Universidad Nacional Autónoma de México

Facultad de Ciencias

Computación distribuida

Práctica 2

Algoritmos básicos

Baños Mancilla Ilse Andrea - 321173988

Flores Arriola Edson Rafael - 423118018

Rivera Machuca Gabriel Eduardo - 321057608

Reporte de práctica

La práctica se desarrolló en 3 partes. Primero se realizaron las funciones *convergecast* y *busqueda* en el laboratorio de la materia. Después se realizaron las funciones auxiliares *k_merge* y *cuadricula*, las cuales fueron fáciles de implementar pues *k_merge* es una generalización del algoritmo merge de mergeSort, algoritmo estudiado en la materia de Estructuras de Datos y Análisis de algoritmos. Finalmente se realizaron las funciones *nodo_sort* y *nodo_generador* con ayuda de pseudocódigos proporcionados por el ayudante de laboratorio y el profesor de la materia.

- **k_merge(arreglo)**

Para implementar esta función se tomó como base el algoritmo merge de mergeSort que dados dos arreglos S1 y S2 y sus respectivas longitudes n1 y n2, toma dos índices i y j (ambas inicializadas en 0), donde i recorre S1 y j recorre S2, así va comparando los elementos y avanzando en los índices hasta que i o j llega a n1 o n2 respectivamente.

En k_merge se tienen k arreglos, pero la idea de ir recorriendo con índices e irlos incrementando cuando se encuentra al valor mínimo se mantiene. Por esta razón se decidió usar un arreglo de índices (inicializado en 0) para irlos incrementando según corresponda, es decir, si el elemento actual en el i-esimo arreglo es el mínimo , entonces solo el i-esimo índice del arreglo de índices se va a incrementar y así poder verificar todos los elementos de cada arreglo.

Además, como se tienen k arreglos, se debe buscar el mínimo entre los elementos actuales de cada uno, por esta razón se usa el for, que busca el elemento mínimo entre los actuales de los arreglos y el índice del arreglo que lo contiene. Sin embargo, puede existir el caso en el que uno de los arreglos ya haya sido recorrido por completo, en este caso ya no tiene sentido seguir incluyéndolo en las comparaciones, por eso se pone la condición `if(indices[i]<len(arreglo[i]))`.

Una vez que se tiene el elemento mínimo y el índice del arreglo en el que se encuentra se procede a agregar el elemento al arreglo de resultado y se usa el índice para aumentarlo en el arreglo de índices.

Como no se sabe cuántas iteraciones se deben hacer para pasar todos los elementos de los k arreglos, el procedimiento de encontrar el mínimo y agregarlo se va a repetir siempre y cuando el índice del mínimo sea válido, es decir, un entero entre 0 y len(arr[i]) para alguna i y con arr el arreglo de entrada. Por esta razón el índice del mínimo se inicializa en -1 cada iteración.

- **cuadricula(arr,cantidad_nodos)**

Para cumplir con esta condición: *Si n > longitud(B1) entonces los elementos del arreglo de salida que no alcancen elemento serán arreglos vacíos.* Se usó `cuadricula = [[] for _ in range(cantidad_nodos)]` lo que genera un arreglo de tamaño cantidad_nodos y lleno de arreglos vacíos.

Para obtener la longitud de cada subarreglo se hace `num_elem=longitud // cantidad_nodos` ("//" porque nos interesa aplicar la función piso, es decir, hacer división emtera). Pero si la división no es exacta entonces habrán subarreglos con elementos de más, para esto se obtiene la cantidad de elementos restantes `elem_restantes = longitud % cantidad_nodos`. Si j=elem_restntes entonces estos elementos serán agregados uno por uno en los primeros j subarreglos, la línea `extra = 1 if i < elem_restantes else 0` ayuda a indicar si se debe agregar un espacio extra al subarreglo o no.

Finalmente se llevan dos índices: inicio y fin, los cuales nos ayudarán a iterar el arreglo de entrada para copiar sus subarreglos.

- **Algoritmo 1. Nodo Generador.**

Para este algoritmo vamos a desarrollarlo usando el pseudocódigo dado en el pdf, este aparece como **Árbol generador**, vamos a desglosar el desarrollo del algoritmo en python y se mencionará un problema que surgió pero se pudo resolver, primero se explicará cómo se integra el algoritmo y luego qué cosas se modifican.

