEN) = NEGRO;
                    Enviar TOKEN a P(id-1);
                    mi_color = BLANCO;
                    token_presente = FALSE;
                }
            }
        case TRABAJO_AGOTADO :
            estado = PASIVO;

            if  (token_presente)
                if  (id==0)
                    color(TOKEN) = BLANCO;
                else if  (mi_color == NEGRO)
                          color(TOKEN) = NEGRO;
                Enviar TOKEN a P(id-1);
                mi_color = BLANCO;
                token_presente = FALSE;
    . . .
```

3.6.2. Integración del algoritmo de detección de fin en el procedimiento de equilibrado de carga.

Un proceso se considerará *pasivo* cuando, al agotar su pila local y emitir una petición de trabajo, ésta sea devuelta al no poder ser atendida por ningún otro proceso.

Cuando un proceso pase al estado pasivo, volverá a enviar el mensaje de petición de trabajo al siguiente proceso, y además, si tiene el testigo, reiniciará la detección de fin enviándolo al proceso anterior en el anillo.

Cuando un proceso activo reciba el testigo de detección de fin, lo guardará para reenviarlo cuando pase al estado pasivo.

Cuando el proceso 0 detecte la situación de fin, recogerá los mensajes de petición de trabajo que haya circulando, y emitirá un segundo testigo de confirmación para que los demás

procesos conozcan dicha situación y puedan terminar correctamente. Los demás procesos, según vayan recibiendo el testigo de confirmación, lo retransmitirán al siguiente proceso y terminarán. El proceso 0 terminará cuando reciba este testigo.

3.7. Difusión de la cota superior.

La difusión y mantenimiento de valores globales de cota superior se lleva a cabo haciendo circular dichos valores por los procesos en una estructura de comunicación en anillo. Es conveniente que la difusión de la cota superior se realice de forma asíncrona con respecto a la computación local de cada proceso. No es necesario que todos los procesos tengan en cada momento el mismo valor de cota superior: en este algoritmo la consistencia no afecta a la corrección. Solamente se requiere que los valores estén tan actualizados como sea posible para así optimizar la búsqueda.

Cuando un proceso obtiene un nuevo valor de cota superior, deberá difundir dicho valor enviándolo al siguiente proceso del anillo. A continuación, el proceso comprueba la existencia de mensajes pendientes desde el proceso anterior del anillo. Si hay algún mensaje de cota superior, lo recibe, actualiza su valor local al menor de ambos, y reenvía el mensaje al proceso siguiente del anillo.

Para evitar un número excesivo de mensajes y llegar así a una situación de interbloqueo, necesitamos alguna forma de asegurar que cada valor de cota superior dé una sola vuelta al anillo, y que en cada momento, haya como máximo un solo mensaje por proceso circulando. Para ello, cada valor de cota superior enviado al anillo irá acompañado del identificador del proceso que lo generó. Cuando un proceso reciba un valor de cota superior enviado por él mismo, no lo reenviará hacia el siguiente proceso del anillo. Además, cada proceso deberá registrar si está pendiente de recibir desde el proceso anterior del anillo algún mensaje enviado por él mismo:

un proceso no difundirá un nuevo mensaje de cota superior (encontrado por él mismo) mientras no reciba el mensaje anteriormente enviado por él mismo. Cuando un proceso recibe un valor desde el proceso anterior del anillo, actualiza su valor local y lo reenvía si su origen era otro proceso. Si el origen del mensaje era el mismo proceso que lo ha recibido, se considera terminada la difusión del valor de cota superior, habilitándose la difusión de nuevos valores.

```
Difusion_Cota_Superior ()
{
    if (difundir_cs_local  &&  ! pendiente_retorno_cs)  {
        Enviar valor local de cs al proceso (id+1)%P;
        pendiente_retorno_cs = TRUE;
        difundir_cs_local = FALSE;
    }
    Sondear si hay mensajes de cota superior pendientes;
    while  (hay mensajes)
    {
        Recibir mensaje con valor de cota superior desde el proceso (id-1+P)%P;
        Actualizar valor local de cota superior;
```

```
        if  (origen mensaje == id  &&  difundir_cs_local)  {
            Enviar valor local de cs al proceso (id+1)%P;
            pendiente_retorno_cs = TRUE;
            difundir_cs_local = FALSE;
        }
        else if  (origen mensaje == id  &&  ! difundir_cs_local)
            pendiente_retorno_cs = FALSE;
        else              //  origen  mensaje == otro proceso
            Reenviar mensaje al proceso (id+1)%P;

