Algorithms and Data Structures (II)

Gabriel Istrate

April 29, 2020

Where we are (II)

Why did we want dynamic sets to start with ?

- To improve algorithms.
- Today: brief respite from data structures.
- Computational geometry
- Time permitting: graph algorithms.
- We’ll see some algorithms that use stacks, queues, red-black trees.

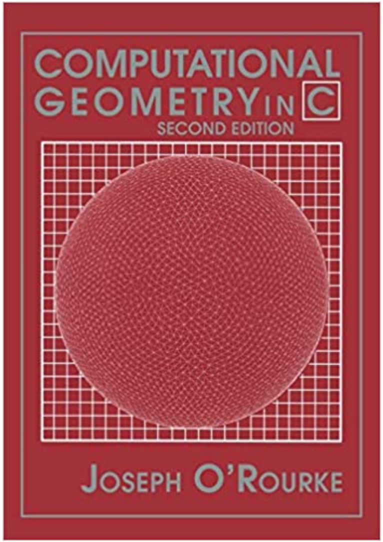
Computational geometry

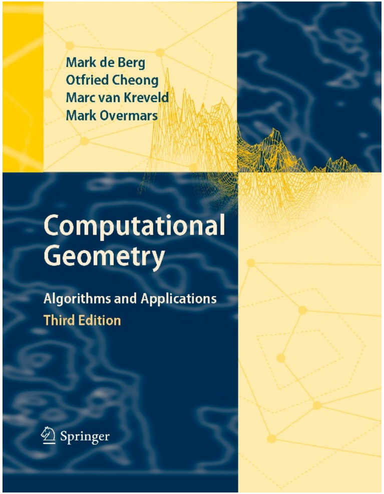
- Studies algorithms for geometric problems.
- Applications: computer graphics, robotics, VLSI, CAD.
- More applications: protein folding, molecular modeling, GIS.
- Huge area ! Only a sampler.
- Scientific conference: SOCG
- Software: CGAL.

Caution

- The biggest "enemy" to algorithms in computational geometry: degeneracy.
- Three points are collinear, three lines intersect at the same point, etc.
- Algorithms need patching to deal with degenerate situations.
- In the interest of teaching: Ignore it.

Want to know more ?

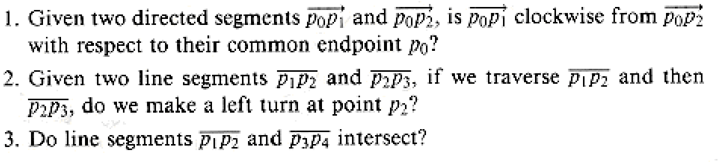




Second book: can LEGALLY download pdf from Springer. See message on the elearning forum for address (or search for it on Google).

Computational geometry

- Input: set of points $\{p_i\}$, $p_i = (x_i, y_i)$. Example: polygon $P = (p_0, p_1, \ldots, p_n)$.
- Given $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, convex combination: any point $p_3 = (x_3, y_3)$ such that $x_3 = \lambda x_1 + (1 - \lambda)x_2, \lambda \in [0, 1]$, similarly $y_3 = \lambda y_1 + (1 - \lambda)y_2$.



1. Given two directed segments $\overrightarrow{p_0p_1}$ and $\overrightarrow{p_0p_2}$, is $\overrightarrow{p_0p_1}$ clockwise from $\overrightarrow{p_0p_2}$ with respect to their common endpoint p_0?
2. Given two line segments $\overline{p_1p_2}$ and $\overline{p_2p_3}$, if we traverse $\overline{p_1p_2}$ and then $\overline{p_2p_3}$, do we make a left turn at point p_2?
3. Do line segments $\overline{p_1p_2}$ and $\overline{p_3p_4}$ intersect?

Cross products

$$
\begin{aligned}
p_1 \times p_2 &= \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} \\
&= x_1 y_2 - x_2 y_1 \\
&= -p_2 \times p_1 \,.
\end{aligned}
$$

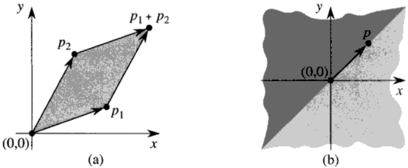


Figure 33.1 **(a)** The cross product of vectors p_1 and p_2 is the signed area of the parallelogram. **(b)** The lightly shaded region contains vectors that are clockwise from p. The darkly shaded region contains vectors that are counterclockwise from p.

Using Cross products

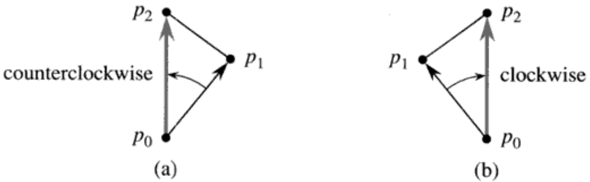


Figure 33.2 Using the cross product to determine how consecutive line segments $\overline{p_0p_1}$ and $\overline{p_1p_2}$ turn at point p_1. We check whether the directed segment $\overrightarrow{p_0p_2}$ is clockwise or counterclockwise relative to the directed segment $\overrightarrow{p_0p_1}$. **(a)** If counterclockwise, the points make a left turn. **(b)** If clockwise, they make a right turn.

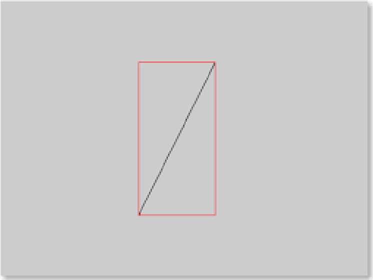
Procedures DIRECTION and ON-SEGMENT

```
DIRECTION($p_i, p_j, p_k$)
    return $(p_k - p_i) \times (p_j - p_i)$
```

```
ON-SEGMENT($p_i, p_j, p_k$)
    if $\min(x_i, x_j) \le x_k \le \max(x_i, x_j)$ and $\min(y_i, y_j) \le y_k \le \max(y_i, y_j)$
        then return TRUE
        else return FALSE
```

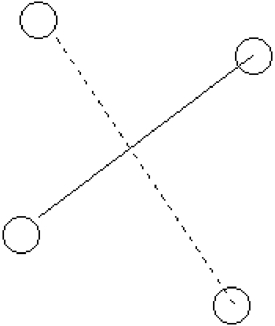
Testing whether two segments intersect

- QUICK REJECT: two segments cannot intersect if their BOUNDING BOXES don't.
- Smallest rectangle containing the segment with sides parallel to the xy axes.
- Bounding box of $\overline{p_1 p_2}$, $p_i = (x_i, y_i)$ is rectangle with corners $(min(x_1, x_2), min(y_1, y_2)$, $(min(x_1, x_2), max(y_1, y_2)$ $(max(x_1, x_2), max(y_1, y_2)$ and $(max(x_1, x_2), min(y_1, y_2)$.



Straddling

- Second stage: check whether each segment "straddles" the other.
- A segment $\overline{p_1 p_2}$ straddles a line if point p_1 lies on one side of the line and point p_2 lies on the other side. If p_1 or p_2 lies on the line, then we say that the segment straddles the line. Two line segments intersect if and only if they pass the quick rejection test and each segment straddles the line containing the other.



Straddling

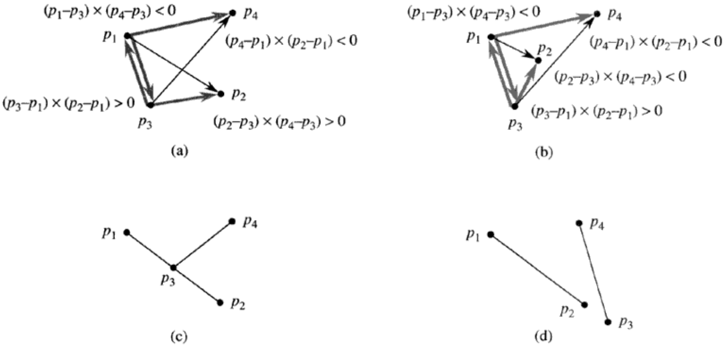
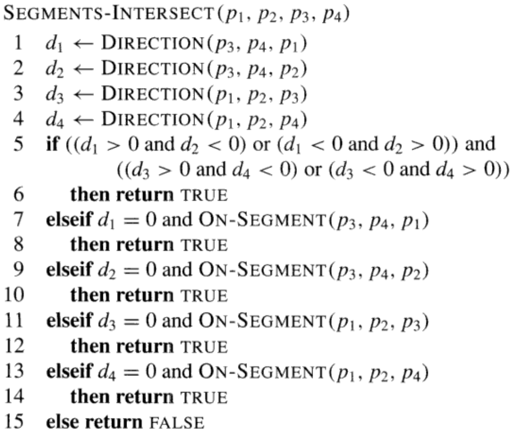


Figure 33.3 Cases in the procedure SEGMENTS-INTERSECT. **(a)** The segments $\overline{p_1p_2}$ and $\overline{p_3p_4}$ straddle each other's lines. Because $\overline{p_3p_4}$ straddles the line containing $\overline{p_1p_2}$, the signs of the cross products $(p_3-p_1)\times(p_2-p_1)$ and $(p_4-p_1)\times(p_2-p_1)$ differ. Because $\overline{p_1p_2}$ straddles the line containing $\overline{p_3p_4}$, the signs of the cross products $(p_1-p_3)\times(p_4-p_3)$ and $(p_2-p_3)\times(p_4-p_3)$ differ. **(b)** Segment $\overline{p_3p_4}$ straddles the line containing $\overline{p_1p_2}$, but $\overline{p_1p_2}$ does not straddle the line containing $\overline{p_3p_4}$. The signs of the cross products $(p_1-p_3)\times(p_4-p_3)$ and $(p_2-p_3)\times(p_4-p_3)$ are the same. **(c)** Point p_3 is collinear with $\overline{p_1p_2}$ and is between p_1 and p_2. **(d)** Point p_3 is collinear with $\overline{p_1p_2}$, but it is not between p_1 and p_2. The segments do not intersect.

Testing whether two segments intersect

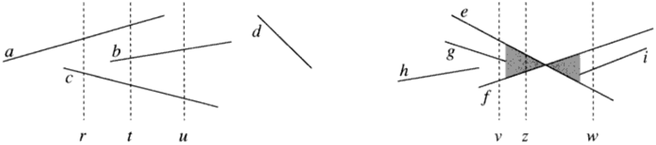


```
SEGMENTS-INTERSECT($p_1, p_2, p_3, p_4$)
$d_1 \leftarrow$ DIRECTION($p_3, p_4, p_1$)
$d_2 \leftarrow$ DIRECTION($p_3, p_4, p_2$)
$d_3 \leftarrow$ DIRECTION($p_1, p_2, p_3$)
$d_4 \leftarrow$ DIRECTION($p_1, p_2, p_4$)
if (($d_1 > 0$ and $d_2 < 0$) or ($d_1 < 0$ and $d_2 > 0$)) and
        (($d_3 > 0$ and $d_4 < 0$) or ($d_3 < 0$ and $d_4 > 0$))
   then return TRUE
elseif $d_1 = 0$ and ON-SEGMENT($p_3, p_4, p_1$)
   then return TRUE
elseif $d_2 = 0$ and ON-SEGMENT($p_3, p_4, p_2$)
   then return TRUE
elseif $d_3 = 0$ and ON-SEGMENT($p_1, p_2, p_3$)
   then return TRUE
elseif $d_4 = 0$ and ON-SEGMENT($p_1, p_2, p_4$)
   then return TRUE
else return FALSE
```

Testing whether any two segments intersect

- Given: n segments $v_1, \ldots v_n$.
- To test: do any two segments intersect ?
- Uses technique called sweeping.
- Running time: $O(n \log n)$. Naive algorithm $O(n^2)$.
- SWEEPING: an imaginary vertical sweep line passes through the given set of geometric objects, usually from left to right. The spatial dimension that the sweep line moves across, in this case the x-dimension, is treated as a dimension of time.
- Provides method for ordering geometric objects, usually by placing them into a dynamic data structure, and for taking advantage of relationships among them.
- line-segment-intersection algorithm: considers all line-segment endpoints in left-to-right order and checks for an intersection each time it encounters an endpoint.

