Estructuras de datos elementales

Capítulo 3 — Análisis de Algoritmos con Julia

Eric S. Téllez

Tabla de contenidos

1. Estructuras de datos elementales

1.1. Introducción

En esta unidad se discutirán las propiedades y operaciones básicas de estructuras como conjuntos, listas, pilas, colas, arreglos, vectores, matrices y matrices dispersas. Los ejemplos de código se muestran en el lenguaje de programación Julia, pero pueden ser traducidos fácilmente a otros lenguajes de programación.

1.2. Conjuntos

Los *conjuntos* son estructuras abstractas que representan una colección de elementos, pueden tener contenido inmutable o mutable. El conjunto puede contener elementos o estar vacío ($\emptyset$). Estaremos representando los conjuntos con llaves, i.e., $A = \{a, b, c\}$; el tamaño de una colección lo representamos con barras, e.g., $|A| = 3$. También es útil consultar por membresía $a \in \{a, b, c\}$ o por su negación, i.e., $d \notin \{a, b, c\}$. La operación *unión* $\cup$ construye un nuevo conjunto a partir de otros $\{a, b\} \cup \{c\} = \{a, b, c\}$; la intersección se indica con el operador $\cap$, e.g. $\{a, b, c\} \cap \{b, d\} = \{b\}$. Es común usar conjuntos mutables en diferentes algoritmos, esto es, que permitan inserciones y borrados sobre la misma estructura; desde el punto de vista de eficiencia, esto puede reducir las operaciones de manipulación de datos así como de la gestión de memoria. Suponga el conjunto $S = \{a, b, c\}$, la función $pop!(S, b)$ resultaría en $\{a, c\}$, y la función $push!(S, d)$ resultaría en $\{a, c, d\}$ al encadenar estas operaciones. Note que el símbolo ! solo se está usando en concordancia con el lenguaje de programación Julia para indicar que la función cambiaría el argumento de entrada; es una convención, no un operador en sí mismo. Note que estamos usando una sintaxis muy sencilla $fun(arg1, arg2, ...)$ para indicar la aplicación de una función u operación a una serie de argumentos.

Hay múltiples formas de representar conjuntos ya que los requerimientos de los algoritmos son diversos y tener la representación correcta puede ser una diferencia importante en el rendimiento. Las implementaciones y algoritmos alrededor pueden llegar a ser muy sofisticados, dependiendo de las características que se desean, algunas de las cuales serán el centro de estudio de este curso.

1.2.1. Ejercicios

1. Implemente un conjunto mutable usando un `Vector` sin elementos repetidos, con las operaciones de membresía, inserción, borrado, unión e intersección. Compare sus resultados (y, si le interesa adelantarse, sus costos) contra el tipo `Set` nativo de Julia.

1.3. Tuplas y estructuras

Las *tuplas* son colecciones abstractas ordenadas, donde incluso puede haber repetición, pueden verse como una secuencia de elementos, e.g., $S = (a, b, c)$; podemos referirnos a la i-ésima posición de la forma S_i, o incluso $S[i]$, si el contexto lo amerita, e.g., pseudo-código que pueda ser transferido a un lenguaje de programación más fácilmente. Es común que cada parte de la tupla pueda contener cierto tipo de dato, e.g., enteros, números de punto flotante, símbolos, cadenas de caracteres, etc. Una tupla es muy amena para ser representada de manera contigua en memoria. En el lenguaje de programación Julia, las tuplas se representan entre paréntesis, e.g., $(1, 2, 3)$.

```
t = (10, 20, 30)
```

```
t[1] * t[3] - t[2]
```

```
280
```

Definición y acceso a los campos de una tupla en Julia

1

Una *estructura* es una tupla con campos nombrados; es muy utilizada en lenguajes de programación, por ejemplo, en Julia la siguiente estructura puede representar un punto en un plano:

```
struct Point
  x::Float32
  y::Float32
end
```

Note la especificación de los tipos de datos que en conjunto describirán cómo dicha estructura se maneja por una computadora, y que en términos prácticos, es determinante para el desempeño. Es común asignar valores satelitales en programas o algoritmos, de tal forma que un elemento simple sea manipulado o utilizado de manera explícita en los algoritmos y tener asociados elementos secundarios que se vean afectados por las operaciones. Los conjuntos, tuplas y las estructuras son excelentes formas de representar datos complejos de una manera sencilla.