        Sondear si hay mensajes de cota superior pendientes;
    }
}
```

3.8. Aspectos de implementación con MPI.

Para implementar nuestro programa de forma modular podemos seguir algunas recomendaciones.

Además del programa principal, implementaremos dos funciones separadas, una para el equilibrado de carga/detección de fin, y otra para la difusión de la cota superior. Para evitar interferencias entre los mensajes de estas funciones, utilizaremos *comunicadores* separados para ellas. ésto nos permite, por ejemplo, esperar mensajes relacionados con el equilibrado de carga (utilizando `MPI_ANY_SOURCE` y/o `MPI_ANY_TAG`) independientemente de los mensajes de difusión de la cota superior.

Desde el punto de vista estratégico, se recomienda seguir la siguiente secuencia de pasos (el orden es importante):

1. Comprender bien el programa `bbseq.cc`.
2. Implementar la función de equilibrado de carga sin incluir la detección de fin. No se detectaría el fin, por lo que la finalización debería ser con una interrupción por parte del usuario. En cualquier caso, se debe comprobar que el número de iteraciones que hace cada proceso es similar al que hace el resto de procesos. Se recomienda empezar haciendo pruebas con tamaños de problema pequeños o medianos y solo 2 procesos.
3. Incorporar la detección de fin a la función de equilibrado de carga.
4. Implementar la función de difusión de cota superior.

Dentro de cada función, utilizaremos *etiquetas* diferentes para cada tipo de mensaje. Así, en la función de equilibrado de carga podríamos tener los siguientes tipos de mensaje: *petición*, *trabajo*, *testigo de detección de fin*, y *fin detectado*. En la función de difusión de cota superior podríamos utilizar la etiqueta para identificar al origen del mensaje de cota superior.

Es importante mantener adecuadamente la información de estado de cada proceso. El estado de un proceso podría incluir:

- si se ha llegado o no a la situación de fin,

- si el proceso está en posesión de testigo de detección de fin,
- el color del proceso,
- el color del testigo de detección de fin (si el proceso posee el testigo),
- si se ha encontrado un nuevo valor de cota superior,
- si está pendiente de retorno algún mensaje de cota superior previamente enviado por el proceso,
- ...

Prevención de interbloqueos.

El algoritmo de BB paralelo muestra un esquema de comunicación asíncrono que debe ser implementado utilizando paso de mensajes. La situación de interbloqueo se produce cuando se forma un ciclo de procesos en el que cada proceso está bloqueado intentando enviar un mensaje al siguiente.

Para evitar los interbloqueos necesitamos limitar el numero máximo de mensajes que los procesos pueden enviar. En el caso del equilibrado de carga, como máximo hay un mensaje por proceso: una petición de trabajo, que, en caso de ser atendida, es cambiada por un mensaje de trabajo. Simultáneamente con los mensajes de trabajo, podría haber circulando un testigo de detección de fin.

En el caso de la difusión de la cota superior, como máximo hay tantos mensajes como procesos, ya que nunca un proceso hace circular simultáneamente más de un valor de cota superior.

3.9. Ejercicios propuestos.

El alumno deberá implementar el algoritmo paralelo de Branch-and-Bound siguiendo las indicaciones del guión de prácticas. Además del código fuente, que incluirá un Makefile, deberá entregarse un archivo pdf con un estudio experimental con las siguientes medidas y cálculos relacionados:

- Medidas de tiempo de ejecución monoprocesador del programa secuencial (bbseq.cc) para 20, 26, 28, 29 y 40 ciudades.
- Medidas de tiempo de ejecución paralelo sobre 2, 3 y 4 procesadores físicos para los mismos problemas usados en las medidas secuenciales.

  Se tomarán medidas para dos casos diferentes:

  a) Incluyendo la difusión de la cota superior.
  b) Excluyendo la difusión de la cota superior.
- Para cada una de las ejecuciones realizadas (número de ciudades, número de procesadores, sin/con difusión de cota) se informará del número de nodos explorados (iteraciones BB realizadas) de cada proceso.
- Para cada configuración de ejecución también se determinará la ganancia en velocidad con respecto al algoritmo secuencial.

3.9.1. Indicaciones para chequear la calidad de la solución paralela obtenida

Como indicaciones de chequeo, lo esperado es que se den las siguientes evidencias:

- Para problemas de tamaño suficientemente grande (a partir de 20), se obtendrá una ganancia en velocidad cercana al número de procesadores o, en algunos casos superior.
- Si la difusión de cota está bien implementada, el programa paralelo sin difusión de cota explorará bastantes más nodos que la versión con difusión de cota.
- Si el equilibrado de carga funciona correctamente, para problemas con al menos 10 ciudades, el número de nodos explorados por cada proceso es muy próximo (considerando el total de nodos explorados) a la media de nodos por proceso.