Divide-and-Conquer Example 3: Multiplication

Integer Multiplication

- How do we multiply two integers x and y?
- Elementary school method: compute a partial product by multiplying every digit of y separately with x and then add up all the partial products.
- Remark: this method works the same in any base.

- Examples: $(12)_{10} \cdot (11)_{10}$ and $(1100)_2 \cdot (1011)_2$

```
   12
 × 11
------
   12
+ 12
------
  132
```

```
   1100
 × 1011
--------
   1100
  1100
 0000
+ 1100
--------
 10000100
```

Runtime of elementary multiplication algorithm

A reasonable model of computation: a **single** operation on a pair of digits (bits) is a primitive computational step.

Assume we are multiplying n-digit (bit) numbers.

- O(n) time to compute a partial product.
- O(n) time to combine it in a running sum of all partial products so far.

⇒ There are n partial products, each consisting of n bits, hence total number of operations is $O(n^2)$.

Can we do better?

Divide-and-conquer for multiplication

Consider n-digit decimal numbers x, y.

$$\begin{aligned} x &= x_{n-1}x_{n-2}\cdots x_0 \\ y &= y_{n-1}y_{n-2}\cdots y_0 \end{aligned}$$

Idea: rewrite each number as the sum of the n/2 high-order digits and the n/2 low-order digits.

$$\begin{aligned} x &= \underbrace{x_{n-1}\cdots x_{n/2}}_{x_H}\underbrace{x_{n/2-1}\cdots x_0}_{x_L} = x_H \cdot 10^{n/2} + x_L \\ y &= \underbrace{y_{n-1}\cdots y_{n/2}}_{y_H}\underbrace{y_{n/2-1}\cdots y_0}_{y_L} = y_H \cdot 10^{n/2} + y_L \end{aligned}$$

where each of x_H , x_L, y_H , y_L is an n/2-digit number.

Examples

$n = 2,\ x = 12,\ y = 11$

$$\underbrace{12}_{x} = \underbrace{1}_{x_H} \cdot \underbrace{10^1}_{10^{n/2}} + \underbrace{2}_{x_L}$$

$$\underbrace{11}_{y} = \underbrace{1}_{y_H} \cdot \underbrace{10^1}_{10^{n/2}} + \underbrace{1}_{y_L}$$

$n = 4,\ x = 1000,\ y = 1110$

$$\underbrace{1000}_{x} = \underbrace{10}_{x_H} \cdot \underbrace{10^2}_{10^{n/2}} + \underbrace{0}_{x_L}$$

$$\underbrace{1110}_{y} = \underbrace{11}_{y_H} \cdot \underbrace{10^2}_{10^{n/2}} + \underbrace{10}_{y_L}$$

Divide-and-conquer for multiplication

$$\begin{aligned} x \cdot y &= (x_H 10^{n/2} + x_L) \cdot (y_H 10^{n/2} + y_L) \\ &= x_H y_H \cdot 10^n + (x_H y_L + x_L y_H) \cdot 10^{n/2} + x_L y_L \end{aligned}$$

Observation: we reduced the problem of solving 1 instance of size n (i.e., one multiplication between two n-digit numbers) to the problem of solving 4 instances, each of size n/2 (i.e., computing the products $x_H y_H$, $x_H y_L$, $x_L y_H$ and $x_L y_L$).

This is a divide and conquer solution!

- Recursively solve the 4 subproblems (of half the size)
- Multiplication by 10^n is easy (shifting): O(n) time.
- Combine the solutions from the 4 subproblems to an overall solution using 3 additions on O(n)-digit numbers: O(n) time.

Karatsuba's Observation

Running time: $T(n) \le 4\,T(n/2) + cn$

- by the Master Theorem: $T(n) = O(n^2)$
- **no** improvement

However, if we only needed three n/2-digit multiplications, then by the Master theorem

$$T(n) \le 3T(n/2) + cn = O(n^{\log 3}) = O(n^{1.59}) = o(n^2).$$

Recall that

$$x \cdot y \;=\; x_H y_H \cdot 10^n + (x_H y_L + x_L y_H) 10^{n/2} + x_L y_L$$

Key observation: we dont need each of $x_H y_L$, $x_L y_H$, we only need their sum.

$$(x_H + x_L)(y_H + y_L) - x_H y_H - x_L y_L = x_H y_L + x_L y_H$$

Hence we only need 3 products: $x_H y_H,\ x_L y_L,\ (x_H + x_L)(y_H + y_L)$

Karatsuba's algorithm: pseudocode

Let k be a small constant.

