DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Vectorización SIMD de BFS

Guido Tagliavini Ponce
Universidad de Buenos Aires
guido.tag@gmail.com

Índice

1. Abstract

En este trabajo presentamos una vectorización del algoritmo tradicional de búsqueda sobre grafos BFS, utilizando en el modelo de cómputo SIMD (Single Instruction, Multiple Data). La complejidad del algoritmo implementado es $\mathcal{O}(n^2)$ donde n es la cantidad de nodos del grafo. Si bien la performance a nivel práctico es inferior a una implementación de BFS en serie con complejidad lineal en el tamaño del grafo, la vectorización que presentamos es novedosa en el sentido de que alcanza a operar sobre 128 nodos del grafo en simultáneo, y deja abierta la posibilidad de utilizar varios nucleos de procesamiento para acelerar el cómputo.

2. Background

2.1. Teoría de grafos

Haremos un repaso breve de algunos conceptos elementales de teoría de grafos.

Definición 2.1. Un grafo dirigido (o digrafo) G es una tupla (V, E) donde V es un conjunto finito y no vacío de elementos llamados nodos, y $E \subset \{(v, w) \in V \times V : v \neq w\}$ es un subconjunto de pares ordenados de vértices, llamados aristas.

Por ejemplo, en el digrafo de la Figura 1 se tiene $V = \{v_1, v_2, v_3, v_4\}$ y
$E = \{(v_1, v_2), (v_1, v_3), (v_2, v_1), (v_3, v_2), (v_3, v_4)\}$.

Definición 2.2. Dados dos nodos $v, w \in V$, decimos que v incide en w si $(v, w) \in E$.

Por claridad, a un par ordenado (v, w) también lo notamos $v \to w$.

Definición 2.3. Dados $v, w \in V$, decimos que v y w son adyacentes si $v \to w \in E$ ó $w \to v \in E$.

Definición 2.4. Un camino dirigido en G es una secuencia de nodos $\langle v_1, \cdots, v_n \rangle$ tal que $v_1 \to v_2 \in E, \cdots, v_{n-1} \to v_n \in E$.

En el digrafo de la Figura 1, $\langle v_1, v_2, v_1 \rangle$ y $\langle v_3, v_2, v_1, v_3 \rangle$ son dos caminos dirigidos.

Definición 2.5. Dados $v, w \in V$, decimos que w es alcanzable desde v si existe un camino dirigido que comienza en v y termina en w.

En el digrafo de la Figura 1, v_4 es alcanzable desde v_2 y v_1 es alcanzable desde v_3. Por el contrario, v_2 no es alcanzable desde v_4.

Definición 2.6. Dados $v, w \in V$ tales que existe un camino de v a w, se define la distancia $d(v, w)$ de v a w como la cantidad de aristas del camino más corto de v a w.

Por ejemplo, en el digrafo de la Figura 1, $d(v_3, v_2) = 1$ y $d(v_2, v_3) = 2$. En particular, esto muestra que la distancia en grafos dirigidos no es reflexiva, i. e., $d(v, w)$ no necesariamente es igual a $d(w, v)$.

Definición 2.7. Dado $w \in V$ y un subconjunto no vacío $S \subset V$, definimos la distancia $d(S, w)$ de S a w como

$$d(S, w) = \min_{v \in S} d(v, w)$$

En el digrafo de la Figura 1, $d(\{v_1, v_2\}, v_3) = \min\{d(v_1, v_3), d(v_2, v_3)\} = 1$.

En este trabajo, todos los grafos de los que hablemos serán dirigidos, por lo que omitiremos la aclaración sobre la dirección.

2.2. Representación de un grafo

Para representar un grafo en una computadora se utilizan, tradicionalmente, dos estructuras: listas de adyacencia y matriz de adyacencia.

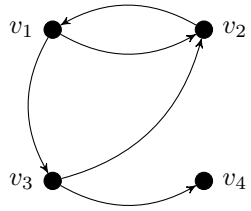

Figura 1: Grafo dirigido

2.2.1. Listas de adyacencia

Dado un grafo $G = (V, E)$, con $V = \{v_1, \cdots, v_n\}$, una representación de G en forma de listas de adyacencia es un arreglo $Adj[1 \cdots n]$, tal que $Adj[i]$ es una lista que contiene exactamente (y sin repeticiones) los nodos en los que incide v_i.

A modo de ejemplo, consideremos el grafo de la Figura 1. Aquí Adj es un arreglo de 4 elementos, y vale

$$\begin{aligned} Adj[1] &= 2 \to 3 \to \textsc{Nil} \\ Adj[2] &= 1 \to \textsc{Nil} \\ Adj[3] &= 2 \to 4 \to \textsc{Nil} \\ Adj[4] &= \textsc{Nil} \end{aligned}$$

Observar que hemos enumerado los nodos en la forma $v_1, \cdots, v_n$ y, posteriormente, para referirnos al nodo v_i utilizamos el número entero i que es su posición en dicho orden. Esta forma de asociar nodos a números enteros será utilizada frecuentemente, sobre todo en los algoritmos.

2.2.2. Matriz de adyacencia

Nuevamente sea $G = (V, E)$ un grafo con $V = \{v_1, \cdots, v_n\}$. Se define la matriz de adyacencia de G como la matriz $A \in \{0, 1\}^{n \times n}$ tal que

$$A_{ij} = \begin{cases} 1 & \text{si } v_i \to v_j \in E \\ 0 & \text{si no} \end{cases}$$

La representación de G en forma de matriz de adyacencia es, obviamente, su matriz de adyacencia. Para el grafo de la Figura 1 esta representación es

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

2.3. Recorridos sobre un grafo

Dado un grafo G, podemos seleccionar uno de sus nodos, al que llamamos *nodo fuente*, y empezar a recorrerlo moviéndonos sucesivamente entre nodos adyacentes. Es evidente que hay infinitas formas distintas de recorrerlo, las cuales podemos obtener variando el orden en que visitamos los nodos o repitiendo visitas a algunos de ellos. Resulta natural pensar que, entre todas estas formas de recorrido, son preferibles aquellas que no repiten nodos, pues no realizan movimientos innecesarios. Por otro lado, también parece deseable que el recorrido visite al menos una vez cada uno de los nodos alcanzables desde el nodo fuente, porque de lo contrario sería incompleto.

Todo esto motiva la siguiente definición.