Sweeping



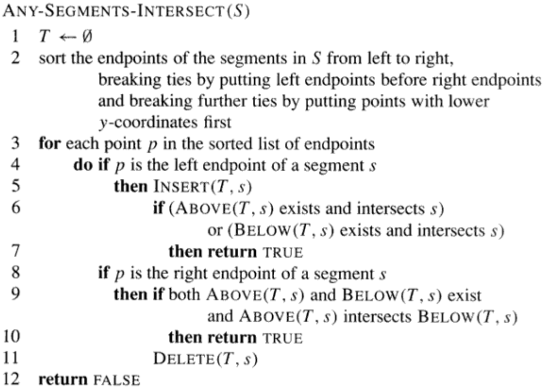
(a) (b)

Figure 33.4 The ordering among line segments at various vertical sweep lines. **(a)** We have $a >_r c$, $a >_t b$, $b >_t c$, $a >_t c$, and $b >_u c$. Segment d is comparable with no other segment shown. **(b)** When segments e and f intersect, their orders are reversed: we have $e >_v f$ but $f >_w e$. Any sweep line (such as z) that passes through the shaded region has e and f consecutive in its total order.

Maintaining sweep line

- Sweeping algorithms: maintain two sets of data.
- sweep-line status: gives the relationships among objects intersected by the sweep line.
- event-point schedule: sequence of x-coordinates, ordered from left to right, that defines the halting positions of the sweep line.
- Call each such halting position an event point. Changes to the sweep-line status occur only at event points.
- Sweep-line status: total order T.
- $INSERT(T, s)$, $DELETE(T, s)$.
- $ABOVE(T, s)$: return segment above s in T.
- $BELOW(T, s)$: return segment below s in T.
- We can perform each of the above operations in $O(\log n)$ time using red-black trees.

Algorithm



```
ANY-SEGMENTS-INTERSECT(S)
$T \leftarrow \emptyset$
sort the endpoints of the segments in $S$ from left to right,
        breaking ties by putting left endpoints before right endpoints
        and breaking further ties by putting points with lower
        $y$-coordinates first
for each point $p$ in the sorted list of endpoints
    do if $p$ is the left endpoint of a segment $s$
          then INSERT($T$, $s$)
               if (ABOVE($T$, $s$) exists and intersects $s$)
                      or (BELOW($T$, $s$) exists and intersects $s$)
                 then return TRUE
       if $p$ is the right endpoint of a segment $s$
          then if both ABOVE($T$, $s$) and BELOW($T$, $s$) exist
                      and ABOVE($T$, $s$) intersects BELOW($T$, $s$)
                 then return TRUE
               DELETE($T$, $s$)
return FALSE
```

Algorithm: example

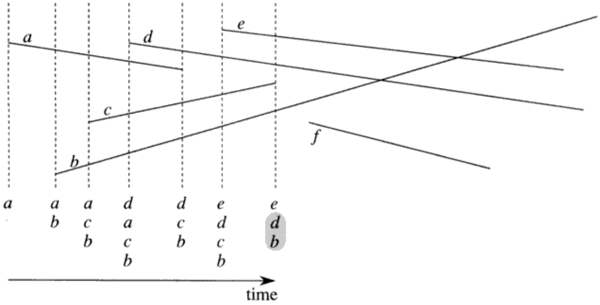


Figure 33.5 The execution of ANY-SEGMENTS-INTERSECT. Each dashed line is the sweep line at an event point, and the ordering of segment names below each sweep line is the total order T at the end of the **for** loop in which the corresponding event point is processed. The intersection of segments d and b is found when segment c is deleted.

Algorithm: correctness/performance

- Can only fail by not reporting intersecting segments.
- $p =$ leftmost intersection point, breaking ties by choosing the one with the lowest y-coordinate. a and $b =$ the segments that intersect at p.
- No intersections occur to the left of $p \Longrightarrow$ the order given by T is correct at all points to the left of p.
- no three segments intersect at the same point $\Longrightarrow$ there exists a sweep line z at which a and b become consecutive in the total order.
- z is to the left of p or goes through p.
- There exists segment endpoint q on z that is the event point at which a and b become consecutive.
- If p is on z, then $q = p$. If p is not on z, then q is to the left of p. In either case, the order given by T is correct just before q is processed.

Algorithm: correctness/performance

- Either a or b is inserted into T, and the other segment is above or below it in the total order. Lines 4-7 detect this case.
- Segments a and b are already in T, and a segment between them in the total order is deleted, making a and b become consecutive. Lines 8-11.
- In either case, the intersection p is found.
- $2n$ insert/delete/tests. Taking $O(\log n)$ time.

Convex hull

- Convex hull of a set of points: smallest convex polygon that contains the set of points.
- place elastic rubber band around set of points and let it shrink.
- Two algorithms: Graham's Scan $O(n \log n)$.
- Jarvis's March $O(n \cdot h)$, h the number of points on the convex hull.
- Other algorithms:
- Incremental: points sorted from left to right forming sequence $p_1, \ldots, p_n$. At stage i add p_i to convex hull $CH(p_1, \ldots, p_{i-1})$, forming $CH(p_1, \ldots, p_i)$.
- Divide-and-conquer: divide into leftmost n/2 points and rightmost n/2 points. Compute convex hulls and combine them.
- Prune-and-search method.

Convex hull

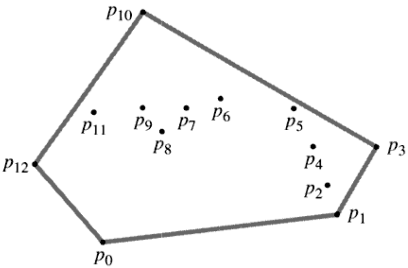
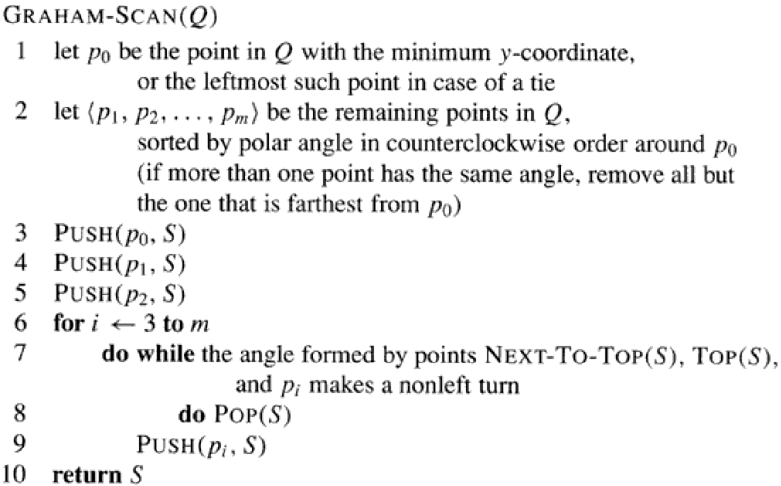


Figure 33.6 A set of points $Q = \{p_0, p_1, \ldots, p_{12}\}$ with its convex hull CH(Q) in gray.

Graham’s scan

- Maintains a stack S of candidate points.
- Each point of Q is pushed onto the stack.
- Points not in $CH(Q)$ eventually popped from the stack.
- $TOP(S)$, $NEXT-TO-TOP(S)$: stack functions, do not change its contents.
- Stack returned by the algorithm: points of $CH(Q)$ in counterclockwise order.

Convex hull algorithm



```
GRAHAM-SCAN($Q$)
let $p_0$ be the point in $Q$ with the minimum $y$-coordinate,
        or the leftmost such point in case of a tie
let $\langle p_1, p_2, \ldots, p_m \rangle$ be the remaining points in $Q$,
        sorted by polar angle in counterclockwise order around $p_0$
        (if more than one point has the same angle, remove all but
        the one that is farthest from $p_0$)
PUSH($p_0$, $S$)
PUSH($p_1$, $S$)
PUSH($p_2$, $S$)
for $i \leftarrow 3$ to $m$
    do while the angle formed by points NEXT-TO-TOP($S$), TOP($S$),
                and $p_i$ makes a nonleft turn
           do POP($S$)
       PUSH($p_i$, $S$)
return $S$
```

Graham’s Scan:Example

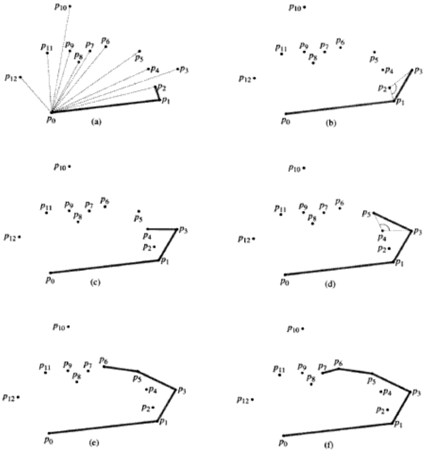
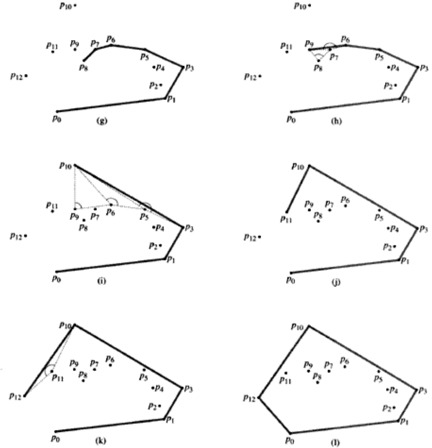


Figure 33.7 The execution of GRAHAM-SCAN on the set Q of Figure 33.6. The current convex hull contained in stack S is shown in gray at each step. **(a)** The sequence $\langle p_1, p_2, \ldots, p_{12} \rangle$ of points numbered in order of increasing polar angle relative to p_0, and the initial stack S containing p_0, p_1, and p_2. **(b)–(k)** Stack S after each iteration of the **for** loop of lines 6–9. Dashed lines show nonleft turns, which cause points to be popped from the stack. In part (h), for example, the right turn at angle $\angle p_7 p_8 p_9$ causes p_8 to be popped, and then the right turn at angle $\angle p_6 p_7 p_9$ causes p_7 to be popped. **(l)** The convex hull returned by the procedure, which matches that of Figure 33.6.

Graham's Scan:Example



(g) (h)

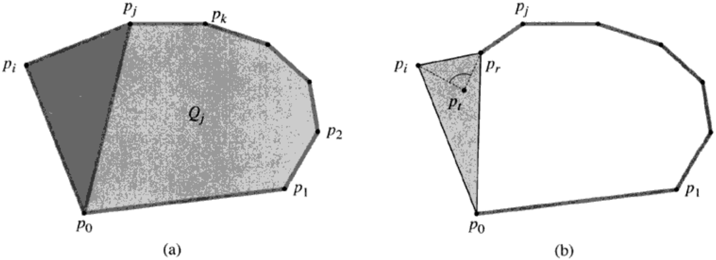
(i) (j)

(k) (l)

Graham's Scan: Correctness and Performance

- Invariant: at the beginning of each iteration of the for loop stack S contains (from bottom to top) exactly the vertices of $CH(Q_{i-1})$ in counterclockwise order.
- Line 1: $\theta(n)$ time.
- Sorting $\theta(n \log n)$ time.
- Testing for left/right turn: vector product $\theta(1)$ time.
- The rest of the algorithm $O(n)$ time.

Graham’s Scan: Correctness



(a) (b)

Figure 33.8 The proof of correctness of GRAHAM-SCAN. **(a)** Because p_i’s polar angle relative to p_0 is greater than p_j’s polar angle, and because the angle $\angle p_k p_j p_i$ makes a left turn, adding p_i to $\mathrm{CH}(Q_j)$ gives exactly the vertices of $\mathrm{CH}(Q_j \cup \{p_i\})$. **(b)** If the angle $\angle p_r p_t p_i$ makes a nonleft turn, then p_t is either in the interior of the triangle formed by p_0, p_r, and p_i or on a side of the triangle, and it cannot be a vertex of $\mathrm{CH}(Q_i)$.

Jarvis's March

- uses a technique known as gift wrapping.
- Simulates wrapping a piece of paper around set Q.
- Start at the same point p_0 as in Graham's scan.
- Pull the paper to the right, then higher until it touches a point. This point is a vertex in the convex hull. Continue this way until we come back to p_0.
- Formally: start at p_0. Choose p_1 as the point with the smallest polar angle from p_0. Choose p_2 as the point with the smallest polar angle from p_1 . . .
- . . . until we reached the highest point p_k.
- We have constructed the right chain.
- Construct the left chain by starting from p_k and measuring polar angles with respect to the negative x-axis.