Integer-Multiply(x, y)

    if $n \le k$ **then**
        return xy
    end if
    write $x = x_H\, 10^{n/2} + x_L$, and $y = y_H\, 10^{n/2} + y_L$
    compute $x_H + x_L$, $y_H + y_L$
    product = **Integer-Multiply**($x_H + x_L$, $y_H + y_L$)
    $x_H y_H$ = **Integer-Multiply**(x_H,y_H)
    $x_L y_L$ = **Integer-Multiply**(x_L, y_L)
    return $x_H\, y_H\, 10^n + (\text{product} - x_H\, y_H - x_L y_L)10^{n/2} + x_L y_L$

Concluding remarks

- To reduce the number of multiplications we do few more additions/subtractions: these are fast compared to multiplications.

- There is no reason to continue with recursion once n is small enough: the conventional algorithm is probably more efficient since it uses fewer additions.

- When we recursively compute $(x_H + x_L)(y_H + y_L)$, each of $x_H + x_L$, $y_H + y_L$ might be $(n/2 + 1)$-digit integers. This does not affect the asymptotics.

Practice Problem 1

Given an array of integers, find the contiguous subarray of at least one element which has the largest sum and return that sum.

- Brute-force solution is $O(n^3)$

  For each pair, compute the sum and then return the maximum

- Divide-and-conquer solution

```
max-subarray(A)
    if n==1 return A[1]
    mid = ceil(n/2)
    lmax = max-subarray(A[1…mid])
    rmax = max-subarray(A[mid+1…n])
    cmax = cross-max(A, mid)
    return max(lmax,rmax,cmax)
```

```
cross-max(A, mid)
    tempsum = 0, leftsum = -inf, rightsum = -inf
    for i = mid downto 1
        tempsum = tempsum + A[i]
        if tempsum > leftsum
            leftsum = tempsum
    tempsum = 0
    for i = mid+1 upto n
        tempsum = tempsum + A[i]
        if tempsum > rightsum
            rightsum = tempsum
    return leftsum+rightsum
```

- Runtime: $T(n) = 2T(n/2)+O(n) = O(n \log n)$

Practice Problem 2

Given an array of integers, find the total number of pairwise inversions, ie count how often A[i] > A[j] (for for i<j)

- Brute-force solution is $O(n^2)$

  For each pair i<j, count if A[i] > A[j]

- Divide-and-conquer solution

  (hint: variation of mergesort)

```
count-inversions(A)
    if n==1 return 0
    mid = ceil(n/2)
    linv = count-inversions (A[1…mid])
    rinv = count-inversions(A[mid+1…n])
    cinv = sort-and-count-cross-inversions(A, mid)
    return linv + rinv + cinv
```

```
sort-and-count-cross-inversions(A, mid)
    lptr = 1, rptr = mid+1, invcount=0
    while lptr<mid or rptr<n
      idx = 1
      if A[lptr] <= A[rptr]
          Anew[idx] = A[lptr]
          lptr = lptr+1
      else
          invcount = invcount + (mid-lptr)+1
          Anew[idx] = A[rptr]
          rptr = rptr+1
      idx = idx+1
  A = Anew
```

- Runtime: T(n) = 2T(n/2)+O(n) = O(n log n)

Graphs and Algorithms on Graphs

Graph

Definition: A directed graph consists of a finite set V of vertices (or nodes) and a set E of directed edges. A directed edge is an ordered pair of vertices (u, v).