Definición 2.8. Decimos que un recorrido sobre un grafo es deseable si

1. No repite nodos.
2. Visita todos los nodos alcanzables desde el nodo fuente.

Podemos generalizar esta noción permitiendo que un recorrido tenga la capacidad de comenzar ya no desde un único nodo de partida, sino desde un *conjunto* de nodos. Recorrer el grafo no es más difícil si los nodos de partida son múltiples, pues es posible transformar el grafo que queremos recorrer agregándole un nuevo nodo que incida en todos los nodos de partida, de modo tal que el problema inicial queda reducido al de recorrer este nuevo grafo con este nuevo nodo como única fuente.

2.3.1. Esquema general

Consideremos un algoritmo que mantiene, en todo momento, un conjunto M de nodos ya visitados o *marcados*, y un conjunto $F = \{v \in V - M : \text{existe } w \in M \text{ tal que } w \to v \in E\}$, es decir, el conjunto de nodos adyacentes a nodos en M, que están fuera de M. Los nodos de F se llaman *nodos fronterizos*. En cada paso, el algoritmo elige un nodo de F, lo agrega a los nodos visitados, y actualiza el conjunto de nodos fronterizos. Repite esta operación hasta que no haya nodos fronterizos.

input : G grafo
S conjunto de nodos fuente
output: M conjunto de nodos visitados

```
1  begin
2      M = ∅
3      F = S
4      while F ≠ ∅ do
5          Elegir un nodo v ∈ F
6          Agregar v al conjunto M
7          Actualizar el conjunto de nodos fronterizos F
8      end
9      return M
10 end
```

Algorithm 1: Recorrido sobre un grafo

El Algoritmo 1 recorrerá exactamente todos los nodos alcanzables desde el conjunto input M (la demostración de ello es sencilla y se puede ver en [2]). Además no repite nodos, porque cada vez que visita un nodo lo agrega al conjunto M, que es disjunto con F. Por lo tanto, este algoritmo realiza un recorrido deseable sobre un grafo.

Notemos que el algoritmo no especifica la forma en que se realiza la elección de $v \in F$ en la línea 4. Por esta razón decimos que representa a una familia de algoritmos que ejecutan recorridos deseables sobre un grafo, donde cada miembro de esta familia se distingue por la forma de realizar la elección del nodo fronterizo.

Dos de los algoritmos que se derivan del Algoritmo 1 son DFS y BFS. A continuación presentamos una breve exposición de los mismos. Si bien daremos una definición precisa de ellos, el lector no familiarizado debería referirse a un tratamiento más detallado (por ejemplo en [1]).

2.3.2. DFS

Depth First Search (en castellano, búsqueda en profundidad) es un algoritmo de recorrido que sigue el esquema del Algoritmo 1. Específicamente elige, entre los nodos fronterizos F, el último de los nodos agregados al conjunto. Esto hace que los nodos se recorran en profundidad, es decir, intentando alejarse, en cada paso, de los nodos fuente.

Una implementación tradicional de DFS utiliza, para representar el conjunto F, una pila. Una pila resulta adecuada ya que el nodo de F elegido en cada paso es el último agregado al conjunto, lo cual coincide con el invariante LIFO (Last In First Out).

En el Algoritmo 2 presentamos esta versión de DFS con una pila. Al grafo G lo representamos mediante listas de adyacencia. Si $V = \{v_1, \cdots, v_n\}$ es el conjunto de nodos de G, entonces al conjunto S de nodos fuente lo representamos como un arreglo de n elementos, tal que

$$S[i] = \begin{cases} 1 & \text{si } v_i \in S \\ 0 & \text{si no} \end{cases}$$

El conjunto M de nodos marcados se representa igual que S.

input : $G = (V, E)$ grafo
S conjunto de nodos fuente
output: M conjunto de nodos visitados

```
1  begin
2      n = |V|
3      Sea F una pila
4      F = ∅
5      Sea M un arreglo de n elementos
6      Inicializar cada posición de M en 0
7      for i = 1 to n do
8          if S[i] = 1 then
9              M[i] = 1
10             Push(F, i)
11         end
12     end
13     while F ≠ ∅ do
14         i = Pop(F)
15         foreach j ∈ Adj[i] do
16             if M[j] = 0 then
17                 M[j] = 1
18                 Push(F, j)
19             end
20         end
21     end
22     return M
23 end
```

Algorithm 2: Iterative-DFS

La función Push agrega un elemento a la pila, y la función Pop extrae el próximo elemento de la misma. Éstas son las funciones básicas en la interfaz de una pila.

En nuestra implementación, M y F no representan conjuntos disjuntos, dado que cada vez que agregamos un nodo a F, lo marcamos en M para no volver a insertarlo en la pila. Si bien esto es inconsistente con el invariante original sobre F y M indicado en 2.3.1, es fácilmente salvable utilizando un arreglo auxiliar para evitar insertar duplicados en F, y modificar M únicamente al extraer los nodos de la pila. Hemos preferido sacrificar un poco de claridad en el algoritmo para ganar en términos temporales y espaciales al evitarnos este arreglo auxiliar.

Calculemos la complejidad del Algoritmo 2. Sea m la cantidad de aristas del grafo G. La inicialización de las variables es $\mathcal{O}(n)$. El ciclo de las líneas 7-12 es $\mathcal{O}(n)$. Falta calcular la complejidad del ciclo 13-22. Dado que sólo agregamos a la pila nodos no marcados, y que al agregar un nodo éste es marcado, entonces a lo sumo agregamos una vez cada nodo a la pila. Esto implica que la línea 14 y la línea 18 se ejecutan $\mathcal{O}(n)$ veces. Más aún, las líneas 15 y 16 se ejecutan $\mathcal{O}(m)$ veces,

porque cada ejecución de la línea 15 se realiza para un eje $i \to j \in E$ distinto. Entonces, el ciclo 13-21 cuesta $\mathcal{O}(n+m)$. En definitiva, la complejidad de DFS es $\mathcal{O}(n+m)$.