Jarvis's March

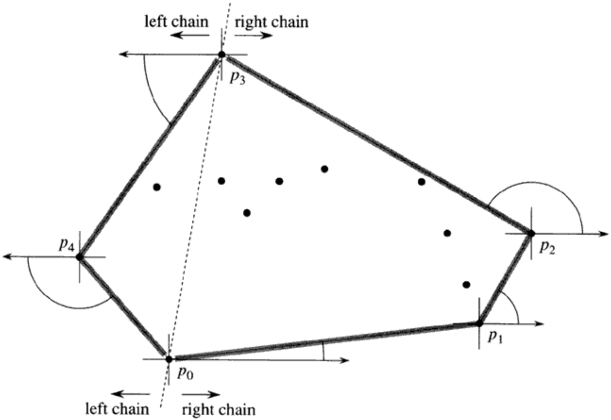


Figure 33.9 The operation of Jarvis's march. The first vertex chosen is the lowest point p_0. The next vertex, p_1, has the smallest polar angle of any point with respect to p_0. Then, p_2 has the smallest polar angle with respect to p_1. The right chain goes as high as the highest point p_3. Then, the left chain is constructed by finding smallest polar angles with respect to the negative x-axis.

Finding closest points

- W.r.t. euclidean distance.
- Brute force: $\theta(n^2)$.
- Divide and conquer algorithm with $O(n \log n)$ complexity.

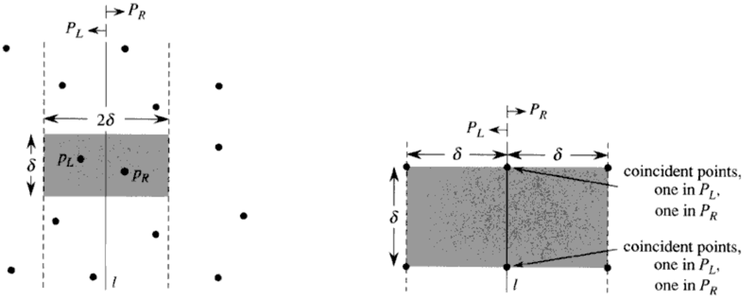
Finding closest points: Idea

- Each iteration: subset $P \subseteq Q$, arrays X and Y.
- Points in X are sorted in increasing order of their x coordinates.
- Points in Y are sorted in increasing order of their y coordinates.
- To maintain upper bound cannot afford to sort in each iteration.
- $|P| \leq 3$: brute force. Otherwise recursive divide-and-conquer.
- **Divide:** Find a vertical line l that bisects set P into two sets P_L and P_R such that $|P_L| = \lceil |P|/2 \rceil$, $|P_R| = \lfloor |P|/2 \rfloor$, all points of P_L to the left, all points of P_R to the right.
- X_L: subarray that contains point of P_L, X_R: subarray that contains point of P_R.
- Similarly for Y.

Finding closest points (III)

- **Conquer**. Recursive calls: P_L, X_L, Y_L and P_R, X_R, Y_R. Returns smallest distances δ_L and δ_R.
- **Combine.** $\delta = \min\{\delta_L, \delta_R\}$.
- Have to test whether some point in P_L is at distance $< \delta$ from some point in P_R.
- Both such points, if they exist, are within the 2δ-wide strip around l.
- Create an array Y' which is Y with all points not in the 2δ-wide strip around l removed, sorted by y-coordinate.
- **For each point p in Y'** try to find points in Y' at distance less than δ.
- Only the 7 points that follow p need to be considered.
- Compute smallest such distance δ'. If $\delta' < \delta$ we found a better pair. Otherwise δ is the smallest distance.
- Correctness, implementation nontrivial.

Finding closest points (IV)



(a)

(b)

Figure 33.11 Key concepts in the proof that the closest-pair algorithm needs to check only 7 points following each point in the array Y'. **(a)** If $p_L \in P_L$ and $p_R \in P_R$ are less than δ units apart, they must reside within a $\delta \times 2\delta$ rectangle centered at line l. **(b)** How 4 points that are pairwise at least δ units apart can all reside within a $\delta \times \delta$ square. On the left are 4 points in P_L, and on the right are 4 points in P_R. There can be 8 points in the $\delta \times 2\delta$ rectangle if the points shown on line l are actually pairs of coincident points with one point in P_L and one in P_R.

Correctness & complexity

- For each point: Consider the $\delta \times 2\delta$ rectangle centered at line *l*.
- At most 8 points within this rectangle.
- Assuming δ_L lower than δ_R, it follows that δ_R among the next 7 points following δ_L.
- $O(n \log n)$ bound from recurrence $T(n) = 2T(n/2) + O(n)$.
- Main difficulty: making sure that X_L, X_R, Y_L, Y_R, Y' sorted by appropriate coordinate.
- Key observation: in each call we wish to form a sorted subset of a sorted array.
- Splitting the array into two halves.
- Can be viewed as the inverse of the operation *MERGE* in *MERGESORT*.
- How to get sorted arrays in the first place ? presort. $\theta(n \log n)$.

Splitting: Pseudocode

```
$length[Y_L] = length[Y_R] = 0$;
for i = 1 to length[Y]
    if ($Y[i] \in P_L$)
    {
      $length[Y_L]$++;
      $Y_L[length[Y_L]] = Y[i]$;
    }
    else
    {
      $length[Y_R] + +$;
      $Y_R[length[Y_R]] = Y[i]$;
    }
}
```

And now for something totally different ...

Graph algorithms.

We live in a highly connected world ...

... and that's important.

A certain disease from Wuhan, China has dramatic effects all over the planet ...

A software bug in the alarm system at the control room of FirstEnergy, in Akron, Ohio knocks out the power grid in the whole Northeast United States (2003).

To understand, for my (and your) generation

How do real networks look like ?

How do network properties impact **the processes** that take place on them ?

Some real networks

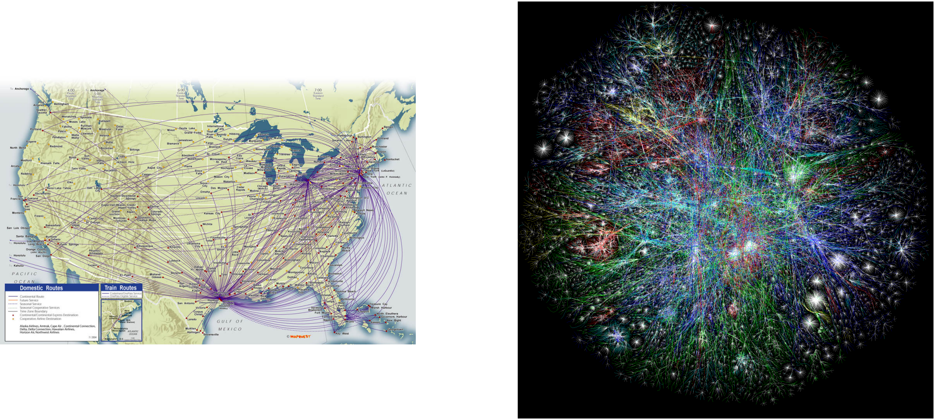


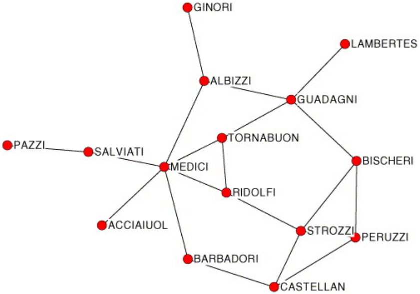
Figure: (a). Air traffic map of the U.S. (b). Physical Internet

Marriage Networks of important families in Medieval Florence.

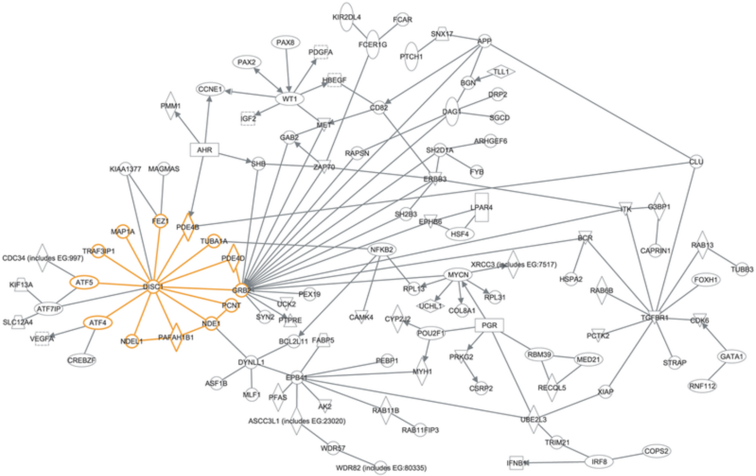
Paul D. McLean

The Art of the Network

STRATEGIC INTERACTION AND PATRONAGE
IN RENAISSANCE FLORENCE

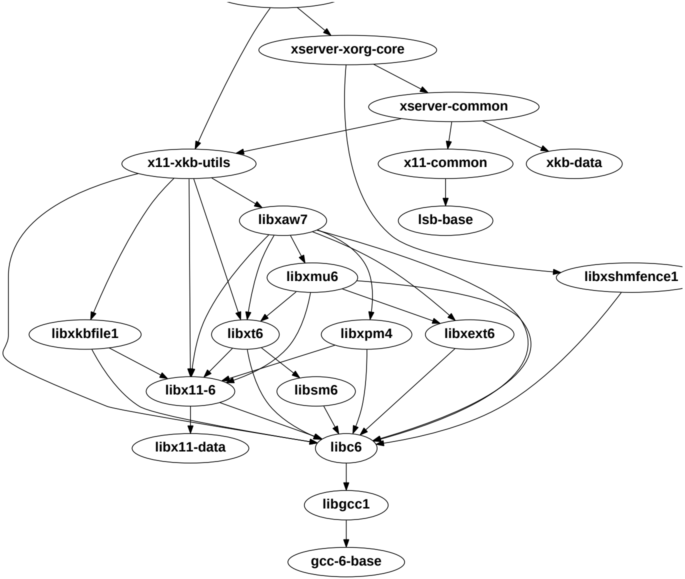


Interactome ...



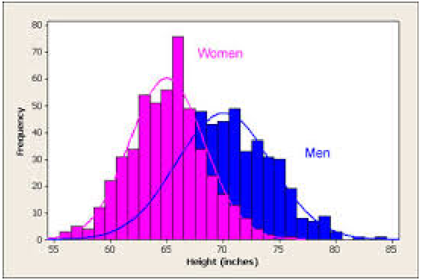
Part of the DISC1 interactome, with genes represented by text in boxes and interactions noted by lines between the genes. From Hennah and Porteous (2009).

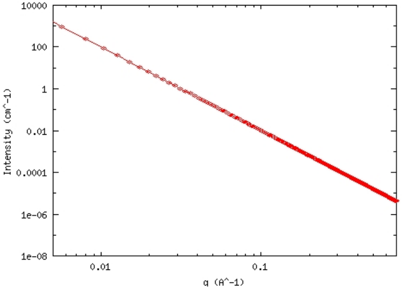
… or even …


xserver-xorg-core
xserver-common
x11-xkb-utils
x11-common
xkb-data
libxaw7
lsb-base
libxmu6
libxshmfence1
libxkbfile1
libxt6
libxpm4
libxext6
libx11-6
libsm6
libx11-data
libc6
libgcc1
gcc-6-base

What's so interesting about networks ?

Small worlds: everyone is "not very far from everyone".

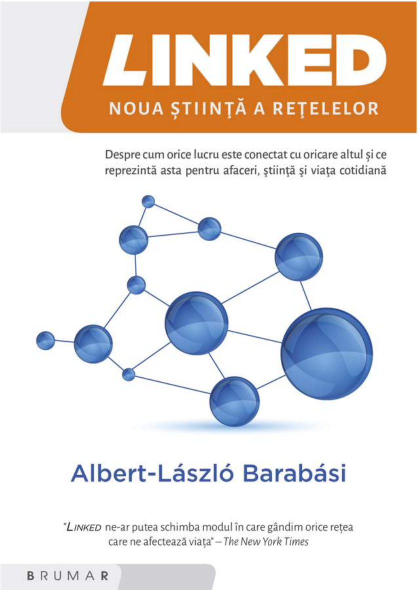




(a). Distribution of heights in the U.S. population.