En Julia, es posible definir funciones o métodos alrededor del tipo de tuplas y estructuras.

2

```
"""
  Calcula la norma de un vector representado
  como una tupla
"""
function norm(u::Tuple)
  s = 0f0
  for i in eachindex(u)
    s += u[i]^2
  end
  sqrt(s)
end

"""
  Calcula la norma de un vector de 2 dimensiones
  representado como una estructura
"""
function norm(u::Point)
```

[1] En algunos lenguajes de programación como Julia, una tupla puede enviarse como *valor* (copiar) cuando se utiliza en una función; por lo mismo, puede guardarse en el *stack*, que es la memoria *inmediata* que se tiene en el contexto de ejecución de una función. En esos casos, se puede optimizar el manejo de memoria (alojar y liberar), lo cual puede ser muy beneficioso para un algoritmo en la práctica. El otro esquema posible es el *heap*, que es una zona de memoria que debe gestionarse (memoria dinámica); es más flexible y *duradera* entre diferentes llamadas de funciones en un programa. Los patrones esperados son dispersos y pueden generar fragmentación.

[2] Es importante saber que si algunos de los campos o datos de una tupla o estructura están en el *heap* entonces solo una parte estará en el *stack*; i.e., en el caso extremo solo serán referencias a datos en el *heap*. Esto puede llegar a complicar el manejo de memoria, pero también puede ser un comportamiento sobre el que se puede razonar y construir.

```
  sqrt(u.x^2 + u.y^2)
end

(norm((1, 1, 1, 1)), norm(Point(1, 1)))
```

```
(2.0f0, 1.4142135f0)
```

Funciones sobre diferentes tipos de datos

Note que la función es diferente para cada tipo de entrada; a este comportamiento se le llama despacho múltiple y será un concepto común en este curso. En otros lenguajes de programación se implementa mediante orientación a objetos.

1.3.1. Ejercicios

1. Usando `@allocated`, `sizeof` e `isbits`, verifique empíricamente si una tupla y una estructura (como `Point`) definidas en esta sección se alojan en el *stack* o requieren el *heap*. ¿Cambia el resultado si alguno de los campos fuera de tamaño variable (e.g., un `Vector` como campo)?

1.4. Arreglos

Los *arreglos* son estructuras de datos que mantienen información de un solo tipo, tienen un costo constante $O(1)$ para acceder a cualquier elemento (también llamado acceso aleatorio) y típicamente se implementan como memoria contigua en una computadora. Al igual que las tuplas, son colecciones ordenadas, las estaremos accediendo a sus elementos con la misma notación. En este curso usaremos arreglos como colecciones representadas en segmentos contiguos de memoria con dimensiones lógicas fijas. A diferencia de las tuplas, es posible reemplazar valores, entonces $S_{ij} \leftarrow a$ (`S[i, j] = a`), reemplazará el contenido de S en la celda especificada por a.

[3]

A diferencia de las tuplas, pueden tener más que una dimensión. La notación para acceder a los elementos se extiende, e.g. para una matriz S (arreglo bidimensional) S_{ij} se refiere a la celda en la fila i columna j, lo mismo que $S[i, j]$. Si pensamos en datos numéricos, un arreglo unidimensional es útil para modelar un *vector* de múltiples dimensiones, un arreglo bidimensional para representar una *matriz* de tamaño $m \times n$, y arreglos de dimensión mayor pueden usarse para tensores. Se representan en memoria en segmentos contiguos, y los arreglos de múltiples dimensiones se representarán de la misma forma, delimitando sus partes mediante aritmética simple, e.g., una matriz de tamaño $m \times n$ necesitará una zona de memoria de $m \times n$ elementos, y se puede acceder a la primera columna en la zona $1, \ldots, m$, la segunda columna en $m + 1, \ldots, 2m$, y la i-ésima en $(i - 1)m + 1, \ldots, im$; esto es,

[3] Julia tiene un soporte para arreglos excepcional, el cual apenas trataremos ya que se enfoca en diferentes áreas del cómputo numérico, y nuestro curso está orientado a algoritmos. En Python, estructuras similares se encuentran en el paquete *Numeric Python* o *numpy*; tenga en cuenta que las afirmaciones sobre el manejo de memoria y representación que estaremos usando se apegan a estos modelos, y no a las *listas* nativas de Python.

se implementa como el acceso en lotes de tamaño fijo en un gran arreglo unidimensional que es la memoria.