- In mathematical terms, a directed graph G=(V,E) is just a binary relation $E \subseteq V \times V$ on a finite set V
- An undirected graph is the special case of a directed graph where $(u,v) \in E$ if and only if $(v,u) \in E$. In this case, an edge may be indicated as the unordered pair {u, v}
- A weighted (directed or undirected) graph is when each edge has a number (weight) associated with it.
- Notational conventions: $|V| = n$, $|E| = m$

Node degrees

- Undirected graphs

  deg(v) = number of edges incident to v

- Directed graphs

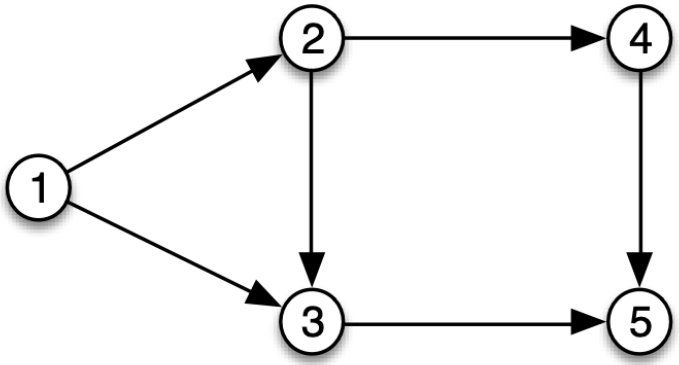
  indeg(v) = number of edges entering v

  outdeg(v) = number of edges leaving v

Examples

Circles denote vertices (nodes).
Lines denote edges connecting vertices.
Arrows on lines indicate the direction along which the edge may be traversed.

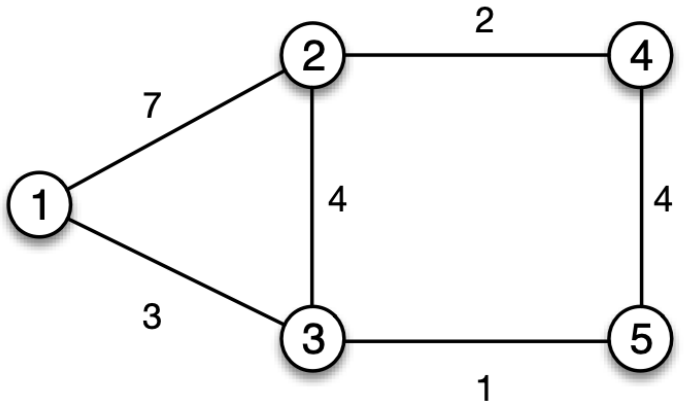
A directed, unweighted graph G
(default edge weight w(e) = 1)



indeg(1) = 0
indeg(3) = 2

outdeg(1) = 2
outdeg(3) = 1

An undirected, weighted graph G'



deg(1) = 2
deg(3) = 3

Applications of graphs (networks)

- Transportation networks: e.g., nodes are cities, edges (potentially weighted) are highways connecting the cities
  - Can we reach a city j from a city i?
  - If yes, what is the shortest (or cheapest) path?

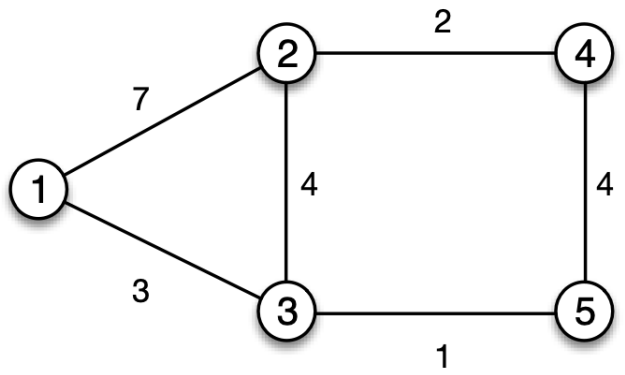
- Information networks: e.g., the World Wide Web can be modeled as a directed graph

- Wireless networks: nodes are devices sitting at locations in physical space and there is an edge from u to v if v is close enough to u to hear from it.

- Social networks: e.g., nodes are people, edges represent friendship

- Dependency networks: e.g., given a list of functions in a large program, find an order to test the functions.

Useful definitions



- A **path** is a sequence of vertices $(x_1,x_2,\ldots,x_n)$ such that consecutive vertices are adjacent, that is, there exists an edge $(x_i, x_{i+1}) \in E$ for all $1 \leq i \leq n-1$.

  Example: (1, 2, 3, 2, 4) in G′ is a path.