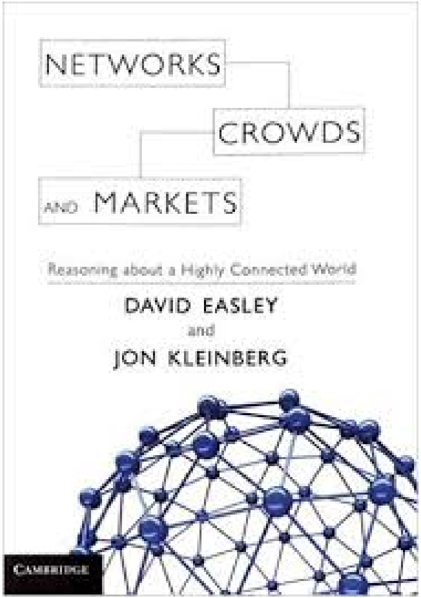
(b). Degree distribution (aproximately) power law. Few "tall" people, "many" well connected people

Want to read something interesting ?



By the way, not only in America ...

Want to read something even more interesting ?

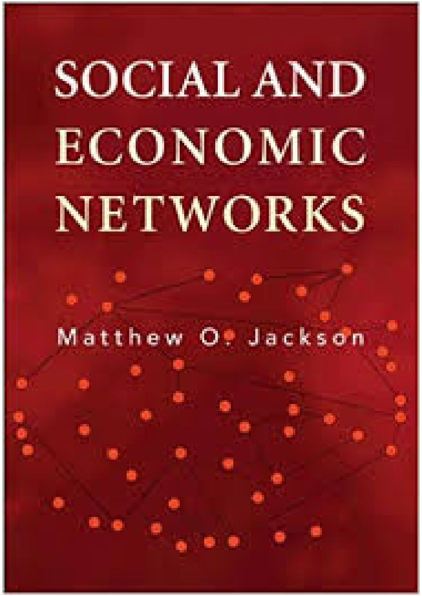


NETWORKS
CROWDS
AND MARKETS

Reasoning about a Highly Connected World

DAVID EASLEY
and
JON KLEINBERG

CAMBRIDGE



SOCIAL AND
ECONOMIC
NETWORKS

Matthew O. Jackson

What’s in it for us, Computer Scientists ?

Can you study large networks without good algorithms ?

To conclude: Many Models and Applications

- Social networks: *who knows who*
- The Web graph: *which page links to which*
- The Internet graph: *which router links to which*
- Citation graphs: *who references whose papers*
- Planar graphs: *which country is next to which*
- Well-shaped meshes: *pretty pictures with triangles*
- Geometric graphs: *who is near who*

Definitions

- A ***graph***

$$G = (V, E)$$

- V is the set of ***vertices*** (also called ***nodes***)
- E is the set of ***edges***

Definitions

- A ***graph***

$$G = (V, E)$$

- V is the set of ***vertices*** (also called ***nodes***)
- E is the set of ***edges***
  - $E \subseteq V \times V$, i.e., E is a ***relation between vertices***
  - an edge $e = (u, v) \in V$ is a pair of vertices $u \in V$ and $v \in V$

Definitions

- A ***graph***

$$G = (V, E)$$

- V is the set of ***vertices*** (also called ***nodes***)

- E is the set of ***edges***
  - $E \subseteq V \times V$, i.e., E is a ***relation between vertices***
  - an edge $e = (u, v) \in V$ is a pair of vertices $u \in V$ and $v \in V$

- An *undirected* graph is characterized by a *symmetric* relation between vertices
  - an edge is a set $e = \{u, v\}$ of two vertices

Graph Representation

- How do we represent a graph $G = (E, V)$ in a computer?

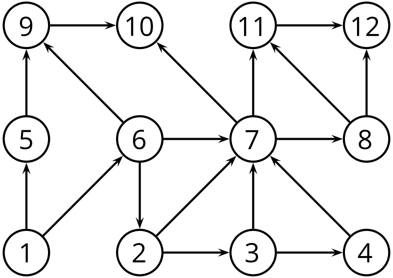
Graph Representation

- How do we represent a graph $G = (E, V)$ in a computer?
- *Adjacency-list representation*
- $V = \{1, 2, \ldots |V|\}$
- G consists of an array *Adj*
- A vertex $u \in V$ is represented by an element in the array *Adj*

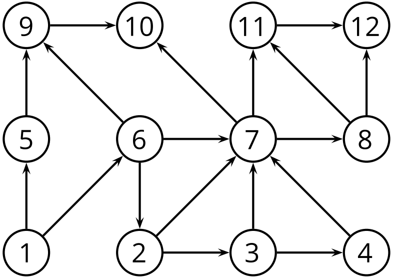
Graph Representation

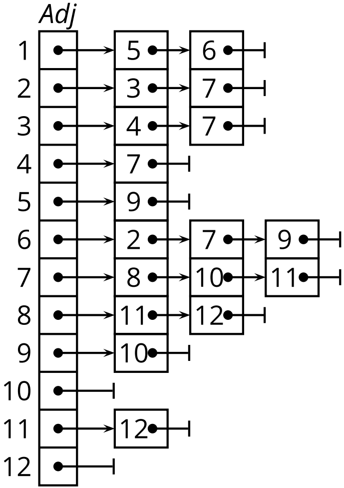
- How do we represent a graph $G = (E, V)$ in a computer?
- *Adjacency-list representation*
- $V = \{1, 2, \ldots |V|\}$
- G consists of an array Adj
- A vertex $u \in V$ is represented by an element in the array Adj
- $Adj[u]$ is the ***adjacency list*** of vertex u
  - the list of the vertices that are adjacent to u
  - i.e., the list of all v such that $(u, v) \in E$

Example

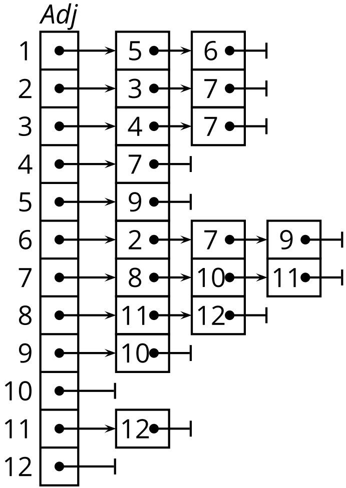


Example


9
10
11
12
5
6
7
8
1
2
3
4

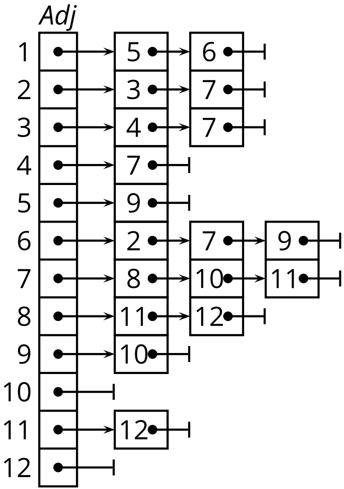

Adj
1 → 5 → 6
2 → 3 → 7
3 → 4 → 7
4 → 7
5 → 9
6 → 2 → 7 → 9
7 → 8 → 10 → 11
8 → 11 → 12
9 → 10
10
11 → 12
12

Using the Adjacency List



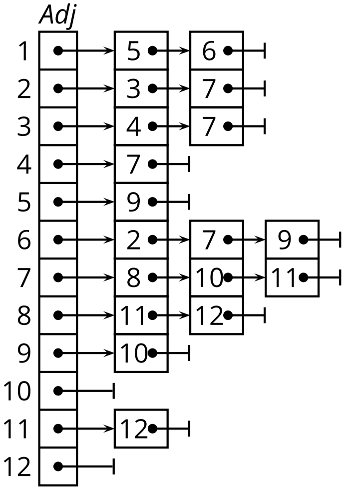
Using the Adjacency List

- Accessing a vertex *u*? ✓.



Using the Adjacency List

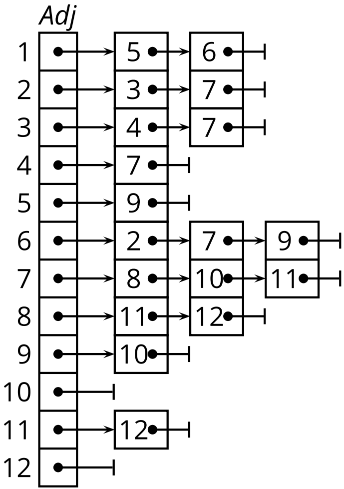
- Accessing a vertex *u*? $O(1)$ ✓.
  - optimal ✓



Using the Adjacency List

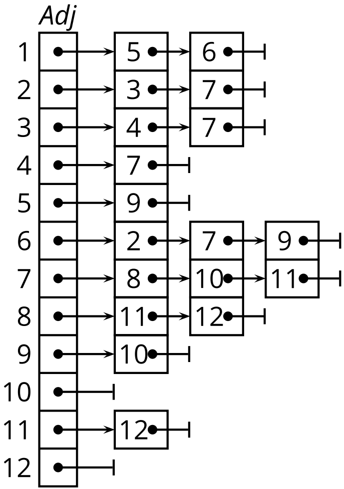
- Accessing a vertex u? $O(1)$ ✓.
  - optimal ✓

- Iteration through V?



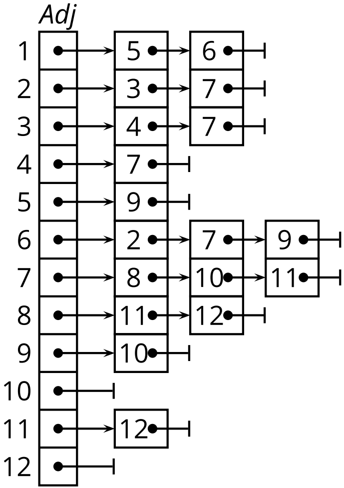
Using the Adjacency List

- Accessing a vertex *u*? $O(1)$ ✓.
  - optimal ✓
- Iteration through *V*? $\Theta(|V|)$
  - optimal ✓



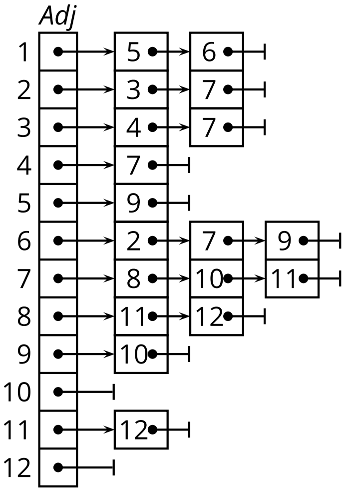
Using the Adjacency List

- Accessing a vertex u? $O(1)$ ✓.
    - optimal ✓
- Iteration through V? $\Theta(|V|)$
    - optimal ✓
- Iteration through E?



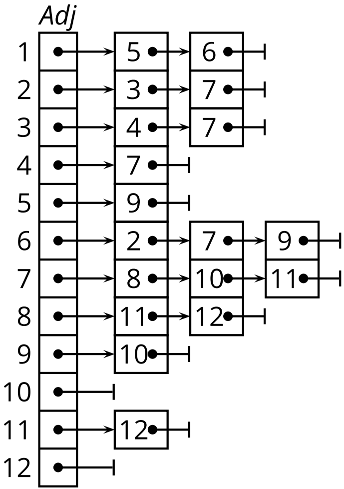
Using the Adjacency List

- Accessing a vertex *u*? $O(1)$ ✓.
  - optimal ✓
- Iteration through *V*? $\Theta(|V|)$
  - optimal ✓
- Iteration through *E*? $\Theta(|V| + |E|)$
  - okay (not optimal)



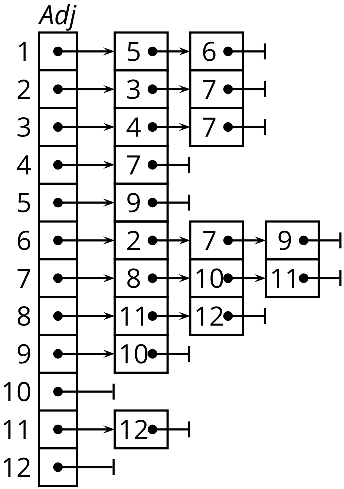
Using the Adjacency List

- Accessing a vertex u? $O(1)$ ✓.
  - optimal ✓
- Iteration through V? $\Theta(|V|)$
  - optimal ✓
- Iteration through E? $\Theta(|V| + |E|)$
  - okay (not optimal)
- Checking $(u, v) \in E$?