[4]

memoria RAM																	
otros datos	columna 1 - x[:, 1]				columna 2 - x[:, 2]				columna 3 - x[:, 3]				columna 4 - x[:, 4]				otros datos
	x[1,1]	x[2,1]	x[3,1]	x[4,1]	x[1,2]	x[2,2]	x[3,2]	x[4,2]	x[1,3]	x[2,3]	x[3,3]	x[4,3]	x[1,4]	x[2,4]	x[3,4]	x[4,4]	

Figura 1: Esquema de una matriz en memoria.

La representación precisa en memoria es significativa en el desempeño de operaciones matriciales como pueden ser el producto entre matrices o la inversión de las mismas. La manera como se acceden los datos es crucial en el diseño de los algoritmos.

El siguiente ejemplo define un vector u de m elementos y una matriz X de tamaño $m \times n$, ambos en un cubo unitario de 4 dimensiones, y define una función que selecciona el producto punto máximo del vector u a los vectores columna de X:

```
function mydot(u, x)
  s = 0f0
  for i in eachindex(u, x)
    s += u[i] * x[i]
  end
  s
end

function getmaxdot(u::Vector, X::Matrix)
  maxpos = 1
  # en la siguiente linea, @view nos permite controlar que
  # no se copien los arreglos, y en su lugar, se usen referencias
  maxdot = mydot(u, @view X[:, 1])
  # obtiene el número de columnas e itera a partir del 2do índice
  mfilas, ncols = size(X)
  for i in 2:ncols
    d = mydot(u, @view X[:, i])
    if d > maxdot
      maxpos = i
      maxdot = d
    end
  end

  (maxpos, maxdot)
end
```

[4] Esta es la manera que en general se manejan los datos en una computadora, y conocerlo de manera explícita nos permite tomar decisiones de diseño e implementación.

```
getmaxdot(rand(Float32, 4), rand(Float32, 4, 1000))
```

```
(19, 2.1391816f0)
```

Figura 2: Algoritmos para encontrar el vector columna con mayor producto en X punto contra u.

En este código se separa el cálculo del producto punto en una función, dando la idea de que es una operación importante; también podemos aislar de esta forma la manera que se accede (el orden) a los vectores. La idea fue acceder columna a columna, lo cual asegura el uso apropiado de los accesos a memoria. En la función `getmaxdot` se resuelve el problema de encontrar el máximo de un arreglo, y se puede observar que sin conocimiento adicional, este requiere $O(n)$ comparaciones, para una matriz de n columnas. Esto implica que cada producto punto se cuenta como $O(1)$, lo cual simplifica el razonamiento. Por la función `mydot` podemos observar que el producto punto tiene un costo de $O(m)$, por lo que la `getmaxdot` tiene un costo de $O(mn)$ operaciones lógicas y aritméticas.

El producto entre matrices es un caso paradigmático por su uso en la resolución de problemas prácticos, donde hay una gran cantidad de trabajo alrededor de los costos necesarios para llevarlo a cabo. En particular, el algoritmo naïve, es un algoritmo con costo cúbico, como se puede ver a continuación:

```
function myprod(A::Matrix, B::Matrix)
  mA, nA = size(A)
  mB, nB = size(B)
  @assert nA == mB
  C = Matrix{Float32}(undef, mA, nB)

  for i in 1:mA
    for j in 1:mB
      rowA = @view A[i, :]
      colB = @view B[:, i]
      C[i, j] = mydot(rowA, colB)
    end
  end

  C
end

A = rand(Float32, 5, 3)
B = rand(Float32, 3, 5)
C = myprod(A, B)
display(C)
```

```
5×5 Matrix{Float32}:
 1.042     1.042     1.042     0.0  0.0
 0.392268  0.392268  0.392268  0.0  0.0
```

```
 1.61397   1.61397   1.61397   0.0  0.0
 0.945663  0.945663  0.945663  0.0  0.0
 0.90737   0.90737   0.90737   0.0  0.0
```

Funciones sobre diferentes tipos de datos

Se pueden ver dos ciclos iterando a lo largo de filas y columnas, adicionalmente un producto punto, el cual tiene un costo lineal en la dimensión del vector, por lo que el costo es cúbico. Esta implementación es directa con la definición misma del producto matricial. Existen diferentes algoritmos para hacer esta operación más eficiente para diferentes casos o características de las matrices, siendo un área de investigación activa.