Observemos que la representación del grafo mediante listas de adyacencias tiene una fuerte influencia en la complejidad calculada, ya que gracias a esto el ciclo 15-20 se ejecuta $\mathcal{O}(m)$ veces. Si en lugar de esto utilizáramos una matriz de adyacencia, dicho ciclo se ejecutaría $\mathcal{O}(n^2)$ veces, ya que para cada uno de los $\mathcal{O}(n)$ nodos extraídos de la pila, habría que recorrer toda una fila de la matriz de adyacencia, buscando los nodos adyacentes. Por lo tanto, DFS representando al grafo con una matriz de adyacencia tiene costo $\mathcal{O}(n^2)$.

Una implementación alternativa de DFS, quizá más natural, es la recursiva, que simula el comportamiento de la pila mediante recursión. La idea del algoritmo es que si queremos recorrer un grafo en profundidad entonces al visitar un nodo debemos hacer DFS recursivamente sobre sus nodos adyacentes. En el Algoritmo 3 podemos ver esta implementación alternativa. La representación de G, S y M que utiliza es la misma que antes.

```
input  : G = (V, E) grafo
         S conjunto de nodos fuente
output: M conjunto de nodos visitados
1  begin
2      n = |V|
3      Sea M un arreglo de n elementos
4      Inicializar cada posición de M en 0
5      for i = 1 to n do
6          if S[i] = 1 then
7              M[i] = 1
8              DFS-Visit(G, M, i)
9          end
10     end
11     return M
12 end
```

Algorithm 3: Recursive-DFS

```
input  : G grafo
         M conjunto de nodos visitados
         i un nodo de G
output:-
1  begin
2      foreach j ∈ Adj[i] do
3          if M[j] = 0 then
4              M[j] = 1
5              DFS-Visit(G, M, j)
6          end
7      end
8  end
```

Algorithm 4: DFS-Visit

Vale la pena aclarar que el arreglo M debe pasarse por referencia a la función DFS-Visit de modo tal que todas las visitas modifiquen un único arreglo M.

2.3.3. BFS

Breadth First Search (en castellano, búsqueda en anchura) es otro algoritmo de recorrido que sigue el esquema del Algoritmo 1. Específicamente elige, entre los nodos fronterizos F, el primero

de los nodos agregados al conjunto. Esto hace que los nodos se recorran en un orden incremental, según la distancia a los nodos fuente.

Una implementación tradicional de BFS utiliza, para representar el conjunto F, una cola. Esta estructura resulta adecuada ya que el nodo de F elegido en cada paso es el último agregado al conjunto, lo cual coincide con el invariante FIFO (First In First Out).

En el Algoritmo 5 presentamos una implementación de BFS. La única diferencia respecto del DFS iterativo es que hemos cambiado la pila para representar F por una cola. Se puede probar, en forma análoga a lo hecho para DFS, que la complejidad de BFS, representando al grafo con listas de adyacencia, es $\mathcal{O}(n+m)$, con m la cantidad de aristas del grafo. Utilizando una matriz de adyacencia, la complejidad es $\mathcal{O}(n^2)$.

```
input  : G = (V, E) grafo
         S conjunto de nodos fuente
output: M conjunto de nodos visitados
1  begin
2      n = |V|
3      Sea F una cola
4      F = ∅
5      Sea M un arreglo de n elementos
6      Inicializar cada posición de M en 0
7      for i = 1 to n do
8          if S[i] = 1 then
9              M[i] = 1
10             Push(F, i)
11         end
12     end
13     while F ≠ ∅ do
14         i = Pop(F)
15         foreach j ∈ Adj[i] do
16             if M[j] = 0 then
17                 M[j] = 1
18                 Push(F, j)
19             end
20         end
21     end
22     return M
23 end
```

Algorithm 5: BFS

3. Vectorización de BFS

La paralelización de este algoritmo se basa fuertemente en el orden por niveles en que BFS visita los nodos. Esto es, primero visita todos los nodos que se encuentran a distancia 1 del conjunto de nodos fuente, luego visita todos los que se encuentran a distancia 2, y así sucesivamente. Formalmente, esto se expresa en que el Algoritmo 5 cumple que los nodos son extraídos de la cola en orden, según la distancia al conjunto S. Por lo tanto, podemos clasificar por niveles a los nodos visitados, según esta distancia, de modo tal que los nodos del nivel i serán aquellos que se encuentran a distancia i de los nodos fuente. Los niveles se irán visitando en orden y todo nodo visitado pertenece a algún nivel. Además, notar que si el grafo a recorrer tiene n nodos entonces a lo sumo hay n niveles, puesto que no hay dos nodos que disten más de n. Por lo tanto, si conocemos los nodos que ocupan cada uno de los n niveles, entonces conocemos el conjunto de

nodos vistados por BFS. En la Figura 2 se puede ver esta clasificación por niveles de los nodos de un grafo. Las aristas de trazo sólido son aquellas por las que se recorre el grafo, mientras que las de trazo punteado son aquellas por las que no se transita.

El esquema básico de las vectorizaciones que propondremos es el indicado en el Algoritmo 6.

input : $G = (V, E)$ grafo
S conjunto de nodos fuente
output: M conjunto de nodos visitados

```
1 begin
2 |  n = |V|
3 |  M = S
4 |  for i = 1 to n do
5 |  |  Computar en paralelo los nodos del nivel i y agregarlos a M
6 |  end
7 |  return M
8 end
```

Algorithm 6: BFS en paralelo

3.1. Algoritmo $\mathcal{O}(n^3)$

Dado el grafo G, enumeremos sus nodos de 1 a n. Supongamos que está representado mediante una matriz de adyacencia, que llamamos A_G. Para cada $k = 1, \cdots, n$, supongamos que $fil_k(A_G)$ es un arreglo de n elementos, que representa la fila k de la matriz A_G. Notemos que $fil_k(A_G)$ indica los nodos en los que incide el nodo k.

Supongamos que deseamos calcular los nodos del nivel $i \geq 1$. El resultado clave es el siguiente.

Proposición 3.1. *Los nodos del nivel i son exactamente aquellos que son incididos por algún nodo del nivel $i-1$ pero que aún no han sido visitados.*

Demostración. Sea v un nodo perteneciente al nivel i, entonces v se encuentra a distancia i del conjunto S de nodos fuente. Consideremos un camino $\langle v_0, \cdots, v_i = v \rangle$ que realice esa distancia. Dicho camino comienza en un nodo fuente y termina en v. El nodo v_{i-1} necesariamente dista $i-1$ del conjunto S, pues $d(S, v_{i-1}) \leq i-1$ y no puede distar menos que esto porque de lo contrario tendríamos $d(S, v) < i$. Por lo tanto, $d(S, v_{i-1}) = i-1$ con lo cual v_{i-1} es un nodo del nivel $i-1$ que incide en v.