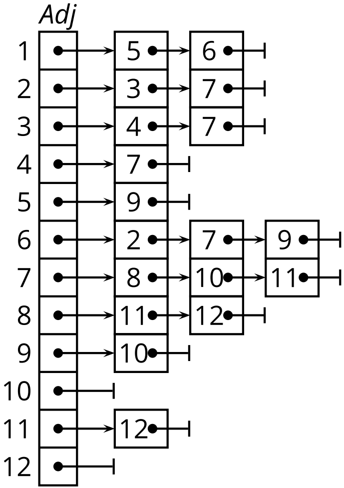
Using the Adjacency List

- Accessing a vertex u? $O(1)$ ✓.
  - optimal ✓
- Iteration through V? $\Theta(|V|)$
  - optimal ✓
- Iteration through E? $\Theta(|V| + |E|)$
  - okay (not optimal)
- Checking $(u, v) \in E$? $O(|V|)$



Using the Adjacency List

- Accessing a vertex u? $O(1)$ ✓.
  - optimal ✓
- Iteration through V? $\Theta(|V|)$
  - optimal ✓
- Iteration through E? $\Theta(|V| + |E|)$
  - okay (not optimal)
- Checking $(u, v) \in E$? $O(|V|)$
  - bad ✗



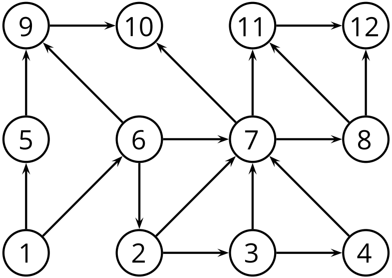
Graph Representation (2)

Graph Representation (2)

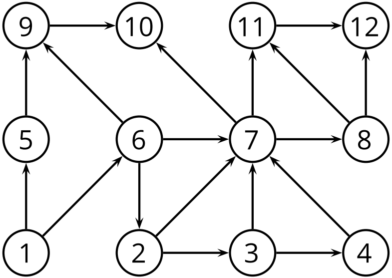
- *Adjacency-matrix representation*
- $V = \{1, 2, \ldots |V|\}$
- G consists of a $|V| \times |V|$ matrix A
- $A = (a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Example

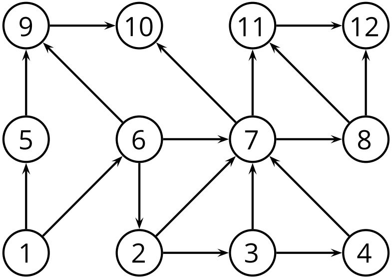


Example



	1	2	3	4	5	6	7	8	9	10	11	12
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												

Example



	1	2	3	4	5	6	7	8	9	10	11	12
1					1	1						
2			1				1					
3				1			1					
4							1					
5									1			
6		1					1		1			
7								1		1	1	
8											1	1
9										1		
10												
11												1
12												

Using the Adjacency Matrix

	1	2	3	4	5	6	7	8	9	10	11	12
1					1	1						
2			1				1					
3				1			1					
4							1					
5									1			
6		1					1		1			
7								1		1	1	
8											1	1
9										1		
10												
11												1
12												

Using the Adjacency Matrix

- Accessing a vertex *u*?

	1	2	3	4	5	6	7	8	9	10	11	12
1					1	1						
2			1				1					
3				1			1					
4							1					
5									1			
6		1					1		1			
7								1		1	1	
8											1	1
9										1		
10												
11												1
12												

Using the Adjacency Matrix

- Accessing a vertex *u*? $O(1)$
  - optimal ✓

	1	2	3	4	5	6	7	8	9	10	11	12
1					1	1						
2			1				1					
3				1			1					
4							1					
5									1			
6		1					1		1			
7								1		1	1	
8											1	1
9										1		
10												
11												1
12												

Using the Adjacency Matrix

- Accessing a vertex *u*? $O(1)$
  - optimal ✓
- Iteration through *V*?

	1	2	3	4	5	6	7	8	9	10	11	12
1					1	1						
2			1				1					
3				1			1					
4							1					
5									1			
6		1					1		1			
7								1		1	1	
8											1	1
9										1		
10												
11												1
12												

Using the Adjacency Matrix

- Accessing a vertex u? $O(1)$
  - optimal ✓
- Iteration through V? $\Theta(|V|)$
  - optimal ✓

	1	2	3	4	5	6	7	8	9	10	11	12
1					1	1						
2			1				1					
3				1			1					
4							1					
5									1			
6		1					1		1			
7								1		1	1	
8											1	1
9										1		
10												
11												1
12												

Using the Adjacency Matrix

- Accessing a vertex u? $O(1)$
  - optimal ✓
- Iteration through V? $\Theta(|V|)$
  - optimal ✓
- Iteration through E?

$$
\begin{array}{c|cccccccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\hline
1 & & & & & 1 & 1 & & & & & & \\
2 & & & 1 & & & & 1 & & & & & \\
3 & & & & 1 & & & 1 & & & & & \\
4 & & & & & & & 1 & & & & & \\
5 & & & & & & & & & 1 & & & \\
6 & & 1 & & & & & 1 & & 1 & & & \\
7 & & & & & & & & 1 & & 1 & 1 & \\
8 & & & & & & & & & & & 1 & 1 \\
9 & & & & & & & & & & 1 & & \\
10 & & & & & & & & & & & & \\
11 & & & & & & & & & & & & 1 \\
12 & & & & & & & & & & & & \\
\hline
\end{array}
$$

Using the Adjacency Matrix

- Accessing a vertex u? $O(1)$
  - optimal ✓
- Iteration through V? $\Theta(|V|)$
  - optimal ✓
- Iteration through E? $\Theta(|V|^2)$
  - possibly very bad ×.

	1	2	3	4	5	6	7	8	9	10	11	12
1					1	1						
2			1				1					
3				1			1					
4							1					
5									1			
6		1					1		1			
7								1		1	1	
8											1	1
9										1		
10												
11												1
12												

Using the Adjacency Matrix

- Accessing a vertex u? $O(1)$
    - optimal ✓
- Iteration through V? $\Theta(|V|)$
    - optimal ✓
- Iteration through E? $\Theta(|V|^2)$
    - possibly very bad ×.
- Checking $(u, v) \in E$?

	1	2	3	4	5	6	7	8	9	10	11	12
1					1	1						
2			1				1					
3				1			1					
4							1					
5									1			
6		1					1		1			
7								1		1	1	
8											1	1
9										1		
10												
11												1
12												

Using the Adjacency Matrix

- Accessing a vertex u? $O(1)$
  - optimal ✓
- Iteration through V? $\Theta(|V|)$
  - optimal ✓
- Iteration through E? $\Theta(|V|^2)$
  - possibly very bad ×.
- Checking $(u, v) \in E$? $O(1)$

	1	2	3	4	5	6	7	8	9	10	11	12
1					1	1						
2			1				1					
3				1			1					
4							1					
5									1			
6		1					1		1			
7								1		1	1	
8											1	1
9										1		
10												
11												1
12												

Using the Adjacency Matrix

- Accessing a vertex u? $O(1)$
  - optimal ✓
- Iteration through V? $\Theta(|V|)$
  - optimal ✓
- Iteration through E? $\Theta(|V|^2)$
  - possibly very bad ×.
- Checking $(u, v) \in E$? $O(1)$
  - optimal ✓

	1	2	3	4	5	6	7	8	9	10	11	12
1					1	1						
2			1				1					
3				1			1					
4							1					
5									1			
6		1					1		1			
7								1		1	1	
8											1	1
9										1		
10												
11												1
12												

Space Complexity

Space Complexity

- Adjacency-list representation

Space Complexity

- Adjacency-list representation

$$\boxed{\Theta(|V| + |E|)}$$

Space Complexity

- Adjacency-list representation

$$\boxed{\Theta(|V| + |E|)}$$

optimal .

Space Complexity

- Adjacency-list representation

$$\boxed{\Theta(|V| + |E|)}$$

  optimal .

- Adjacency-matrix representation

Space Complexity

- Adjacency-list representation

$$\boxed{\Theta(|V| + |E|)}$$

optimal .

- Adjacency-matrix representation

$$\boxed{\Theta(|V|^2)}$$

Space Complexity

- Adjacency-list representation

$$\boxed{\boxed{\Theta(|V| + |E|)}}$$

optimal .

- Adjacency-matrix representation

$$\boxed{\boxed{\Theta(|V|^2)}}$$

possibly very bad ×.

Space Complexity

- Adjacency-list representation

$$\boxed{\boxed{\Theta(|V| + |E|)}}$$

  optimal .

- Adjacency-matrix representation

$$\boxed{\boxed{\Theta(|V|^2)}}$$

  possibly very bad ×.

- When is the adjacency-matrix “very bad”?

Choosing a Graph Representation

- Adjacency-list representation
  - generally good, especially for its optimal space complexity
  - bad for ***dense*** graphs and algorithms that require random access to edges
  - preferable for ***sparse*** graphs or graphs with ***low degree***

Choosing a Graph Representation

- Adjacency-list representation
  - generally good, especially for its optimal space complexity
  - bad for ***dense*** graphs and algorithms that require random access to edges
  - preferable for ***sparse*** graphs or graphs with ***low degree***

- Adjacency-matrix representation
  - suffers from a bad space complexity
  - good for algorithms that require random access to edges
  - preferable for ***dense*** graphs

Breadth-First Search

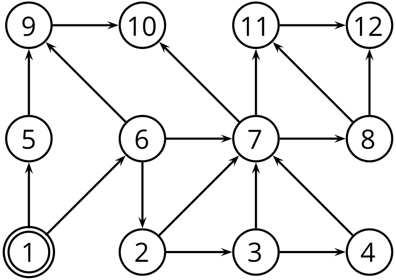
- One of the simplest but fundamental algorithms

Breadth-First Search

- One of the simplest but fundamental algorithms
- *Input:* $G = (V, E)$ and a vertex $s \in V$
  - explores the graph, touching all vertices that are reachable from s
  - iterates through the vertices at increasing distance (edge distance)
  - computes the distance of each vertex from s
  - produces a ***breadth-first tree*** rooted at s
  - works on both *directed* and *undirected* graphs

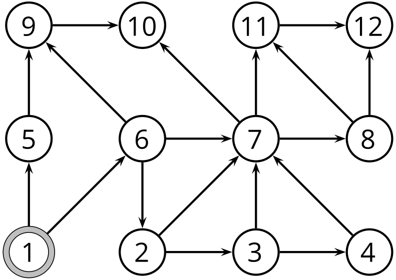
BFS Algorithm

```
BFS($G, s$)
for each vertex $u \in V(G) \setminus \{s\}$
    $color[u] =$ WHITE
    $d[u] = \infty$
    $\pi[u] =$ NIL
$color[s] =$ GRAY
$d[s] = 0$
$\pi[s] =$ NIL
$Q = \emptyset$
ENQUEUE($Q, s$)
while $Q \neq \emptyset$
    $u =$ DEQUEUE($Q$)
    for each $v \in Adj[u]$
        if $color[v] ==$ WHITE
            $color[v] =$ GRAY
            $d[v] = d[u] + 1$
            $\pi[v] = u$
            ENQUEUE($Q, v$)
    $color[u] =$ BLACK
```



BFS Algorithm

```
BFS($G, s$)
for each vertex $u \in V(G) \setminus \{s\}$
    $color[u] =$ WHITE
    $d[u] = \infty$
    $\pi[u] =$ NIL
$color[s] =$ GRAY
$d[s] = 0$
$\pi[s] =$ NIL
$Q = \emptyset$
ENQUEUE($Q, s$)
while $Q \neq \emptyset$
    $u =$ DEQUEUE($Q$)
    for each $v \in Adj[u]$
        if $color[v] ==$ WHITE
            $color[v] =$ GRAY
            $d[v] = d[u] + 1$
            $\pi[v] = u$
            ENQUEUE($Q, v$)
    $color[u] =$ BLACK
```