1.4.1. Matrices dispersas

Cuando una matriz tiene una gran proporción de elementos iguales a cero, almacenar explícitamente cada entrada desperdicia memoria y tiempo de cómputo. Las *matrices dispersas* solo registran los elementos distintos de cero, aprovechando la dispersión. La representación más simple es la *lista de coordenadas* (*coordinate list*, COO), que guarda tres arreglos paralelos: las filas, las columnas y los valores de las entradas no nulas.

```
using SparseArrays

filas = [1, 2, 2, 3]
columnas = [1, 1, 3, 3]
valores = Float32[4.0, 1.0, 2.0, 5.0]

S = sparse(filas, columnas, valores)
```

```
3×3 SparseMatrixCSC{Float32, Int64} with 4 stored entries:
 4.0
 1.0       2.0
          5.0
```

Construcción de una matriz dispersa a partir de su representación en coordenadas (COO)

Julia representa internamente sus matrices dispersas en el formato *compressed sparse column* (CSC), donde se registran los índices de fila y los valores no nulos agrupados por columna, junto con un arreglo de punteros que indica dónde inicia cada columna en memoria; esto favorece los accesos por columna, congruente con el resto del lenguaje. Las representaciones dispersas solo son convenientes cuando la proporción de elementos no nulos es pequeña; de otra forma, el costo adicional de indexar (en vez de calcular la posición directamente como en un arreglo denso) puede no compensar el ahorro de memoria.

1.4.2. Ejercicios

1. Modifique `myprod` para explorar diferentes órdenes de los ciclos anidados (i-k-j, k-i-j, k-j-i, etc.) y mida con `@benchmark` el efecto en el tiempo de ejecución para matrices de tamaño creciente. Explique sus resultados en términos de localidad de referencia y el orden en que Julia almacena los arreglos en memoria (*column-major*).

1.5. Listas

Las *listas* son estructuras de datos ordenadas lineales, esto es, no se asume que los elementos se guardan de manera contigua y los accesos al i-ésimo elemento cuestan $O(i)$. Se soportan inserciones y borrados. Por ejemplo, sea $L = [a, b, c, d]$ una lista con cuatro elementos, $L_2 = b$, $insert!(L, 2, z)$ convertirá $L = [a, z, b, c, d]$ (note que b se desplazó y no se reemplazó como se esperaría en un arreglo). La operación $deleteat!(L, 2)$ regresará la lista a su valor previo a la inserción. Estas operaciones que modifican la lista también tienen diferentes costos dependiendo de la posición, e.g., donde el inicio y final de la secuencia (también llamados *head/cabeza* y *tail/cola*) suelen ser más eficientes que accesos aleatorios, ya que se tienen referencias a estas posiciones en memoria. Es de especial importancia la navegación por la lista mediante operaciones de sucesor *succ* y predecesor *pred*, que pueden encadenarse para obtener acceso a los elementos. A diferencia de un arreglo, las listas no requieren una notación simple para acceso aleatorio a los elementos; los accesos típicos son a los extremos de la lista (cabeza y cola), sucesor y predecesor.