- A path is **simple** when all vertices are distinct.

  Example: (1, 2, 4) in G′ is a simple path.

- A **simple cycle** is a simple path that ends where it starts ($x_n = x_1$).

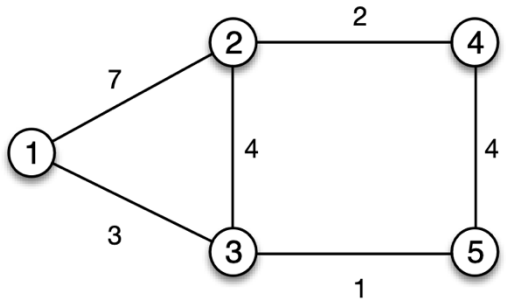
  Example: (1, 2, 3, 1) in G′ is a simple cycle.

Useful definitions

- The **distance** from u to v, denoted by dist(u, v), is the length of the shortest path from u to v.
  - Weighted graphs:

    Shortest path from u to v: a path of minimum length among all paths from u to v.

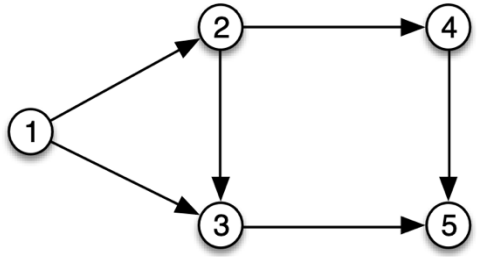
    Example: in G, dist(1, 4) = 8



  - Unweighted graphs: (assume weight on each edge is 1

    length of path P = # edges on P

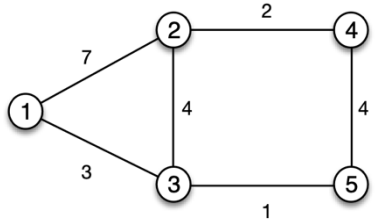
    Example: in G, dist(1, 4) = 2.



Useful definitions

- An undirected graph is **connected** when there is a path between every pair of vertices.

  Example: G' is connected.



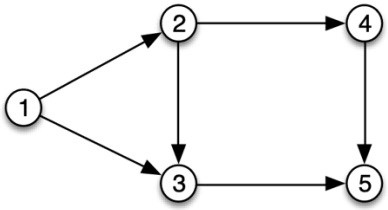
- The **connected component** of a node u is the set of all nodes in the graph reachable by a path from u.

  Example: the connected component of 1 in G' is {1, 2, 3, 4, 5}

- A directed graph is **strongly connected** if for every pair of vertices u, v, there is a path from u to v and from v to u.

- The **strongly connected component** of a node u in a directed graph is the set of nodes v in the graph such that there is a path from u to v and from v to u.

  Example: the strongly connected component of 1 in G is {1}.



Trees and its properties

Definition: A tree is a **connected acyclic** graph (undirected graphs). Or;
A **rooted** graph such that there is a unique path from the root to any other vertex (all graphs).

A tree is the most widely used special type of graph: it is the minimal connected graph.

Theorem: Let G be an undirected graph. Any two of the following properties imply the third property, and that G is a tree.

- G is connected
- G is acyclic
- $|E|=|V|-1$

Matchings and Bipartite graphs

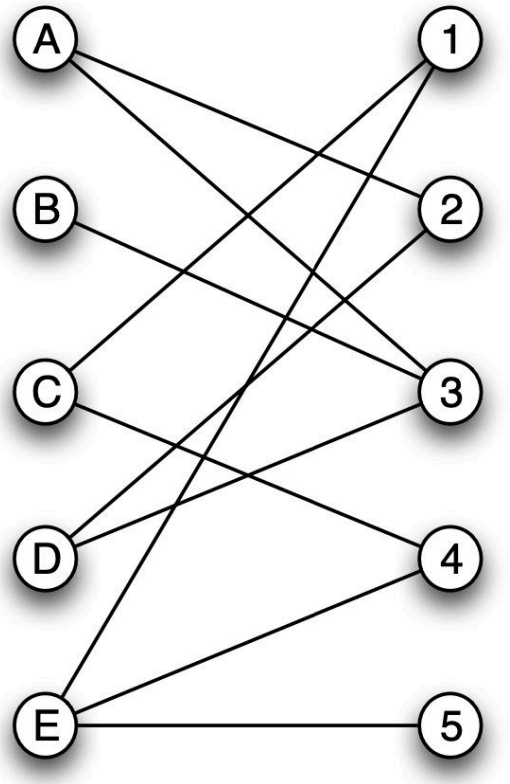
Bipartite graphs: vertices can be split into two subsets such that there are no edges between vertices in the same subset.