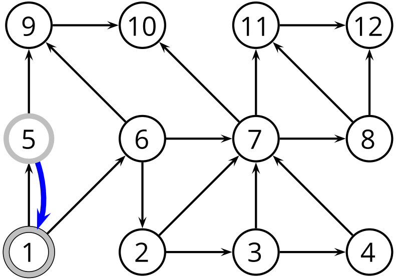


$u = 1$

$Q = \emptyset$

BFS Algorithm

```
BFS($G, s$)
    for each vertex $u \in V(G) \setminus \{s\}$
        $color[u] = $ WHITE
        $d[u] = \infty$
        $\pi[u] = $ NIL
    $color[s] = $ GRAY
    $d[s] = 0$
    $\pi[s] = $ NIL
    $Q = \emptyset$
    ENQUEUE($Q, s$)
    while $Q \neq \emptyset$
        $u = $ DEQUEUE($Q$)
        for each $v \in Adj[u]$
            if $color[v] == $ WHITE
                $color[v] = $ GRAY
                $d[v] = d[u] + 1$
                $\pi[v] = u$
                ENQUEUE($Q, v$)
        $color[u] = $ BLACK
```

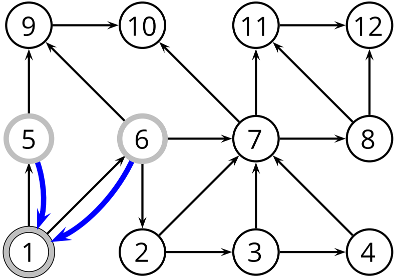


$u = 1$

$Q = \{5\}$

BFS Algorithm

```
**BFS**$(G, s)$
    **for** each vertex $u \in V(G) \setminus \{s\}$
        $color[u] = $ WHITE
        $d[u] = \infty$
        $\pi[u] = $ NIL
    $color[s] = $ GRAY
    $d[s] = 0$
    $\pi[s] = $ NIL
    $Q = \emptyset$
    ENQUEUE$(Q, s)$
    **while** $Q \neq \emptyset$
        $u = $ DEQUEUE$(Q)$
        **for** each $v \in Adj[u]$
            **if** $color[v] == $ WHITE
                $color[v] = $ GRAY
                $d[v] = d[u] + 1$
                $\pi[v] = u$
                ENQUEUE$(Q, v)$
        $color[u] = $ BLACK
```

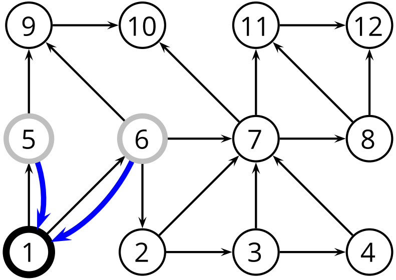


$u = 1$

$Q = \{5, 6\}$

BFS Algorithm

```
BFS$(G, s)$
for each vertex $u \in V(G) \setminus \{s\}$
    $color[u]$ = WHITE
    $d[u] = \infty$
    $\pi[u]$ = NIL
$color[s]$ = GRAY
$d[s] = 0$
$\pi[s]$ = NIL
$Q = \emptyset$
ENQUEUE$(Q, s)$
while $Q \neq \emptyset$
    $u$ = DEQUEUE$(Q)$
    for each $v \in Adj[u]$
        if $color[v]$ == WHITE
            $color[v]$ = GRAY
            $d[v] = d[u] + 1$
            $\pi[v] = u$
            ENQUEUE$(Q, v)$
    $color[u]$ = BLACK
```

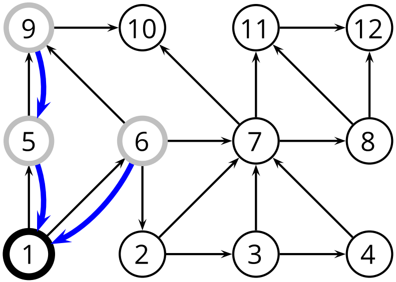


$u = 5$

$Q = \{6\}$

BFS Algorithm

```
BFS($G, s$)
for each vertex $u \in V(G) \setminus \{s\}$
    $color[u] = $ WHITE
    $d[u] = \infty$
    $\pi[u] = $ NIL
$color[s] = $ GRAY
$d[s] = 0$
$\pi[s] = $ NIL
$Q = \emptyset$
ENQUEUE($Q, s$)
while $Q \neq \emptyset$
    $u = $ DEQUEUE($Q$)
    for each $v \in Adj[u]$
        if $color[v] ==$ WHITE
            $color[v] = $ GRAY
            $d[v] = d[u] + 1$
            $\pi[v] = u$
            ENQUEUE($Q, v$)
    $color[u] = $ BLACK
```

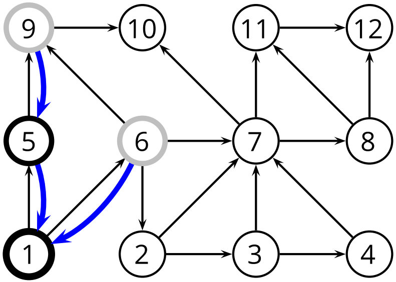


$u = 5$

$Q = \{6, 9\}$

BFS Algorithm

```
BFS($G, s$)
    for each vertex $u \in V(G) \setminus \{s\}$
        $color[u]$ = WHITE
        $d[u] = \infty$
        $\pi[u]$ = NIL
    $color[s]$ = GRAY
    $d[s] = 0$
    $\pi[s]$ = NIL
    $Q = \emptyset$
    ENQUEUE($Q, s$)
    while $Q \neq \emptyset$
        $u$ = DEQUEUE($Q$)
        for each $v \in Adj[u]$
            if $color[v]$ == WHITE
                $color[v]$ = GRAY
                $d[v] = d[u] + 1$
                $\pi[v] = u$
                ENQUEUE($Q, v$)
        $color[u]$ = BLACK
```

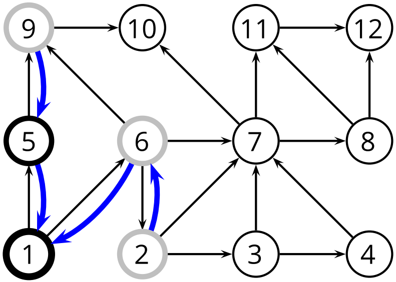


$u = 6$

$Q = \{9\}$

BFS Algorithm

```
BFS(G, s)
for each vertex $u \in V(G) \setminus \{s\}$
    $color[u] = $ WHITE
    $d[u] = \infty$
    $\pi[u] = $ NIL
$color[s] = $ GRAY
$d[s] = 0$
$\pi[s] = $ NIL
$Q = \emptyset$
ENQUEUE$(Q, s)$
while $Q \neq \emptyset$
    $u = $ DEQUEUE$(Q)$
    for each $v \in Adj[u]$
        if $color[v] == $ WHITE
            $color[v] = $ GRAY
            $d[v] = d[u] + 1$
            $\pi[v] = u$
            ENQUEUE$(Q, v)$
    $color[u] = $ BLACK
```

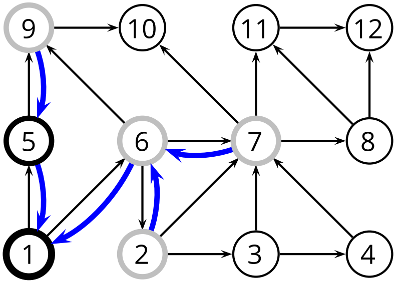


$u = 6$

$Q = \{9, 2, 7\}$

BFS Algorithm

```
BFS($G, s$)
    for each vertex $u \in V(G) \setminus \{s\}$
        $color[u]$ = WHITE
        $d[u] = \infty$
        $\pi[u]$ = NIL
    $color[s]$ = GRAY
    $d[s] = 0$
    $\pi[s]$ = NIL
    $Q = \emptyset$
    ENQUEUE($Q, s$)
    while $Q \neq \emptyset$
        $u$ = DEQUEUE($Q$)
        for each $v \in Adj[u]$
            if $color[v]$ == WHITE
                $color[v]$ = GRAY
                $d[v] = d[u] + 1$
                $\pi[v] = u$
                ENQUEUE($Q, v$)
        $color[u]$ = BLACK
```

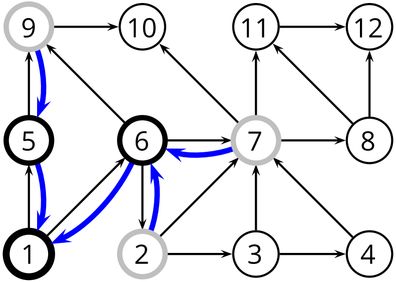


$u = 6$

$Q = \{9, 2, 7\}$

BFS Algorithm

```
BFS(G, s)
for each vertex $u \in V(G) \setminus \{s\}$
    $color[u] = $ WHITE
    $d[u] = \infty$
    $\pi[u] = $ NIL
$color[s] = $ GRAY
$d[s] = 0$
$\pi[s] = $ NIL
$Q = \emptyset$
ENQUEUE$(Q, s)$
while $Q \neq \emptyset$
    $u = $ DEQUEUE$(Q)$
    for each $v \in Adj[u]$
        if $color[v] == $ WHITE
            $color[v] = $ GRAY
            $d[v] = d[u] + 1$
            $\pi[v] = u$
            ENQUEUE$(Q, v)$
    $color[u] = $ BLACK
```

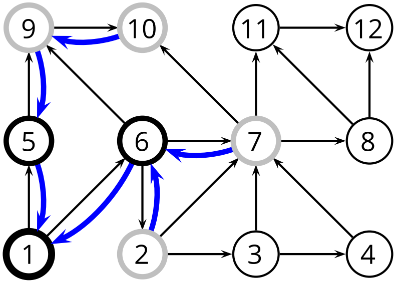


$u = 9$

$Q = \{2, 7\}$

BFS Algorithm

```
BFS(G, s)
for each vertex $u \in V(G) \setminus \{s\}$
    $color[u] =$ WHITE
    $d[u] = \infty$
    $\pi[u] =$ NIL
$color[s] =$ GRAY
$d[s] = 0$
$\pi[s] =$ NIL
$Q = \emptyset$
ENQUEUE$(Q, s)$
while $Q \neq \emptyset$
    $u =$ DEQUEUE$(Q)$
    for each $v \in Adj[u]$
        if $color[v] ==$ WHITE
            $color[v] =$ GRAY
            $d[v] = d[u] + 1$
            $\pi[v] = u$
            ENQUEUE$(Q, v)$
    $color[u] =$ BLACK
```

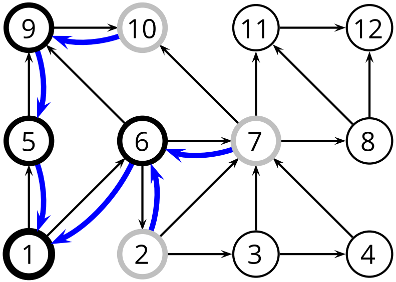


$u = 9$

$Q = \{2, 7, 10\}$

BFS Algorithm

```
BFS($G, s$)
for each vertex $u \in V(G) \setminus \{s\}$
    $color[u] = $ WHITE
    $d[u] = \infty$
    $\pi[u] = $ NIL
$color[s] = $ GRAY
$d[s] = 0$
$\pi[s] = $ NIL
$Q = \emptyset$
ENQUEUE($Q, s$)
while $Q \neq \emptyset$
    $u = $ DEQUEUE($Q$)
    for each $v \in Adj[u]$
        if $color[v] == $ WHITE
            $color[v] = $ GRAY
            $d[v] = d[u] + 1$
            $\pi[v] = u$
            ENQUEUE($Q, v$)
    $color[u] = $ BLACK
```