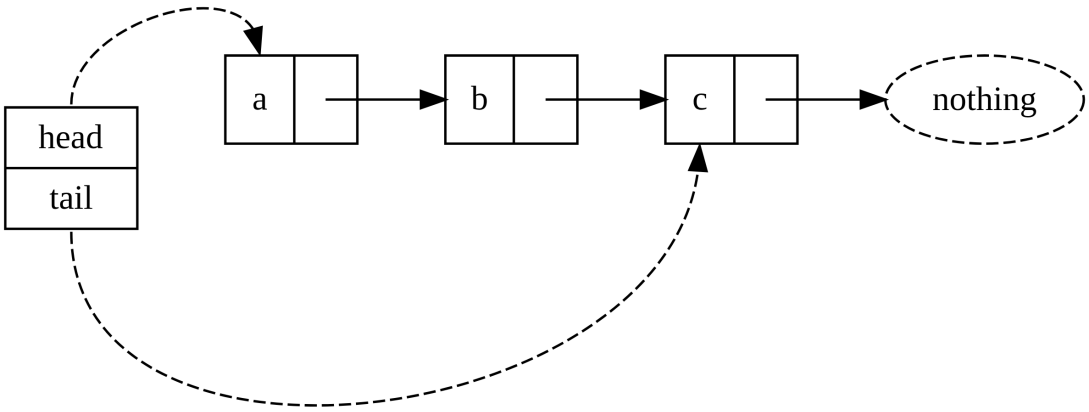

Figura 3: Una lista ligada simple

La Figura 3 muestra una lista ligada, que es una implementación de lista que puede crecer fácilmente, funciona en el *heap* de memoria por lo que cada bloque requiere memoria dinámica. Cada bloque es una estructura; se pueden distinguir dos tipos: la lista, que contiene referencias al primer nodo y al último nodo, y los *nodos de datos*, que contienen los elementos de la colección y referencias al

siguiente nodo, también llamado *sucesor*. El nodo *nothing* es especial y significa que no hay más elementos.

El siguiente código muestra la definición de una lista ligada.

Listado 1 Código para una lista ligada simple

```
struct Node
  data::Int
  next::Union{Node,Nothing}
end

node = Node(10, Node(20, Node(30, nothing)))

println(node)
(node.data, node.next.data, node.next.next.data)
```

```
Node(10, Node(20, Node(30, nothing)))
```

```
(10, 20, 30)
```

En el Listado 1 se ignora la referencia a *tail* (*head* se guarda en *node*), por lo que las operaciones sobre *tail* requieren recorrer la lista completa, costando $O(n)$ en el peor caso para una lista de n elementos.

Por la manera en que son accedidos los datos, se tienen dos tipos de listas muy útiles: las *colas* y las *pilas*. Las *colas* son listas que se acceden solo por sus extremos, y emulan la política de *el primero en entrar es el primero en salir* (first in - first out, FIFO), y es por eso que se les llama colas haciendo referencia a una cola para realizar un trámite o recibir un servicio. Las *pilas* o *stack* son listas con la política *el último en entrar es el primero en salir* (last in - first out, LIFO). Mientras que cualquier lista puede ser útil para implementarlas, algunas maneras serán mejores que otras dependiendo de los requerimientos de los problemas siendo resueltos; sin embargo, es importante recordar sus políticas de acceso para comprender los algoritmos que las utilicen.

Entre las operaciones comunes tenemos las siguientes:

- *push!(L, a)*: insertar a al final de la lista L.
- *pop!(L)*: remueve el último elemento en L.
- *deleteat!(L, pos)*: remueve el elemento en la posición *pos*, se desplazan los elementos.
- *insert!(L, pos, valor)*: inserta *valor* en la posición *pos* desplazando los elementos anteriores.

1.5.1. Ejercicios

- Implemente *insert!* y *deleteat!*
- ¿Cuál sería la implementación de *succ* y *pred* en una lista ligada?
- ¿Cuáles serían sus costos?
- Añadiendo más memoria, ¿cómo podemos mejorar *pred*?

1.6. Grafos

Otras estructuras de datos elementales son los *grafos*. Un grafo $G = (V, E)$ es una tupla compuesta por un conjunto de vértices V y el conjunto de aristas E. Por ejemplo, el grafo con $A = (\{a, b, c, d\}, \{(a, b), (b, c), (c, d), (d, a)\})$

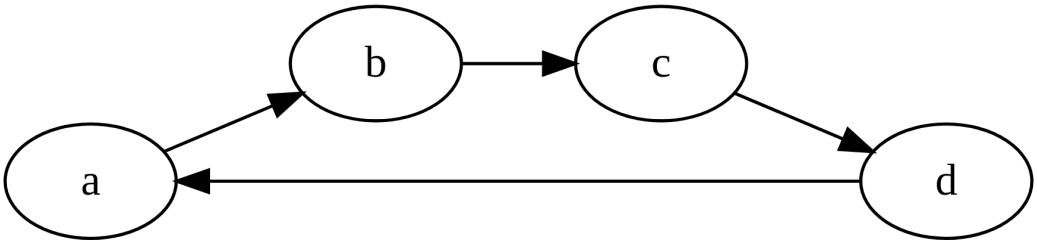

Figura 4: Un grafo dirigido simple

Los grafos son herramientas poderosas para representar de manera abstracta problemas que implican relaciones entre elementos. En algunos casos es útil asociar funciones a los vértices y las aristas. Tenga en cuenta los siguientes ejemplos:

- $peso : V \to \mathbb{R}$, la cual podría usarse como $peso(a) = 1.5$.
- $costo : V \times V \to \mathbb{R}$, la cual podría usarse como $costo(a, b) = 2.0$.