- Applications: social networks, coding theory
- Notation: $G = (X \cup Y, E)$, where X and Y is disjoint and $X \cup Y$ is the set of vertices in G and every edge in E has one endpoint in X and one endpoint in Y.

Example: suppose there are 5 people and 5 jobs and certain people qualify for certain jobs.

Matchings and Bipartite graphs

Matching: a subset of the edges where every node appears at most once.



Goal: find a **one-to-one matching** (also called, a perfect matching) of people to jobs, if one exists.

Degree theorem

Theorem: In any graph, the sum of the degrees of all vertices is equal to twice the number of the edges

Proof:
Every edge is incident to two vertices, thus contributes twice to the total sum of the degrees. (Summing the degrees of all vertices simply counts all instances of some edge being incident to some vertex.)

Running time of graph algorithms

Input: graph G=(V,E),|V|=n,|E|=m

- Linear graph algorithms run in O(n+m)time
  - Lower bound on m (assume connected graphs)?
  - Upper bound on m (assume simple graphs)?

- More general running times: the best performance is determined by the relationship between n and m
  - For example, $O(n^3)$ is better than $O(m^2)$ if the graph is dense (that is, $m = \Omega(n^2)$ edges)

Representing graphs: adjacency matrix

We want to represent a graph G = (V, E), |V | = n, |E| = m.

Adjacency matrix for G: an n × n matrix A such that

$$A[i,j] = \begin{cases} 1, & \text{if edge } (i,j) \in E \\ 0, & \text{otherwise} \end{cases}$$

Space required for adjacency matrix A: $\Theta(n^2)$.

Remark: Space requirements can be improved if the graph is

- undirected: A is symmetric ⇒ only store its upper triangle
- unweighted: only need 1 bit per entry

Pros/cons of adjacency matrix representation

Representing $G = (V, E)$, $|V| = n$, $|E| = m$ by its adjacency matrix has the following pros/cons.

Advantages:

- check whether edge $e \in E$ in constant time
- easy to adapt if the graph is weighted
- suitable for dense graphs where $m = \Theta(n^2)$

Drawbacks:

- requires $\Omega(n^2)$ space even if G is sparse ($m = o(n^2)$).
- does not allow for linear time algorithms in sparse graphs (at least when all matrix entries must be examined).

Representing graphs: adjacency list

An alternative representation for graph G = (V, E), |V| = n, |E| = m is as follows.

Adjacency list: recall that vertex j is adjacent to vertex i if (i, j) ∈ E; then the adjacency list for vertex i is simply the list of vertices adjacent to vertex i.

The adjacency list representation of a graph consists of an array A with n entries such that A[i] points to the adjacency list of vertex i.

Space requirements of adjacency list

Need

- an array of n pointers: O(n) space; plus
- the sum of the lengths of all adjacency lists:
  - directed G: maintain the list of vertices with incoming edges from v and the list of vertices with outgoing edges to v.
    - length of adjacency lists of v = outdeg(v) + indeg(v)
    - length of all adjacency lists = $\sum$ outdeg(v) + indeg(v) = 2m
  - undirected G: maintain the list of vertices adjacent to v
    - length of adjacency list of v = deg(v)
    - length of all adjacency lists = $\sum$ deg(v) = 2m

$\Rightarrow$ Total space: O(n + m)

Pros/cons of adjacency list representation

Representing G = (V, E), |V| = n, |E| = m by its adjacency list has the following pros/cons.

Advantages:

- Allocates no unnecessary space: O(n+m) space to represent a graph n vertices and m edges
- suitable for linear or non-linear time algorithms

Drawbacks:

- Searching for an edge takes O(n) time

Adjacency list vs adjacency matrix

We prefer **adjacency matrix** when

- we need determine quickly whether an edge is in the graph
- the graph is dense
- the graph is small (it is a simpler representation).