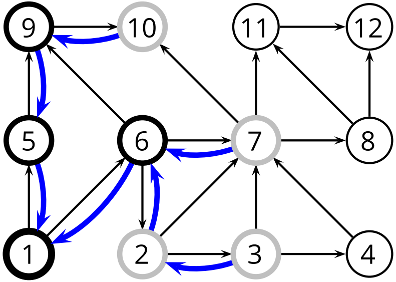


$u = 2$

$Q = \{7, 10\}$

BFS Algorithm

```
BFS($G, s$)
for each vertex $u \in V(G) \setminus \{s\}$
    $color[u] = $ WHITE
    $d[u] = \infty$
    $\pi[u] = $ NIL
$color[s] = $ GRAY
$d[s] = 0$
$\pi[s] = $ NIL
$Q = \emptyset$
ENQUEUE($Q, s$)
while $Q \neq \emptyset$
    $u = $ DEQUEUE($Q$)
    for each $v \in Adj[u]$
        if $color[v] == $ WHITE
            $color[v] = $ GRAY
            $d[v] = d[u] + 1$
            $\pi[v] = u$
            ENQUEUE($Q, v$)
    $color[u] = $ BLACK
```

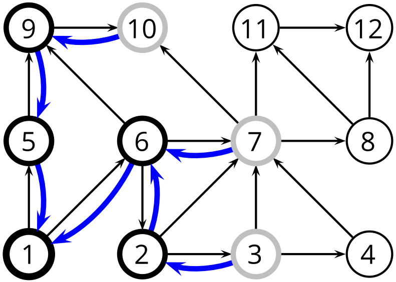


$u = 2$

$Q = \{7, 10, 3\}$

BFS Algorithm

```
BFS($G, s$)
    for each vertex $u \in V(G) \setminus \{s\}$
        $color[u] = $ WHITE
        $d[u] = \infty$
        $\pi[u] = $ NIL
    $color[s] = $ GRAY
    $d[s] = 0$
    $\pi[s] = $ NIL
    $Q = \emptyset$
    ENQUEUE($Q, s$)
    while $Q \neq \emptyset$
        $u = $ DEQUEUE($Q$)
        for each $v \in Adj[u]$
            if $color[v] ==$ WHITE
                $color[v] = $ GRAY
                $d[v] = d[u] + 1$
                $\pi[v] = u$
                ENQUEUE($Q, v$)
        $color[u] = $ BLACK
```

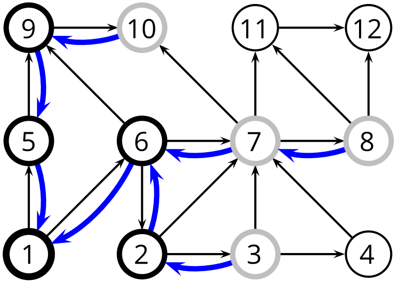


$u = 7$

$Q = \{10, 3\}$

BFS Algorithm

```
BFS($G, s$)
for each vertex $u \in V(G) \setminus \{s\}$
    $color[u] = $ WHITE
    $d[u] = \infty$
    $\pi[u] = $ NIL
$color[s] = $ GRAY
$d[s] = 0$
$\pi[s] = $ NIL
$Q = \emptyset$
ENQUEUE($Q, s$)
while $Q \neq \emptyset$
    $u = $ DEQUEUE($Q$)
    for each $v \in Adj[u]$
        if $color[v] == $ WHITE
            $color[v] = $ GRAY
            $d[v] = d[u] + 1$
            $\pi[v] = u$
            ENQUEUE($Q, v$)
    $color[u] = $ BLACK
```

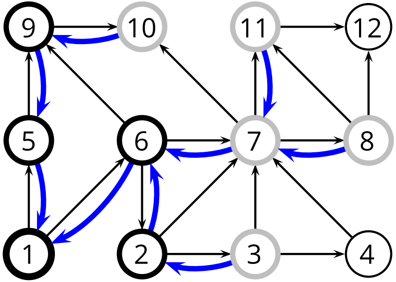


$u = 7$

$Q = \{10, 3, 8\}$

BFS Algorithm

```
BFS(G, s)
for each vertex $u \in V(G) \setminus \{s\}$
    $color[u] = $ WHITE
    $d[u] = \infty$
    $\pi[u] = $ NIL
$color[s] = $ GRAY
$d[s] = 0$
$\pi[s] = $ NIL
$Q = \emptyset$
ENQUEUE$(Q, s)$
while $Q \neq \emptyset$
    $u = $ DEQUEUE$(Q)$
    for each $v \in Adj[u]$
        if $color[v] == $ WHITE
            $color[v] = $ GRAY
            $d[v] = d[u] + 1$
            $\pi[v] = u$
            ENQUEUE$(Q, v)$
    $color[u] = $ BLACK
```

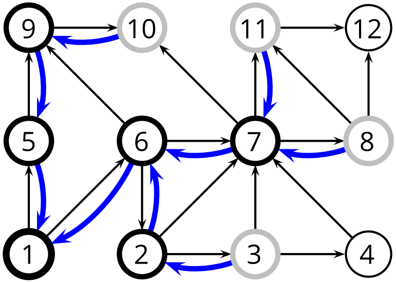


$u = 7$

$Q = \{10, 3, 8, 11\}$

BFS Algorithm

```
BFS($G, s$)
for each vertex $u \in V(G) \setminus \{s\}$
    $color[u] = \text{WHITE}$
    $d[u] = \infty$
    $\pi[u] = \text{NIL}$
$color[s] = \text{GRAY}$
$d[s] = 0$
$\pi[s] = \text{NIL}$
$Q = \emptyset$
ENQUEUE($Q, s$)
while $Q \neq \emptyset$
    $u$ = DEQUEUE($Q$)
    for each $v \in Adj[u]$
        if $color[v] == \text{WHITE}$
            $color[v] = \text{GRAY}$
            $d[v] = d[u] + 1$
            $\pi[v] = u$
            ENQUEUE($Q, v$)
    $color[u] = \text{BLACK}$
```

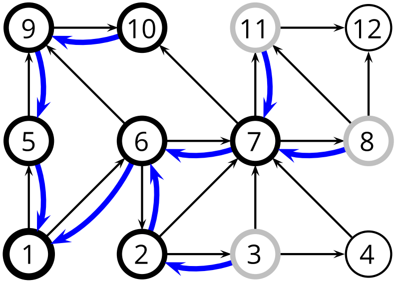


$u = 10$

$Q = \{3, 8, 11\}$

BFS Algorithm

```
BFS($G, s$)
    for each vertex $u \in V(G) \setminus \{s\}$
        $color[u]$ = WHITE
        $d[u] = \infty$
        $\pi[u]$ = NIL
    $color[s]$ = GRAY
    $d[s] = 0$
    $\pi[s]$ = NIL
    $Q = \emptyset$
    ENQUEUE($Q, s$)
    while $Q \neq \emptyset$
        $u$ = DEQUEUE($Q$)
        for each $v \in Adj[u]$
            if $color[v]$ == WHITE
                $color[v]$ = GRAY
                $d[v] = d[u] + 1$
                $\pi[v] = u$
                ENQUEUE($Q, v$)
        $color[u]$ = BLACK
```

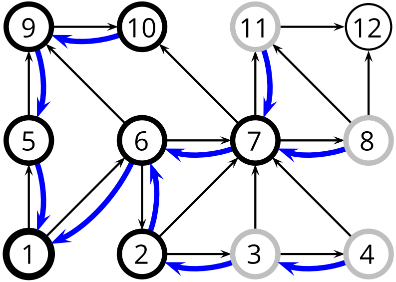


$u = 3$

$Q = \{8, 11\}$

BFS Algorithm

```
BFS(G, s)
for each vertex $u \in V(G) \setminus \{s\}$
    $color[u] = $ WHITE
    $d[u] = \infty$
    $\pi[u] = $ NIL
$color[s] = $ GRAY
$d[s] = 0$
$\pi[s] = $ NIL
$Q = \emptyset$
ENQUEUE$(Q, s)$
while $Q \neq \emptyset$
    $u = $ DEQUEUE$(Q)$
    for each $v \in Adj[u]$
        if $color[v] == $ WHITE
            $color[v] = $ GRAY
            $d[v] = d[u] + 1$
            $\pi[v] = u$
            ENQUEUE$(Q, v)$
    $color[u] = $ BLACK
```

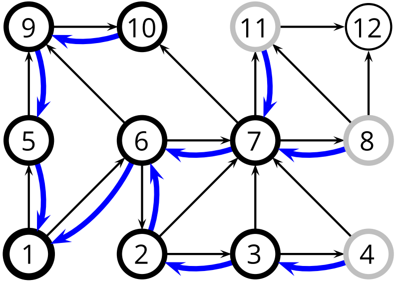


$u = 3$

$Q = \{8, 11, 4\}$

BFS Algorithm

```
BFS($G, s$)
for each vertex $u \in V(G) \setminus \{s\}$
    $color[u]$ = WHITE
    $d[u] = \infty$
    $\pi[u]$ = NIL
$color[s]$ = GRAY
$d[s] = 0$
$\pi[s]$ = NIL
$Q = \emptyset$
ENQUEUE($Q, s$)
while $Q \neq \emptyset$
    $u$ = DEQUEUE($Q$)
    for each $v \in Adj[u]$
        if $color[v]$ == WHITE
            $color[v]$ = GRAY
            $d[v] = d[u] + 1$
            $\pi[v] = u$
            ENQUEUE($Q, v$)
    $color[u]$ = BLACK
```

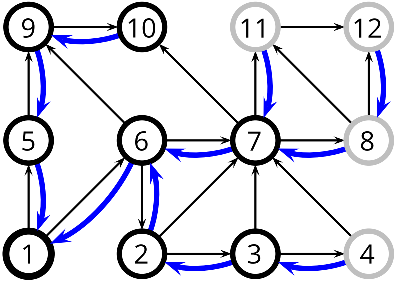


$u = 8$

$Q = \{11, 4\}$

BFS Algorithm

```
BFS($G, s$)
for each vertex $u \in V(G) \setminus \{s\}$
    $color[u]$ = WHITE
    $d[u] = \infty$
    $\pi[u]$ = NIL
$color[s]$ = GRAY
$d[s] = 0$
$\pi[s]$ = NIL
$Q = \emptyset$
ENQUEUE($Q, s$)
while $Q \neq \emptyset$
    $u$ = DEQUEUE($Q$)
    for each $v \in Adj[u]$
        if $color[v]$ == WHITE
            $color[v]$ = GRAY
            $d[v] = d[u] + 1$
            $\pi[v] = u$
            ENQUEUE($Q, v$)
    $color[u]$ = BLACK
```

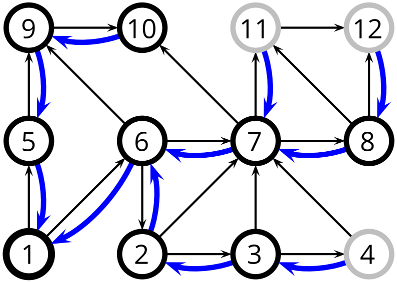


$u = 8$

$Q = \{11, 4, 12\}$

BFS Algorithm

```
BFS($G, s$)
for each vertex $u \in V(G) \setminus \{s\}$
    $color[u]$ = WHITE
    $d[u] = \infty$
    $\pi[u]$ = NIL
$color[s]$ = GRAY
$d[s] = 0$
$\pi[s]$ = NIL
$Q = \emptyset$
ENQUEUE($Q, s$)
while $Q \neq \emptyset$
    $u$ = DEQUEUE($Q$)
    for each $v \in Adj[u]$
        if $color[v]$ == WHITE
            $color[v]$ = GRAY
            $d[v] = d[u] + 1$
            $\pi[v] = u$
            ENQUEUE($Q, v$)
    $color[u]$ = BLACK
```

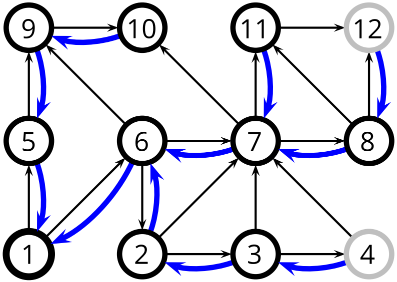


$u = 11$

$Q = \{4, 12\}$

BFS Algorithm

```
BFS($G, s$)
    for each vertex $u \in V(G) \setminus \{s\}$
        $color[u] = $ WHITE
        $d[u] = \infty$
        $\pi[u] = $ NIL
    $color[s] = $ GRAY
    $d[s] = 0$
    $\pi[s] = $ NIL
    $Q = \emptyset$
    ENQUEUE($Q, s$)
    while $Q \neq \emptyset$
        $u = $ DEQUEUE($Q$)
        for each $v \in Adj[u]$
            if $color[v] == $ WHITE
                $color[v] = $ GRAY
                $d[v] = d[u] + 1$
                $\pi[v] = u$
                ENQUEUE($Q, v$)
        $color[u] = $ BLACK
```