La estructura del grafo puede accederse mediante las funciones:

- $in(G, v) = \{u \mid (u, v) \in E\}$
- $out(G, u) = \{v \mid (u, v) \in E\}$

así como el número de vértices que entran y salen como:

- $indegree(G, v) = |in(G, v)|$.
- $outdegree(G, u) = |out(G, u)|$.

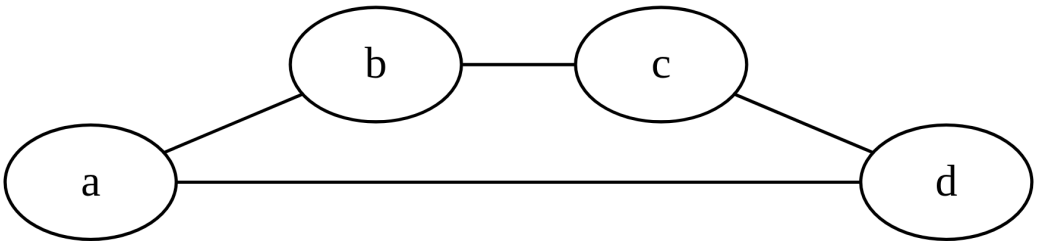

Figura 5: Un grafo cuyas aristas no están dirigidas

Un grafo puede tener aristas no dirigidas, el grafo con $B = (\{a, b, c, d\}, \{\{a, b\}, \{b, c\}, \{c, d\}, \{d, a\}\})$, no reconocerá orden en las aristas.

Por lo tanto, podremos decir que $(a, b) \in E_A$ pero $(b, a) \notin E_A$. Por otro lado tenemos que $\{a, b\} \in E_B$, y forzando un poco la notación, $(a, b) \in E_B$, $(b, a) \in E_B$; para los conjuntos de aristas de A y B. La estructura puede ser accedida mediante $neighbors(G, u) = \{v \mid \{u, v\} \in E\}$.

Un grafo puede estar representado de diferentes maneras, por ejemplo, un arreglo bidimensional (matriz), donde $S_{ij} = 1$ si hay una arista entre los vértices i y j; y $S_{ij} = 0$ si no existe una arista. A esta representación se le llama matriz de adyacencia. Si el grafo tiene pocos 1's vale la pena tener una representación diferente; este es el caso de las listas de adyacencia, donde se representa cada fila o cada columna de la matriz de adyacencia como una lista de los elementos diferentes de cero.

Existen otras representaciones como la lista de coordenadas, *coordinate lists* (COO), o las representaciones dispersas comprimidas, *sparse row* (CSR) y *compressed sparse column* (CSC) (Scott y Tůma 2023). Todas estas representaciones tratan de disminuir el uso de memoria y aprovechar la gran dispersión para realizar operaciones solo cuando sea estrictamente necesario.

Un *árbol* es un grafo en el cual no existen ciclos, esto es, no existe forma de, en una caminata sobre los vértices a través de las aristas y sin repetir aristas, regresar a un vértice antes visitado.

En algunos casos, es conveniente identificar vértices especiales en un árbol $T = (V, E)$. Un vértice es la *raíz* del árbol, $root(T)$, es especial ya que seguramente se utilizará como acceso al árbol y por tanto contiene un camino a cada uno de los vértices en V. Cada vértice puede tener, o no, *hijos* $children(T, u) = \{v \mid (u, v) \in E\}$. Se dice que u es una *hoja* (leaf) si $children(T, u) = \emptyset$, e *interno* (inner) si no es ni raíz ni hoja.

Al igual que en los grafos más generales, en los árboles es útil definir funciones sobre vértices y aristas, así como marcar tipos de vértices, e.g., posición u color, que simplifiquen el razonamiento para con los algoritmos asociados.