We use an **adjacency list** otherwise.

Searching a graph

Given a transportation network and a city s, we want to find all cities reachable from s.

This problem is known as s-t connectivity.

Input: a graph $G = (V,E)$, a vertex $s \in V$

Output: all vertices $t \in V$ such that there is a path from s to t

An algorithms for s-t connectivity: BFS

Breadth-first search (BFS): explore G starting from s **outward in all possible directions**, adding reachable nodes one **layer** at a time.

- First add all nodes that are joined by an edge to s: these nodes form the first layer.

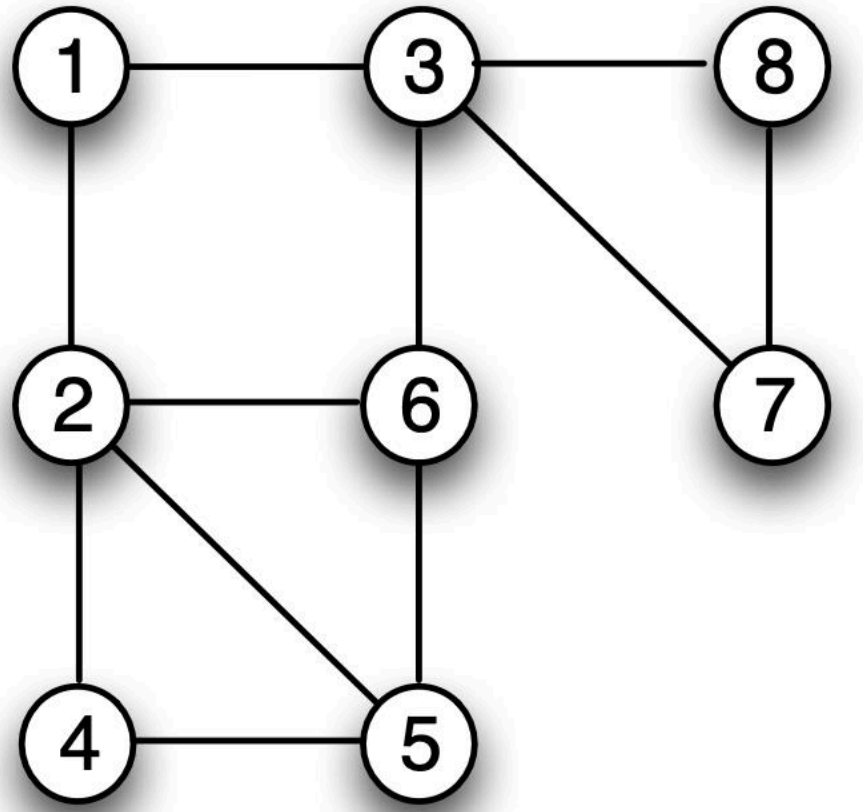
  If G is unweighted, these are the nodes at distance 1 from s.

- Then add all nodes that are joined by an edge to a node in the first layer: these nodes form the second layer.

  If G is unweighted, these are the nodes at distance 2 from s.

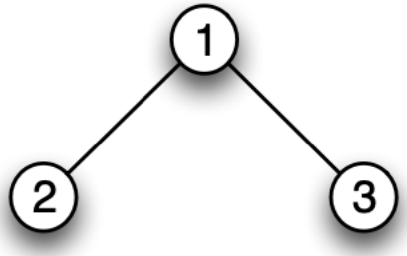
- And so on and so forth.