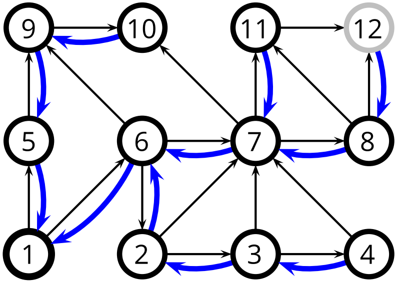


$u = 4$

$Q = \{12\}$

BFS Algorithm

```
BFS($G, s$)
for each vertex $u \in V(G) \setminus \{s\}$
        $color[u] = \text{WHITE}$
        $d[u] = \infty$
        $\pi[u] = \text{NIL}$
$color[s] = \text{GRAY}$
$d[s] = 0$
$\pi[s] = \text{NIL}$
$Q = \emptyset$
ENQUEUE($Q, s$)
while $Q \neq \emptyset$
        $u = $ DEQUEUE($Q$)
        for each $v \in Adj[u]$
                if $color[v] == \text{WHITE}$
                        $color[v] = \text{GRAY}$
                        $d[v] = d[u] + 1$
                        $\pi[v] = u$
                        ENQUEUE($Q, v$)
        $color[u] = \text{BLACK}$
```

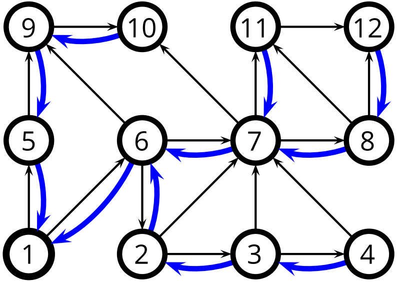


$u = 12$

$Q = \emptyset$

BFS Algorithm

```
BFS(G, s)
for each vertex $u \in V(G) \setminus \{s\}$
    $color[u] = $ WHITE
    $d[u] = \infty$
    $\pi[u] = $ NIL
$color[s] = $ GRAY
$d[s] = 0$
$\pi[s] = $ NIL
$Q = \emptyset$
ENQUEUE$(Q, s)$
while $Q \neq \emptyset$
    $u = $ DEQUEUE$(Q)$
    for each $v \in Adj[u]$
        if $color[v] ==$ WHITE
            $color[v] = $ GRAY
            $d[v] = d[u] + 1$
            $\pi[v] = u$
            ENQUEUE$(Q, v)$
    $color[u] = $ BLACK
```



Complexity of BFS

```
BFS($G, s$)
for each vertex $u \in V(G) \setminus \{s\}$
    $color[u] = $ WHITE
    $d[u] = \infty$
    $\pi[u] = $ NIL
$color[s] = $ GRAY
$d[s] = 0$
$\pi[s] = $ NIL
$Q = \emptyset$
ENQUEUE($Q, s$)
while $Q \neq \emptyset$
    $u = $ DEQUEUE($Q$)
    for each $v \in Adj[u]$
        if $color[v] == $ WHITE
            $color[v] = $ GRAY
            $d[v] = d[u] + 1$
            $\pi[v] = u$
            ENQUEUE($Q, v$)
    $color[u] = $ BLACK
```

Complexity of BFS

```
BFS($G, s$)
for each vertex $u \in V(G) \setminus \{s\}$
    $color[u] = $ WHITE
    $d[u] = \infty$
    $\pi[u] = $ NIL
$color[s] = $ GRAY
$d[s] = 0$
$\pi[s] = $ NIL
$Q = \emptyset$
ENQUEUE($Q, s$)
while $Q \neq \emptyset$
    $u = $ DEQUEUE($Q$)
    for each $v \in Adj[u]$
        if $color[v] == $ WHITE
            $color[v] = $ GRAY
            $d[v] = d[u] + 1$
            $\pi[v] = u$
            ENQUEUE($Q, v$)
    $color[u] = $ BLACK
```

- We enqueue a vertex only if it is white, and we immediately color it gray; thus, we enqueue every vertex *at most once*

Complexity of BFS

```
BFS($G, s$)
for each vertex $u \in V(G) \setminus \{s\}$
    $color[u] = \text{WHITE}$
    $d[u] = \infty$
    $\pi[u] = \text{NIL}$
$color[s] = \text{GRAY}$
$d[s] = 0$
$\pi[s] = \text{NIL}$
$Q = \emptyset$
ENQUEUE($Q, s$)
while $Q \neq \emptyset$
    $u = $ DEQUEUE($Q$)
    for each $v \in Adj[u]$
        if $color[v] == \text{WHITE}$
            $color[v] = \text{GRAY}$
            $d[v] = d[u] + 1$
            $\pi[v] = u$
            ENQUEUE($Q, v$)
    $color[u] = \text{BLACK}$
```

- We enqueue a vertex only if it is white, and we immediately color it gray; thus, we enqueue every vertex *at most once*
- So, the (dequeue) while loop executes $O(|V|)$ times

Complexity of BFS

```
BFS($G, s$)
for each vertex $u \in V(G) \setminus \{s\}$
    $color[u] = \text{WHITE}$
    $d[u] = \infty$
    $\pi[u] = \text{NIL}$
$color[s] = \text{GRAY}$
$d[s] = 0$
$\pi[s] = \text{NIL}$
$Q = \emptyset$
ENQUEUE($Q, s$)
while $Q \neq \emptyset$
    $u$ = DEQUEUE($Q$)
    for each $v \in Adj[u]$
        if $color[v] == \text{WHITE}$
            $color[v] = \text{GRAY}$
            $d[v] = d[u] + 1$
            $\pi[v] = u$
            ENQUEUE($Q, v$)
    $color[u] = \text{BLACK}$
```

- We enqueue a vertex only if it is white, and we immediately color it gray; thus, we enqueue every vertex *at most once*
- So, the (dequeue) while loop executes $O(|V|)$ times
- For each vertex u, the inner loop executes $\Theta(|E_u|)$, for a total of $O(|E|)$ steps

Complexity of BFS

```
BFS($G, s$)
for each vertex $u \in V(G) \setminus \{s\}$
    $color[u] =$ WHITE
    $d[u] = \infty$
    $\pi[u] =$ NIL
$color[s] =$ GRAY
$d[s] = 0$
$\pi[s] =$ NIL
$Q = \emptyset$
ENQUEUE($Q, s$)
while $Q \neq \emptyset$
    $u =$ DEQUEUE($Q$)
    for each $v \in Adj[u]$
        if $color[v] ==$ WHITE
            $color[v] =$ GRAY
            $d[v] = d[u] + 1$
            $\pi[v] = u$
            ENQUEUE($Q, v$)
    $color[u] =$ BLACK
```

- We enqueue a vertex only if it is white, and we immediately color it gray; thus, we enqueue every vertex *at most once*
- So, the (dequeue) while loop executes $O(|V|)$ times
- For each vertex u, the inner loop executes $\Theta(|E_u|)$, for a total of $O(|E|)$ steps
- So, $O(|V| + |E|)$

Depth-First Search

Depth-First Search

- Immediately follow the links of the most recently-visited vertex, then backtrack when you reach a dead-end
  - i.e., backtrack when the current vertex has no more adjacent vertices that have not yet been visited

Depth-First Search

- Immediately follow the links of the most recently-visited vertex, then backtrack when you reach a dead-end
    - i.e., backtrack when the current vertex has no more adjacent vertices that have not yet been visited

- *Input:* $G = (V, E)$
    - explores the graph, touching *all vertices*

Depth-First Search

- Immediately follow the links of the most recently-visited vertex, then backtrack when you reach a dead-end
    - i.e., backtrack when the current vertex has no more adjacent vertices that have not yet been visited

- *Input:* $G = (V, E)$
    - explores the graph, touching *all vertices*
    - produces a ***depth-first forest***, consisting of all the ***depth-first trees*** defined by the DFS exploration

Depth-First Search

- Immediately follow the links of the most recently-visited vertex, then backtrack when you reach a dead-end
  - i.e., backtrack when the current vertex has no more adjacent vertices that have not yet been visited

- *Input:* $G = (V, E)$
  - explores the graph, touching *all vertices*
  - produces a ***depth-first forest***, consisting of all the ***depth-first trees*** defined by the DFS exploration
  - associates ***two time-stamps*** to each vertex
    - $d[u]$ records when u is first discovered
    - $f[u]$ records when DFS finishes examining u's edges, and therefore backtracks from u

DFS Algorithm

```
DFS($G$)
    for each vertex $u \in V(G)$
        $color[u] = \text{WHITE}$
        $\pi[u] = \text{NIL}$
    $time = 0$ // “global” variable
    for each vertex $u \in V(G)$
        if $color[u] == \text{WHITE}$
            DFS-VISIT($u$)
```

```
DFS-VISIT($u$)
    $color[u] = \text{GREY}$
    $time = time + 1$
    $d[u] = time$
    for each $v \in Adj[u]$
        if $color[v] == \text{WHITE}$
            $\pi[v] = u$
            DFS-VISIT($v$)
    $color[u] = \text{BLACK}$
    $time = time + 1$
    $f[u] = time$
```

Complexity of DFS

Complexity of DFS

- The loop in **DFS-VISIT**(u) (lines 4–7) accounts for $\Theta(|E_u|)$

Complexity of DFS

- The loop in **DFS-VISIT**(u) (lines 4–7) accounts for $\Theta(|E_u|)$
- We call **DFS-VISIT**(u) *once* for each vertex u
  - either in **DFS**, or recursively in **DFS-VISIT**
  - because we call it only if $color[u] = \text{WHITE}$, but then we immediately set $color[u] = \text{GREY}$

Complexity of DFS

- The loop in **DFS-VISIT**(u) (lines 4–7) accounts for $\Theta(|E_u|)$

- We call **DFS-VISIT**(u) *once* for each vertex u
  - either in **DFS**, or recursively in **DFS-VISIT**
  - because we call it only if $color[u] = \text{WHITE}$, but then we immediately set $color[u] = \text{GREY}$

- So, the overall complexity is $\Theta(|V| + |E|)$

Applications of DFS: Topological Sort

Applications of DFS: Topological Sort

- **Problem:** (topological sort)

  Given a *directed acyclic graph* (DAG)

  - find an ordering of vertices such that you only end up with *forward links*

Applications of DFS: Topological Sort

- **Problem:** (topological sort)

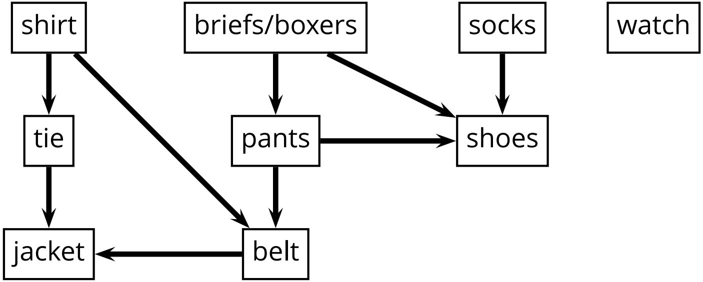
  Given a *directed acyclic graph* (DAG)

  - find an ordering of vertices such that you only end up with *forward links*

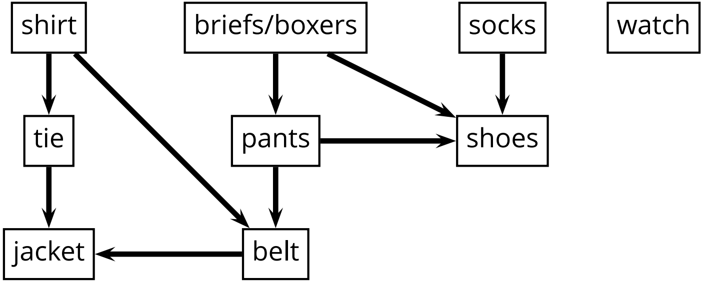
- **Example:** dependencies in software packages
  - find an installation order for a set of software packages
  - such that every package is installed only after all the packages it depends on

Topological Sort Algorithm

Topological Sort Algorithm



Topological Sort Algorithm



```
TOPOLOGICAL-SORT$(G)$
1    DFS$(G)$
2    output $V$ sorted in reverse order of $f[\cdot]$
```