Primero se inicializa el nodo al igual de que la clase nodo, además se asignan los atributos propios del algoritmo como padre(con valor None), hijos y mensajes esperados (con valor del tamaño de los vecinos), aquí se tuvo el primer problema pues hijos se establece como list(), sin en embargo al implementar el algoritmo marcaba errores entonces se modificó list() por []. Teniendo esto, el algoritmo empieza revisando si el nodo que recibe es el nodo distinguido, es decir, la raíz del árbol, esto lo hacemos revisando el id del nodo, si este es igual a 0 entonces es el nodo distinguido, en caso que sea el nodo distinguido vamos a establecer su padre como sí mismo, establecemos de nuevo mensaje esperados con el tamaño de sus vecinos y mandamos el mensaje GO a sus vecinos, aquí estuvo el segundo problema que al resolverlo aquí se va a suponer que los siguientes mensajes van a tener la misma estructura, al enviar solamente GO perdíamos información como quien enviaba el mensaje y valores extra que se pedía, entonces se crea una variable mensaje como: mensaje = (tipo_de_mensaje, id_remitente, valor_del_mensaje), en este caso para el mensaje que envía el nodo distinguido se establece como, mensaje = (GO, self.id_nodo,

None). Después, entramos en un ciclo while para suponer la entrada de mensajes y salida, aquí al iniciar se desempaqueta el mensaje que tenga el nodo, guardamos los datos como tipo, remitente y valor, con esto se sigue, el pseudocódigo dice que si un nodo p_j recibe el mensaje de tipo GO entonces si el nodo no tiene padre, el padre del nodo se establecerá como el remitente y los mensajes esperados disminuye en 1, al disminuir los mensajes esperados se revisa si su valor es igual a 0, de ser así creamos el mensaje como mensaje = (BACK, self.id_nodo, self.id_nodo) y este mensaje se envía al padre pero como la función recibe solamente tuplas entonces lo mandamos como [padre]; en el caso contrarío de que mensajes esperados no sea 0, vamos a mandar el mensaje = (GO, self.id_nodo, None) a sus vecinos menos al padre, para esto creamos una nueva tupla en la que excluimos al padre o al remitente, teniendo esto se manda el mensaje a los vecinos. Ahora, en el caso de que el nodo que recibe el tipo de mensaje GO sí tenga padre lo que se hace es crear el mensaje = (BACK, self.id_nodo, [remitente]). Finalmente, en caso de que el tipo de mensaje sea BACK, se disminuye en 1 los mensajes esperados y se revisa si valor es diferente a None, recordemos que los únicos mensajes que tienen valor diferente a None son los que se envían hacia los remitentes o los padres, entonces el que recibe el mensaje va a agregar a la tupla de hijos el id del remitente para guardarlo como hijo. Ahora se revisa, si los mensajes esperados ya es igual a 0, entonces se revisa si el nodo es su propio padre, e igual recordemos que el único que cumple esta condición es el nodo distinguido que actúa como la raíz, así no tiene un padre como tal, si este nodo no es su mismo padre entonces manda el mensaje = (BACK, self.id_nodo, self.id_nodo) a su padre, así finaliza el algoritmo que lo que hace es mandar un mensaje desde la raíz del árbol hasta las hojas y cuando llega a las hojas regresa un mensaje, cada que se recibe un mensaje de regrese agrega a los remitentes a los hijos y así se puede obtener la información del árbol.

- **NodoSort**

Para implementar este algoritmo se usó el siguiente pseudocódigo visto en clase:

```
A=arreglo a ordenar
n = longitud de A
p = número de nodos

algoritmo coordinador
begin
tam=n/p
for i =1 to p-1 do:
      izq = (i-1)*tam
      der = (i*tam)-1
      Ai=A[izq,der]
      send(i,Ai)
end for
izq = (p-1)*tam
der = n-1
A0=A[izq,der]
sort(A)
for i=1 to p-1 do:
      recv(i,A)
end for
A=k-merge(p,A0,A1,...,Ap-1)
end

algoritmo trabajador
begin
recv(0,A)
sort(A)
send(o,A)
end
```

Como el algoritmo se divide en dos partes, uno para el nodo coordinador y otro para los nodos trabajadores, se usó un if para decidir qué parte del algoritmo se ejecutará:

```
if self.id_nodo == 0:
…… implementación del coordinador
else
…… implementación de los trabajadores
```

Para la parte del nodo coordinador, gracias a que se implementó la función cuadrícula que divide un arreglo en p partes, la parte resaltada en rosa se omitió y se cambió por:

```
subarreglos = cuadricula(arr, p)
for i in range(1, p):
      yield env.timeout(TICK)
      self.canal_salida.envia(("ORDENAR", subarreglos[i]), [i])
```

Lo cual hace lo mismo aunque separado, primero divide el arreglo y después el coordinador envía la “orden” ORDENAR y el subarreglo i al trabajador i.