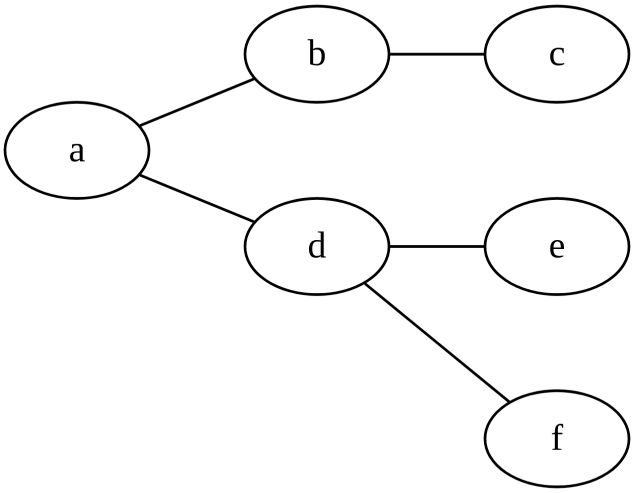

Figura 6: Árbol con aristas no dirigidas

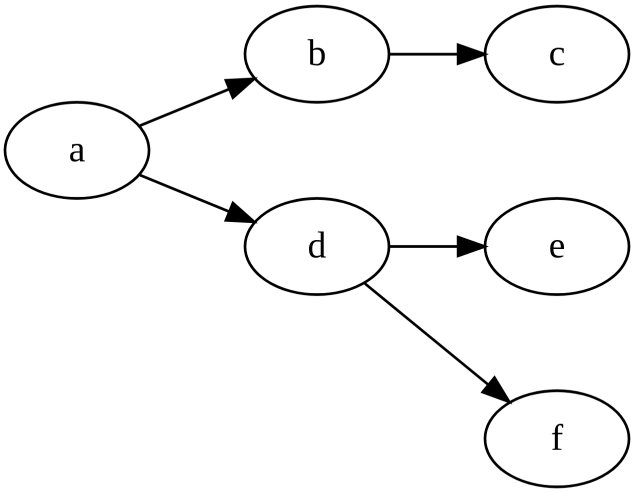

Figura 7: Árbol con aristas dirigidas, note que es fácil saber si hay un vértice o nodo que se distinga como raíz, o nodos que sean hojas.

Los nodos y las aristas de un grafo pueden *recorrerse* de diferentes maneras, donde se aprovechan las relaciones representadas. En un grafo general podría ser importante solo visitar una vez cada vértice, o guiarse en el recorrido por alguna heurística o función asociada a vértices o aristas.

El recorrido *primero a lo profundo*, Depth First Search (DFS), comienza en un nodo dado y de manera *voraz* avanzará recordando orden de visita y avanzando al ver un nuevo nodo repitiendo el procedimiento hasta que todos los vértices alcanzables sean visitados. El siguiente pseudo-código lo implementa:

```{julia}
#| lst-label: lst-dfs
#| lst-cap: Pseudo-código DFS

function operación!(vértice)
  #... operaciones sobre el vértice siendo visitado ...
end

function DFS(grafo, vértice, visitados)
  operación!(vértice)
  push!(visitados, vértice)
  for v in neighbors(grafo, vértice)
    if !(v in visitados)
      operación!(v)
      push!(visitados, v)
      DFS(grafo, v, visitados)
    end
  end
end

# ... código de preparación del grafo
visitados = Set()
DFS((vértices, aristas), vérticeinicial, visitados)
# ... código posterior a la visita DFS

```

Las llamadas recursivas a DFS tienen el efecto de *memorizar* el orden de visita anterior y *recordarlo* cuando se sale de una visita anidada. Entonces, hay una memoria implícita utilizada, implementada por el *stack* de llamadas. La función *operación!* es una abstracción de cualquier cosa que deba hacerse sobre los nodos siendo visitados.