Recíprocamente, si un nodo v aún no visitado es incidido por un nodo del nivel $i-1$, entonces $d(S, v) > i-1$, porque de lo contrario ya habría sido visitado en algún nivel inferior a i, y además $d(S, v) \leq i$ porque existe un camino de S a v que usa i aristas, que es el camino que pasa por el nodo incidente del nivel $i-1$. Entonces $d(S, v) = i$, con lo cual v se encuentra en el nivel i. □

Antes de usar esta proposición necesitamos introducir algunos conceptos y operaciones involucradas en el algoritmo.

Definición 3.1. Dado un arreglo $X[1 \cdots n]$, decimos que X es un arreglo binario de n elementos si para cada $k = 1, \cdots, n$, $X[k]$ vale 0 o 1.

Definición 3.2. Definimos la operación binaria AND, cerrada en los arreglos binarios, del siguiente modo. Dados X, Y arreglos binarios de n elementos, X AND Y es un arreglo binario de n elementos tal que para cada $k = 1, \cdots, n$,

$$(X \text{ AND } Y)[k] = X[k] \text{ and } Y[k]$$

donde el operador and es la conjunción lógica clásica. Análogamente se definen OR y NOT como

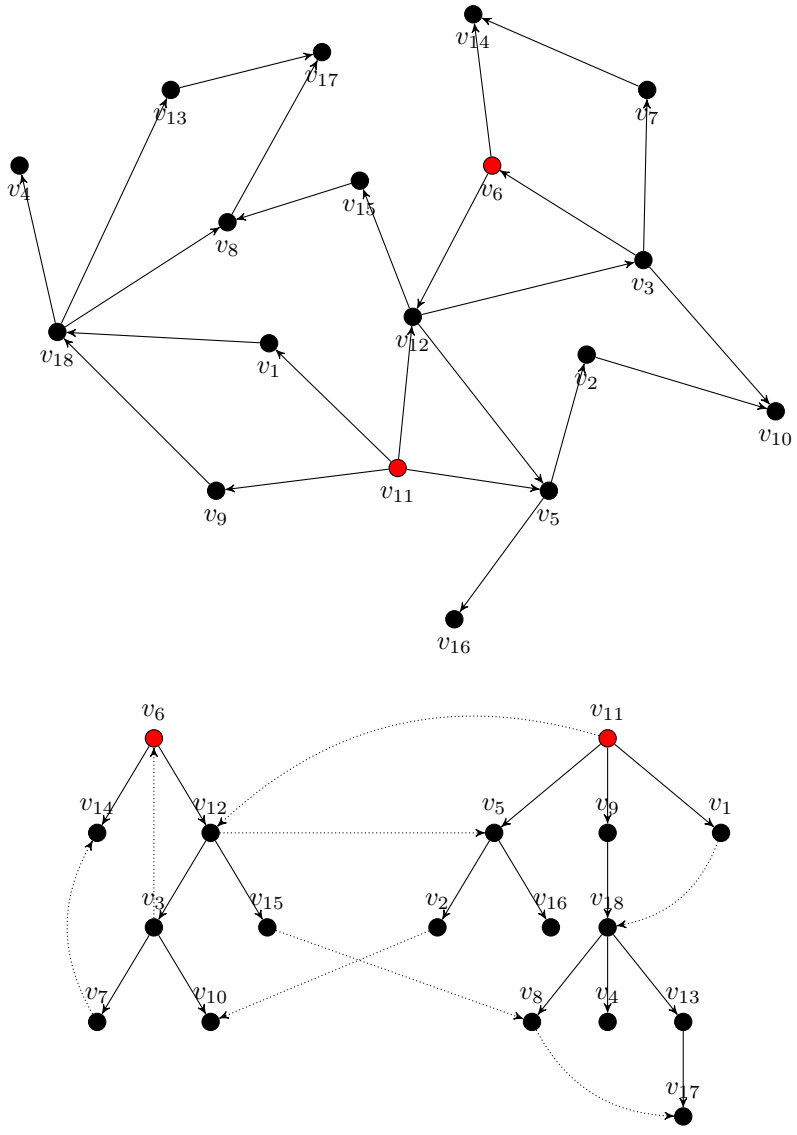

Figura 2: Clasificación de los nodos de un grafo por niveles

$$(X \text{ OR } Y)[k] = X[k] \text{ or } Y[k]$$

$$(\text{NOT } X)[k] = \text{not } X[k]$$

donde el operador or es la disyunción lógica y not es la negación lógica.

Definimos X como el arreglo de n elementos que contiene los nodos del nivel $i-1$, es decir,

$$X[k] = \begin{cases} 1 & \text{si el nodo } k \text{ pertenece al nivel } i-1 \\ 0 & \text{si no} \end{cases}$$

Llamemos M al arreglo binario que indica los elementos marcados hasta el momento. Si $X[k] = 1$ entonces los nodos del nivel i en los que incide el nodo k son exactamente $fil_k(A_G)$ AND (NOT M). Definamos un arreglo I_k que indica si el elemento $X[k]$ se encuentra en el nivel $i-1$, esto es, $I_k = \langle X[k], X[k], \cdots, X[k] \rangle$. Entonces $fil_k(A_G)$ AND (NOT M) AND I_k contiene ceros si el nodo k no se encuentra en el nivel $i-1$ y contiene los nodos del nivel i en los que incide k en caso de que k pertenezca al nivel $i-1$.

Definamos Y como el arreglo binario que indica los nodos pertenecientes al nivel i.

Proposición 3.2. $Y = OR_{1 \leq k \leq n}(fil_k(A_G)$ *AND* $(NOT\ M)$ *AND* $I_k)$.