Como necesitamos que el nodo coordinador ordene el primer subarreglo, hacemos:

```
arr_0 = subarreglos[0]
arr_0.sort()
```

Después se reciben los subarreglos ordenados por los nodos trabajadores:

```
for i in range (1, p):
            #recv(i,Ai)
            orden, arr_i = yield self.canal_entrada.get()
            if orden == "DONE":
                  arr_aux.append(arr_i)
```

Si se recibió el mensaje DONE quiere decir que el arreglo fue ordenado correctamente y se guardan en un arreglo auxiliar para después usar k_merge y juntar todos los subarreglos en uno solo ordenado. Es importante mencionar que el subarreglo ordenado por el nodo coordinador ya se encuentra en el arreglo auxiliar.

Ahora para la parte de un nodo trabajador, simplemente se recibe el mensaje del coordinador con el arreglo a ordenar, se ordena y después se devuelve al coordinador.

```
orden, arr_trabajo = yield self.canal_entrada.get()
            if orden == "ORDENAR":
            #sort(A)
            arr_trabajo.sort()
            #send(0,A)
            yield env.timeout(TICK)
            self.canal_salida.envia(("DONE", arr_trabajo), [0])
```

Si se recibió la orden de ORDENAR, entonces el nodo ordena el arreglo recibido y después lo vuelve a enviar al coordinador con el mensaje DONE indicando que se pudo ordenar correctamente.

- **convergecast**

El objetivo de este algoritmo es hacer llegar la información de las hojas de un árbol a la raíz. Puesto que se supone una topología de estrella, se puede utilizar la topología completa del sistema como árbol generador de este. Sin embargo, podemos usar NodoGenerador para obtener el árbol generador en topologías con ciclos.

Para su implementación, hacemos que cada nodo espere los resultados de sus hijos para después tomarlos en cuenta y ejecutar su función `f(val_set_i)`. Seguidamente, cada nodo envía los resultados a su padre, terminando cuando la raíz ejecuta su propia función.

En nuestra implementación, el nodo raíz `(id_nodo == 0)` inicializa el algoritmo enviando mensajes `("INIT", id, ∅)` a sus vecinos. Cada nodo luego define a quién envió `INIT` como su padre. Si tiene hijos, reenvía el `INIT`. De lo contrario envía un mensaje a su padre `("BACK", id, {valor})`.

Así, al recibir los mensajes, cada nodo `i` añade los valores a `val_set_i`. Sí `i` es raíz, aplica la función `f(val_set_i)`. Si no, se repite el proceso.

- **busqueda**

El objetivo del algoritmo es realizar una búsqueda en paralelo a través de la topología del sistema. Para hacerlo, hacemos uso de un coordinador que divide el sistema en sub-arreglos y los distribuye entre el resto de nodos. Cada nodo revisa localmente si el elemento buscado se encuentra en su subarreglo e informan al coordinador con un mensaje de éxito (`FOUND`) o fracaso `(NOT_FOUND)` de acuerdo al resultado de la búsqueda. El coordinador entonces combina los resultados y si al menos un nodo encontró el elemento, la búsqueda termina con éxito.

En nuestra implementación concreta, el coordinador es el nodo 0, quien usa la función auxiliar `cuadricula(arr, n)` para dividir el arreglo en `n` partes (siendo `n` el número de vecinos + 1). Además, este retiene la primera parte y reparte las demás con mensajes `("GO", subarreglo, elemento)`.

Así, cada nodo recibe un mensaje `GO`. y recorre su sub-arreglo local buscando el elemento buscado. Si lo encuentra, envía (`"FOUND", arr, elemento`) al coordinador. De lo contrario, responde con (`"NOT_FOUND", arr, elemento`).

Si el coordinador recibe al menos un mensaje con `FOUND`, el coordinador hace `self.contenido = True`. De lo contrario, hace `self.contenido = False`.