El *recorrido a lo ancho*, Breadth First Search (BFS), visita los vértices locales antes que los alejados, contrario al avance voraz utilizado por DFS.

```
#| lst-label: lst-bfs
#| lst-cap: Pseudo-código BFS

function BFS(grafo, vértice, visitados, cola)
```

```
  operación!(vértice)
  push!(visitados, vértice)
  push!(cola, vértice)

  while length(cola) > 0
    u = popfirst!(cola)
    for v in neighbors(grafo, u)
      if v   visitados
        operación!(v)
        push!(visitados, v)
        push!(cola, v)
      end
    end
  end
end

# ... código de preparación del grafo
visitados = Set()
BFS((vértices, aristas), vérticeinicial, visitados)
# ... código posterior a la visita BFS
```

El BFS hace uso explícito de la memoria para guardar el orden en que se visitarán los vértices (*cola*); se utiliza un conjunto para marcar vértices ya visitados (*visitados*) con la finalidad de evitar un recorrido infinito.

1.6.1. Ejercicios

- Implemente un grafo dirigido mediante listas de adyacencia.
- Implemente un grafo no dirigido mediante lista de adyacencia.
- Implemente el algoritmo de recorrido DFS y BFS con implementaciones de grafos.

1.7. Actividades

1.7.1. Actividad 1: Producto punto máximo (top-1)

Dada la matriz Q de tamaño $d \times m$ y la matriz X de tamaño $d \times n$, para cada vector columna $q \in Q$, encontrar el vector columna $x \in X$ que maximiza su producto punto. Dicho de otra forma, el problema consiste en asociar cada q con $\arg\max\{q \cdot x \mid x \in X\}$. Este será el problema de *encontrar el producto punto máximo* o **top-1**. Obtenga un arreglo de identificadores de columna como resultado (*nns*) y otro con el valor del producto punto máximo (*dots*) para cada columna en Q.

$$\mathbf{Q} = \begin{pmatrix} q_{1,1} & q_{1,2} & \cdots & q_{1,m} \\ q_{2,1} & q_{2,2} & \cdots & q_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ q_{d,1} & q_{d,2} & \cdots & q_{d,m} \end{pmatrix}$$

$$\mathbf{X} = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{d,1} & x_{d,2} & \cdots & x_{d,n} \end{pmatrix}$$

a. Cree el algoritmo **A1** con operaciones matriciales explícitas.
b. Adapte el algoritmo *getmaxdot* de las notas (Cap. 2) para el mismo lenguaje de tu implementación para resolver el mismo problema, llamémoslo **A2**.

1. Considere un conjunto de datos como una tupla Q, X.
2. Considere las combinaciones de $m \in \{10^3, 10^6, 10^9\}$, $n \in \{10^3, 10^6, 10^9\}$, $d \in \{8, 16, 32\}$.
3. Las matrices Q y X deberán ser aleatorias cuyos vectores columna estén en la esfera unitaria.
4. Analice la memoria necesaria por cada algoritmo (**A1** y **A2**) para cada combinación de argumentos; grafique y discuta.
5. Reporte los costos en tiempo real para $m = 10^3, n = 10^6$ y $d = 16$ para cada **A1** y **A2**.
6. Grafique la distribución del arreglo *dots* para la corrida previa, i.e., $m = 10^3, n = 10^6$ y $d = 16$.

1.7.2. Entregable

Su trabajo se entregará en PDF y con el notebook fuente, se entregará en el sistema en archivos separados. El código se debe documentar, así como mantener una estructura del documento que permita a un lector interesado entender el problema, sus experimentos y metodología, así como sus conclusiones. Tenga en cuenta que los notebooks pueden alternar celdas de texto y código.

No olvide estructurar su reporte, en particular el reporte debe cubrir los siguientes puntos:

- Título del reporte, su nombre.
- Introducción.
- Actividad. Producto punto máximo (top-1).
- Conclusiones
- Lista de referencias. Nota, una lista de referencias que no fueron utilizadas en el cuerpo del texto será interpretada como una lista vacía.

Nota: no olvides revisar la rúbrica del curso.

1.8. Bibliografía

Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2022). Introduction to Algorithms (2nd ed.). MIT Press.

- Parte III: Cap 10 Elementary Data Structures.
- Parte VI: Cap 22 Elementary Graph Algorithms.
- Parte VII: Cap 28 Matrix Operations.

Scott, Jennifer, y Miroslav Tůma. 2023. “An Introduction to Sparse Matrices”. En *Algorithms for Sparse Linear Systems*. Springer International Publishing. https://doi.org/10.1007/978-3-031-25820-6_1.