Demostración. Se sigue de la Proposicion 3.1, que dice que los nodos del nivel i son aquellos incididos por algún nodo del nivel $i-1$ pero que aún no han sido visitados. □

Esta proposición nos da una forma de computar los nodos de cada nivel, dando lugar al Algoritmo 7.

```
input  : G = (V, E) grafo
         S conjunto de nodos fuente
output: M conjunto de nodos visitados
1  begin
2      n = |V|
3      Sean M, X, Y arreglos de n elementos
4      for i = 1 to n do
5          M[i] = S[i]
6          X[i] = S[i]
7      end
8      for i = 1 to n do
9          Setear cada posición de Y en 0
10         for k = 1 to n do
11             Sea I un arreglo de n elementos
12             Inicializar cada posición de I en X[k]
13             Y = Y OR (fil_k(A_G) AND (NOT M) AND I)
14         end
15         M = M OR Y
16         X = Y
17     end
18     return M
19 end
```

Algorithm 7: Cómputo utilizando la Proposición 3.2

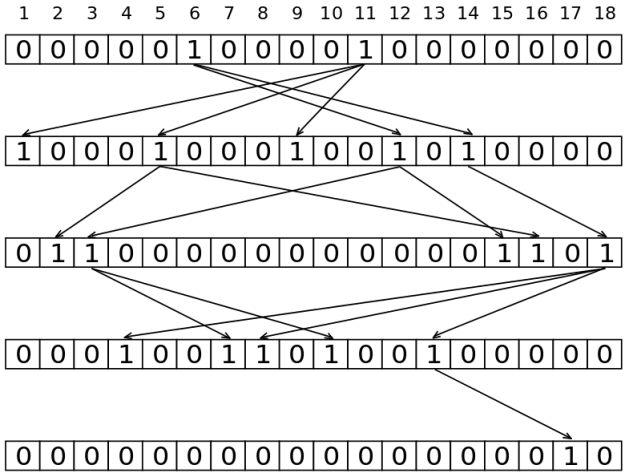

Figura 3: Niveles representados por arreglos de bits

En la Figura 3 se pueden ver cada uno de los niveles del grafo de la Figura 2 representado mediante arreglos de bits. Esto ilustra cómo es posible pensar un recorrido BFS en términos de arreglos de bits. La figura incluye sólo las aristas transitadas por la búsqueda.

Notemos que si bien en cada iteración del ciclo 10-14 computamos estrictamente los nodos del nivel i (es decir, Y nunca contiene nodos que no pertenezcan al nivel que se está calculando) podemos relajar un poco esta condición ya que, para llegar al resultado, lo único que nos necesitamos saber es si cada nodo del grafo pertenece a algún nivel, y no nos interesa saber a cuál. Esto es, en lugar de pedir que Y contenga sólo los nodos del nivel i, pedimos que contenga los nodos de los niveles $0, \cdots, i$. Pero notemos que esta condición ya es cumplida por el arreglo M a lo largo del ciclo. Esta situación es un indicio de que podemos prescindir del arreglo Y, utilizando directamente M. Lo que falta es ver cómo hacemos crecer el conjunto M a lo largo de las iteraciones. En otras palabras, teniendo todos los nodos de los niveles $0, \cdots, i-1$, ¿cómo agregamos los nodos del nivel i?. La respuesta es sencilla y está sintetizada en la siguiente proposición.

Proposición 3.3. *Sea M el conjunto de nodos de los niveles $1, \cdots, i-1$. Entonces los nodos del nivel i son exactamente aquellos que no están en M y que son incididos por algún nodo de M.*

Demostración. Si v es un nodo del nivel i, entonces no pertenece a M por definición, y además es incidido por algún nodo de M por la Proposición 3.1.

Recíprocamente, sea v un nodo tal que $v \notin M$ y es incidido por algún nodo de M. Como $v \notin M$ entonces se encuentra en un nivel mayor o igual que i, es decir, $d(S, v) \geq i$. Como es incidido por algún nodo de M entonces existe un camino desde un nodo fuente hacia v de longitud i o menos, con lo cual $d(S, v) \leq i$. Entonces $d(S, v) = i$ y por ende v se encuentra en el nivel i. □

Por lo tanto, para agregar a M los nodos del nivel i basta agregar los nodos incididos por aquellos contenidos en M, pero que no estén en M. Dado que un conjunto no contiene elementos repetidos, agregar elementos que ya estén en M no hace daño, con lo cual podemos agregar nodos a M sin preocuparnos por la condición de la no pertenencia.

Pero podemos ir todavía más allá y admitir que M tenga, además de todos los nodos de los niveles $0, \cdots, i$, algunos nodos de niveles superiores a i. Esto queda plasmado en el Algoritmo 8 que, a lo largo del ciclo 8-12, modifica el arreglo M al mismo tiempo que lo utiliza para calcular nuevos valores, pues el arreglo I se construye a partir de elementos de M que posiblemente hayan sido modificados en la ejecución actual del ciclo 7-13.

Para probar que el algoritmo sigue siendo correcto, veamos que el siguiente invariante se mantiene:

input : $G = (V, E)$ grafo
S conjunto de nodos fuente
output: M conjunto de nodos visitados

```
1  begin
2  |  n = |V|
3  |  Sea M un arreglo de n elementos
4  |  for i = 1 to n do
5  |  |  M[i] = S[i]
6  |  end
7  |  for i = 1 to n do
8  |  |  for k = 1 to n do
9  |  |  |  Sea I un arreglo de n elementos
10 |  |  |  Inicializar cada posición de I en M[k]
11 |  |  |  M = M OR (fil_k(A_G) AND I)
12 |  |  end
13 |  end
14 |  return M
15 end
```

Algorithm 8: VECTORIZED-BFS-NO-BRANCHING

Al principio de cada iteración del ciclo 8-12, el arreglo M sólo contiene (como conjunto) elementos visitados por BFS y, particularmente, contiene todos los nodos de los niveles $0, \cdots, i-1$.

Debemos ver que el invariante es cierto al ingresar al ciclo y que se mantiene de una iteración a la siguiente.

- **Inicialización.** Al ingresar al ciclo, M contiene únicamente nodos fuente, que son visitados por BFS. En este punto vale $i = 1$, y M contiene todos los nodos del nivel 0. Entonces vale el invariante.
- **Mantenimiento.** Supongamos que vale el invariante al principio de una iteración. Entonces M sólo contiene elementos visitados por BFS y, en particular, todos los nodos de los niveles $0, \cdots, i-1$. Por la línea 11, sólo se agregan a M elementos incididos por nodos de ese mismo conjunto. Entonces todos los nodos agregados a M serán alcanzables desde los nodos que contenía M al ingresar al ciclo 8-12. Además, al ingresar al ciclo 8-12, M sólo contiene, por la validez del invariante, elementos visitados por BFS. Luego, el ciclo 8-12 sólo agrega a M elementos alcanzables desde nodos visitados por BFS, y por ende los nodos agregados también son visitados por BFS.