Example graph

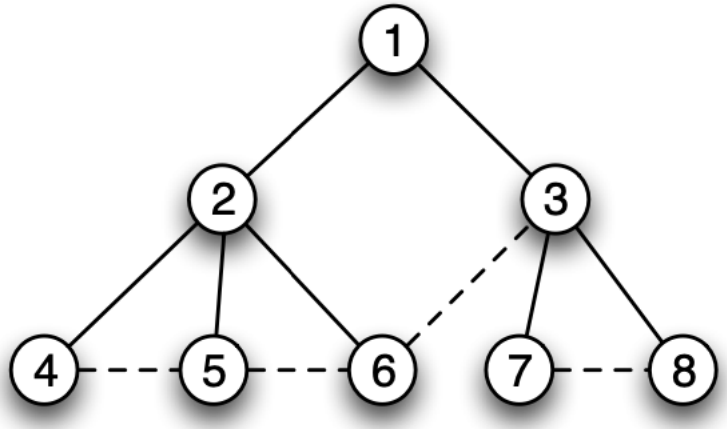


BFS layers of the example graph

Layer 1



Layer 2



Solid edges appear in G and in the output of BFS.
Dotted edges appear in G but do not in the output of BFS.
Ties are broken by selecting the node with the smallest index.

Properties of BFS output layers

Formally,

- Layer L_0 contains s.
- Layer L_1 contains all nodes v such that $(s, v) \in E$.
- For $i \geq 1$, layer L_i contains all nodes that
  - have an edge from a node in layer L_{i-1}; and
  - do not belong to a previous layer.

Fact:

L_i is the set of nodes that are at distance i from s.

Equivalently, the length of the shortest s-v path for all $v \in L_i$ equals i.

Properties of BFS output layers

Proof: (by induction)

Basis: true for layer L_0.

Hypothesis: suppose L_i is the set of nodes at distance i from s, for some $i \geq 0$.

Step: The only vertices added to L_{i+1} are those that

- have an edge from a node in L_i; and
- do not have an edge from a node in any previous layer L_k, for $k < i$

Then L_{i+1} contains the nodes that are at distance 1+i from s.

Properties of BFS

- When is a node v reachable from s added to the BFS graph?
  A node v is added to the tree when it is discovered, that is, when some node u is being explored and an edge (u, v) is found for the first time. Then u becomes the parent of v since u is responsible for discovering v.

- Why would a node fail to appear in the BFS graph?
  Because there is no path from s to that node.

- Why is the graph produced by BFS a tree?
  Because it is connected and acyclic.
  - The order in which the vertices are visited matters for the final tree but not for the distances computed.

- What are graph edges that do not appear in the BFS tree?
  - They are either
    - edges between nodes in the same layer; or
    - edges between nodes in adjacent layers.

  This is a property of all BFS trees

Implementing of BFS

We need to store the nodes **discovered** at layer L_i in order to *explore* them later (once we have finished exploring layer L_i).

To this end we use a queue.

- FIFO data structure: add to the end of the queue, extract from the head of the queue.
- Implemented as a double-linked list: maintain explicit pointers to the head and tail elements. Then **enqueue** and **dequeue** operations take constant time.

BFS pseudocode

```
BFS(G = (V, E), s ∈ V)
    array discovered[V] initialized to 0
    array dist[V] initialized to ∞
    array parent[V] initialized to NIL
    queue q
    discovered[s] = 1
    dist[s] = 0
    parent[s] = NIL
    enqueue(q, s)
    while size(q) > 0 do
        u =dequeue(q)
        for (u, v) ∈ E do
            if discovered[v] == 0 then
                discovered[v] = 1
                dist[v] = dist[u] + 1
                parent[v] = u
                enqueue(q, v)
            end if
        end for
    end while
```

BFS applications: connected components

- BFS(s) naturally produces the connected component R(s) of vertex s, that is, the set of nodes reachable from s.
  - Exploring the vertices in a different order can yield different algorithms for finding connected.

- How can we produce all the connected components of G?
  - Consider two distinct vertices s and t in G: how do their connected components compare?

BFS applications: connected components

Fact: For any two vertices u and v their connected components are either the same or disjoint.

Proof:
Consider any two nodes s, t such that there is a path between them: their connected components are the same (why?).

Now consider any two nodes s, t such that there is no path between them: their connected components are disjoint. If not, there is a node v that belongs to both components, hence a path between s and v and a path between t and v. Then there is a path between s and t, contradiction.

BFS applications: connected components

AllConnectedComponents(G = (V, E))

- Start with an arbitrary node s; run BFS(G, s) and output the resulting BFS tree as one connected component.
- Continue with any node u that has not been visited by BFS(G, s); run BFS from u and output the resulting BFS tree as one connected component.
- Repeat until all nodes in V have been visited.