  Veamos ahora que al concluir el ciclo 8-12, M contiene todos los nodos de los niveles 0 a i. Es claro que contendrá todos los nodos de los niveles $0, \cdots, i-1$, por la validez del invariante y porque el ciclo 8-12 no quita elementos de M. Además agrega todos los nodos del nivel i, porque k itera sobre los nodos del nivel $i-1$ (entre muchos otros) y agrega todos los nodos en los que inciden. Por la Proposición 3.1, el ciclo 8-12 agrega, en particular, todos los nodos del nivel i.

  Al comenzar una nueva iteración del ciclo 7-13 se incrementa el valor de i y se reestablece el invariante.
- **Finalización.** Al finalizar el ciclo vale que $i = n+1$ y por lo tanto se tiene que M contiene sólo nodos visitados por BFS y todo nodo perteneciente a un nivel entre 0 y n está en M. Esto es, M contiene exactamente los nodos que visita BFS.

Entonces VECTORIZED-BFS-NO-BRANCHING es correcto. Observemos que la principal ventaja que tiene frente al Algoritmo 7 es su menor requerimiento espacial ya que no necesita de los arreglos

auxiliares X e Y. Más aún, si realizamos los cómputos directamente sobre el arreglo de entrada S podemos hacer que el algoritmo sea in-place.

El Algoritmo 8 es una versión vectorizada de BFS ya que las operaciones AND, OR y NOT se pueden realizar en paralelo utilizando el modelo SIMD. Al emplear esta técnica de paralelización se busca exprimir al máximo la performance del procesador por lo que se evita la introducción de instrucciones de decisión, que perjudican la capacidad de branching prediction. El nombre de VECTORIZED-BFS-NO-BRANCHING se debe, justamente, a que nunca utiliza una estructura de decisión.

3.1.1. Complejidad

La complejidad del Algoritmo 8 es claramente $\mathcal{O}(n^3)$ pues el costo está dominado por las líneas 7-13, conformadas por dos ciclos anidados más las operaciones lógicas a nivel de arreglos, que son $\mathcal{O}(n)$.

3.1.2. Implementación

Implementamos el Algoritmo 8 en assembler de un procesador Intel® Core™ i5-3470, utilizando la tecnología SSE de Intel para la vectorización. Dicha implementación es escencialmente igual al algoritmo exhibido, ya que las operaciones AND y OR son computadas por las funciones `PAND` y `POR` brindadas por SSE. La diferencia yace en la forma en que realizamos tales operaciones lógicas sobre arreglos. Mientras que en el algoritmo estamos considerando que dichas operaciones se realizan elemento a elemento, la implementación las ejecuta en forma vectorizada. Más precisamente, para realizar la instrucción de la línea 11 utilizamos registros XMM de 16B, levantando en ellos bloques consecutivos de los arreglos M, $fil_k(A_G)$ e I.

Dado que todos los arreglos son binarios, podemos dotarlos de una representación compacta en forma de arreglos de bits (bits dispuestos en forma contigua en memoria). Esta opción resulta adecuada, además, debido al tipo de operaciones SSE que realizamos. En general, dado que las funciones vectorizadas operan sobre bloques de bytes de un registro, no es posible procesar en paralelo bloques con una granularidad más pequeña que 1B. La particularidad de las funciones `PAND` y `POR` es que su efecto es el mismo en cualquier bloque de cualquier tamaño del registro involucrado, pues operan sobre cada uno de los bits independientemente.

Bajo esta representación podemos paralelizar a nivel de bits y por lo tanto operar de a $16 \times 8 = 128$ elementos de los arreglos en simultaneo. Teniendo en cuenta esto, hemos requerido que el arreglo S de entrada, así como cada fila de la matriz de adyacencia de G, tengan una longitud múltiplo de 16B. Pese a que es posible relajar esta precondición (podemos pedir solamente que la longitud sea mayor o igual a 16B), optamos por mantenerla ya que hace más sencillo el código y no agrega costo espacial ni temporal. El único requerimiento adicional sobre el input es el almacenamiento en memoria por filas de la matriz de adyacencia A_G, debido a que cada fila de dicha matriz es leída de a bloques consecutivos.

3.2. Algoritmo $\mathcal{O}(n^2)$

El Algoritmo 7 desperdicia mucho tiempo ejecutando iteraciones del ciclo 10-14, para los nodos k que no se encuentran en el nivel $i-1$ (aquellos para los que $X[k] = 0$). En esos casos, el arreglo I no contiene más que ceros y no se modifica el contenido del arreglo Y. Vamos a sacrificar un poco de capacidad de predicción de saltos introduciendo una guarda que verifique esto.

El Algoritmo 9 introduce esta mínima pero importante modificación. Veamos que esto reduce la complejidad temporal de la función a $\mathcal{O}(n^2)$.

3.2.1. Complejidad

Para cada $i = 0, \cdots, n$, llamemos x_i a la cantidad de nodos en el nivel i. El costo del algoritmo está dominado por la ejecución de las líneas 8-17. La línea 8 es $\mathcal{O}(1)$ y se ejecuta $\mathcal{O}(n)$ veces. La línea 9 es $\mathcal{O}(n)$ y se ejecuta $\mathcal{O}(n)$ veces. La línea 10 es $\mathcal{O}(1)$ y se ejecuta $\mathcal{O}(n^2)$ veces. Las líneas 15

```
input  : G = (V, E) grafo
         S conjunto de nodos fuente
output: M conjunto de nodos visitados
1  begin
2      n = |V|
3      Sean M, X, Y arreglos de n elementos
4      for i = 1 to n do
5          M[i] = S[i]
6          X[i] = S[i]
7      end
8      for i = 1 to n do
9          Setear cada posición de Y en 0
10         for k = 1 to n do
11             if X[k] = 1 then
12                 Y = Y OR (fil_k(A_G) AND (NOT M))
13             end
14         end
15         M = M OR Y
16         X = Y
17     end
18     return M
19 end
```

Algorithm 9: VECTORIZED-BFS-BRANCHING

y 16 son $\mathcal{O}(n)$ y se ejecutan $\mathcal{O}(n)$ veces. Las líneas 11-13 cuestan $\mathcal{O}(n)$ y se ejecutan $\mathcal{O}\left(\sum_{i=1}^{n} x_i\right)$ veces. Entonces el costo total del algoritmo es $\mathcal{O}\left(n^2 + n\sum_{i=1}^{n} x_i\right)$. Pero $x_1 + \cdots + x_n \leq n$, pues los nodos no se repiten a lo largo de los niveles. Entonces $\sum_{i=1}^{n} x_i = \mathcal{O}(n)$ y el costo del algoritmo resulta ser $\mathcal{O}(n^2)$.

3.2.2. Implementación

Al igual que el algoritmo anterior, la implementación resulta sencilla a partir del algoritmo. Lo único nuevo aquí es la operación NOT, aunque su naturaleza es la misma que la de AND y OR, y por lo tanto es igualmente fácil de traducir a instrucciones SSE.

3.3. Experimentación

Hemos implementado los algoritmos BFS, VECTORIZED-BFS-NO-BRANCHING y VECTORIZED-BFS-BRANCHING, y se midieron los tiempos de ejecución de todos ellos, para distintos tipos de entradas.

Se esperaba que VECTORIZED-BFS-NO-BRANCHING tuviera, en cualquier escenario, la peor performance de todas, dada su complejidad $\mathcal{O}(n^3)$. La duda estaba entre BFS, de costo $\mathcal{O}(n+m)$, y VECTORIZED-BFS-BRANCHING, de costo $\mathcal{O}(n^2)$. Notemos que si bien la versión vectorizada sólo depende de la cantidad de nodos del grafo, BFS depende de la cantidad de aristas m que cuanto más denso es el grafo más se aproxima a $n(n-1) = \mathcal{O}(n^2)$. Por este motivo, esperábamos que BFS tuviera un mejor desempeño para grafos ralos, pero que al incrementarse la densidad su tiempo de ejecución se equiparara al de VECTORIZED-BFS-BRANCHING.

En los siguientes gráficos se muestran los tiempos registrados para estos dos algoritmos. Estos tiempos se midieron utilizando el registro TSC (Time Stamp Counter) del procesador. Si bien también hemos puesto a prueba la implementación de la vectorización sin branching, no los incluimos aquí ya que sus tiempos son un orden de magnitud mayor a los de los otros dos algoritmos. Hemos medido tiempos para grafos de distintos tamaños. Además, hemos variado la densidad

de los grafos, utilizando un parámetro de densidad $p \in [0, 1]$. Este valor p es la probabilidad de que, elegidos (en orden) dos nodos, el primero incida en el segundo. Finalmente, el tamaño del conjunto de nodos fuente tuvo, en todos los casos, cardinalidad aproximadamente $n/100$, y tales nodos fueron tomados al azar.

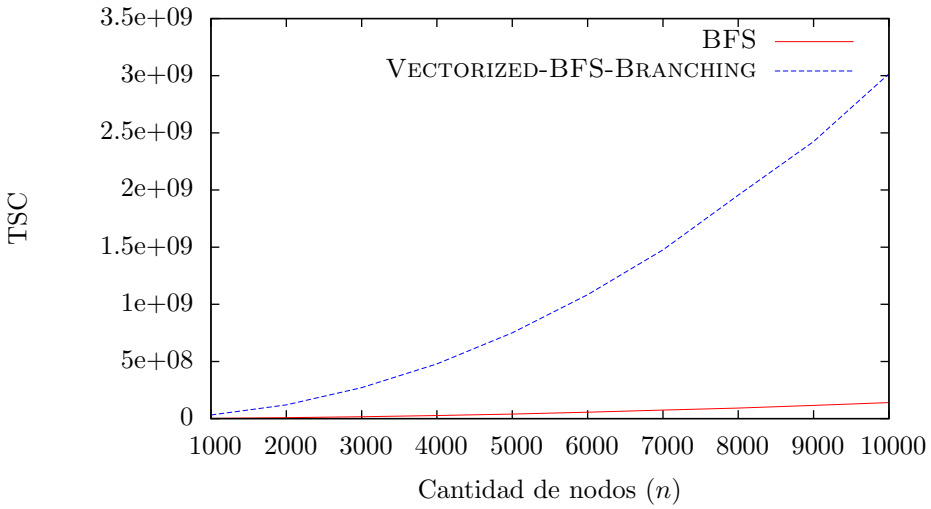

Figura 4: Densidad $p = 0{,}1$

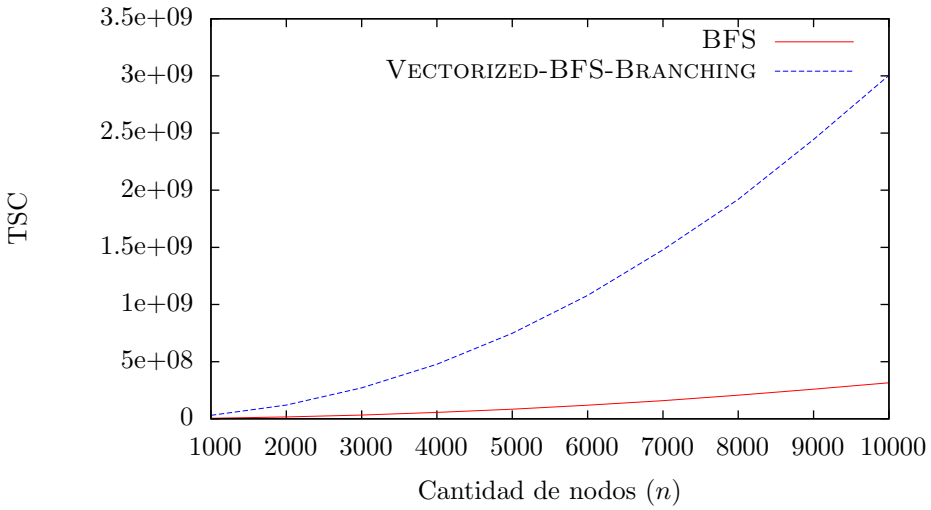

Figura 5: Densidad $p = 0{,}25$

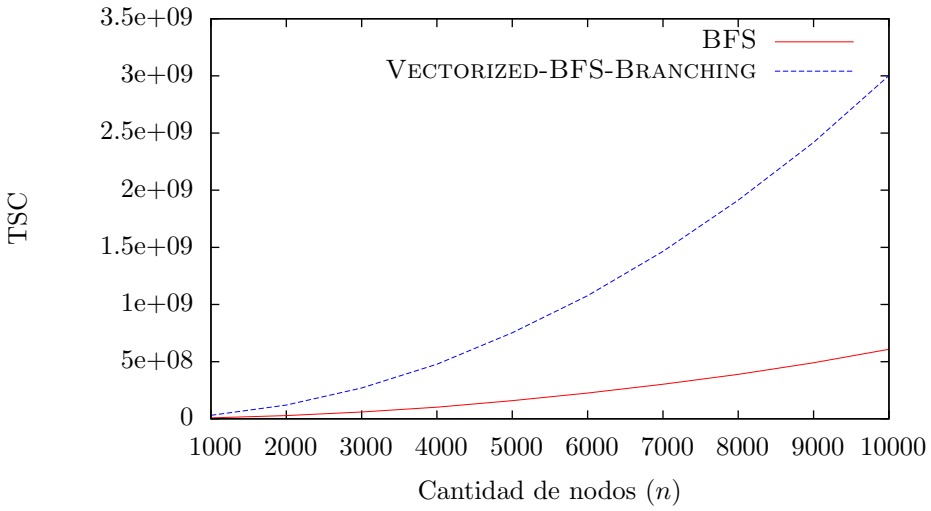

Figura 6: Densidad $p = 0{,}5$

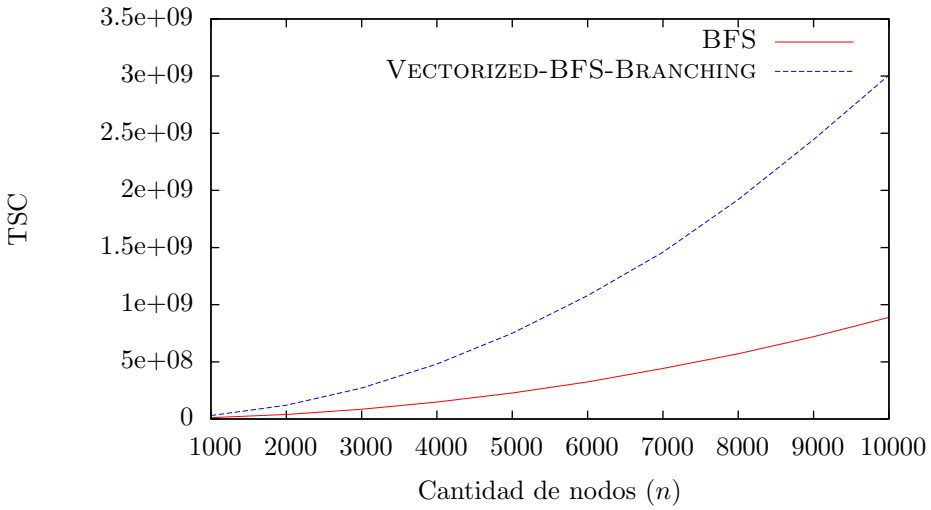

Figura 7: Densidad $p = 0{,}75$

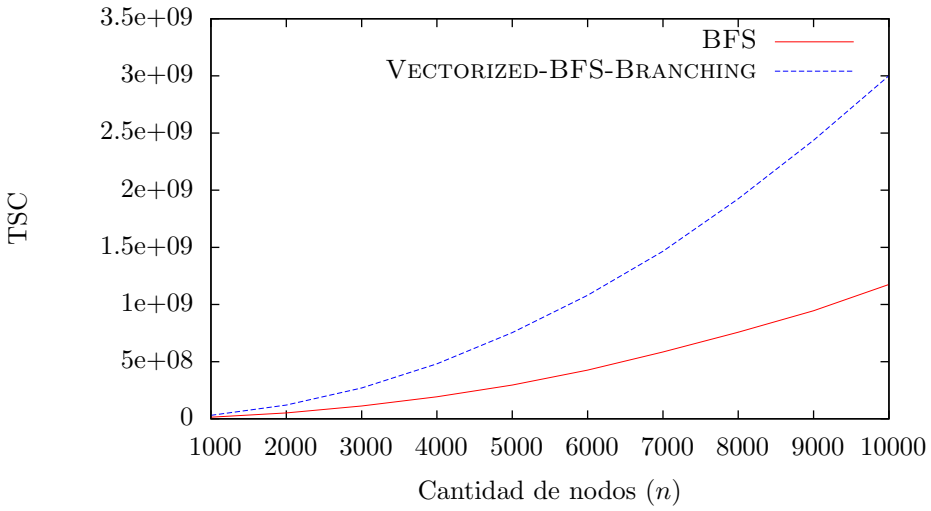

Figura 8: Densidad $p = 1$

Los resultados nos sorprendieron pues aún en los casos en los que el grafo era denso ($p \in \{0{,}75; 1\}$) BFS era más del doble de rápido que VECTORIEZD-BFS-BRANCHING. Puede observarse en el caso $p = 1$, que el comportamiento asintótico es el mismo, difirierdo los costos de ambas versiones en una constante. Para ver de dónde proviene esta constante, revisemos el Algoritmo 9. Las líneas 9, 15 y 16 tienen un costo $\Theta(n)$ y se ejecutan n veces cada una, con lo cual cada una aporta un costo $\Theta(n^2)$ en total. Más aún, cada uno de estos factores cuadráticos tienen asociada una constante que no es pequeña, pues las operaciones involucradas son accesos a memoria principal. Recordemos que estas operaciones eran realizadas dada la necesidad de mantener separadamente los nodos calculados del nivel anterior, los nodos del nivel actual, y todos los nodos marcados. Contrariamente, BFS no incurre en estos costos, pues sólo necesita mantener un único conjunto de nodos marcados.

4. Conclusión

Si bien la vectorización presentada no es una propuesta superadora de la implementación de BFS tradicional, resulta interesante desde el punto de vista teórico.

Como trabajo futuro podría estudiarse una adaptación de las vectorizaciones presentadas a algoritmos multi-núcleo. Esto permitiría distribuir entre varios cores el costo del overhead mencionado, y, al incrementar la cantidad de núcleos, eventualmente obtener tiempos de ejecución aceptables.

Referencias

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, 2001.

[2] Jonathan L. Gross and Jay Yellen. *Graph Theory and Its Applications.* Chapman Hall, 2006.