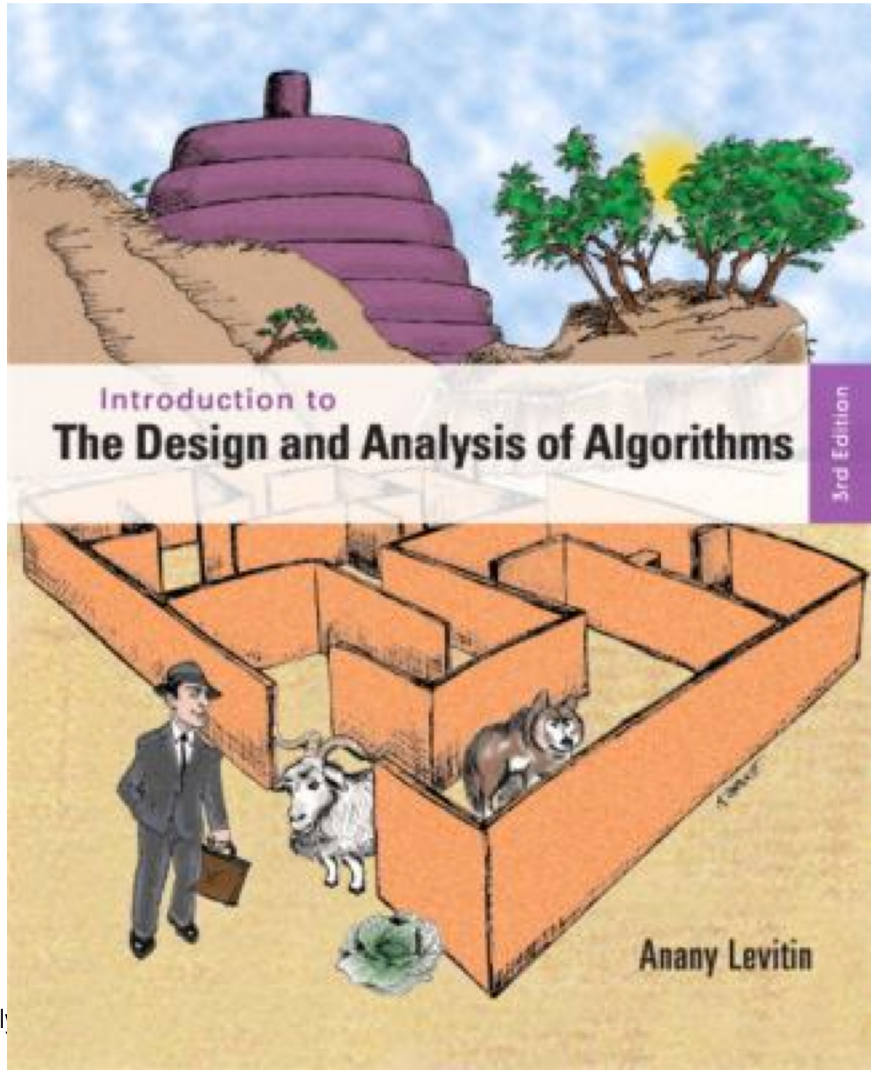
Chapter 3

Brute Force and Exhaustive Search



Brute Force

A straightforward approach, usually based directly on the problem's statement and definitions of the concepts involved

Examples:

1. Computing a^n ($a > 0$, n a nonnegative integer)
2. Computing $n!$
3. Multiplying two matrices
4. Searching for a key of a given value in a list

An Important Algorithm Design Strategy

- Applicable to a very wide variety of problems
- Yields reasonable algorithms for some important problems

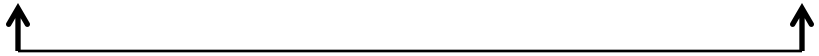
- Expense of designing a more efficient algorithm may be unjustifiable
- Useful for solving small-sized instances of a problem, though inefficient in general
- Can serve as a yardstick with which to judge more efficient alternatives

Brute-Force Sorting Algorithm

Selection Sort Scan the array to find its smallest element and swap it with the first element. Then, starting with the second element, scan the elements to the right of it to find the smallest among them and swap it with the second elements. Generally, on pass i ($0 \le i \le n-2$), find the smallest element in $A[i..n-1]$ and swap it with $A[i]$:

$A[0] \le \ldots \le A[i-1] \mid A[i], \ldots, A[min], \ldots, A[n-1]$

in their final positions



Example: 7 3 2 5

Selection Sort

89	45	68	90	29	34	**17**
17	| 45	68	90	**29**	34	89
17	29	| 68	90	45	**34**	89
17	29	34	| 90	**45**	68	89
17	29	34	45	| 90	**68**	89
17	29	34	45	68	| 90	**89**
17	29	34	45	68	89	| 90

FIGURE 3.1 Example of sorting with selection sort. Each line corresponds to one iteration of the algorithm, i.e., a pass through the list tail to the right of the vertical bar; an element in bold indicates the smallest element found. Elements to the left of the vertical bar are in their final positions and are not considered in this and subsequent iterations.

Selection Sort

ALGORITHM *SelectionSort*$(A[0..n-1])$

```
//Sorts a given array by selection sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in ascending order
for i ← 0 to n − 2 do
    min ← i
    for j ← i + 1 to n − 1 do
        if A[j] < A[min]   min ← j
    swap A[i] and A[min]
```

Analysis of Selection Sort

- Input size: number of elements n
- Basic operation: *key comparison*
  - Number of times it is executed depends only on array's size

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2}$$

- Complexity: $\Theta(n^2)$
- What can we say about the number of key swaps???
  - Note: Number of key swaps is in $\Theta(n)$
  - Positively distinguishes selection sort from other sorting algorithms

Bubble Sort

- Compare adjacent elements and swap them if they are out of order

Bubble Sort

ALGORITHM *BubbleSort*$(A[0..n-1])$

//Sorts a given array by bubble sort

//Input: An array $A[0..n-1]$ of orderable elements

//Output: Array $A[0..n-1]$ sorted in ascending order

for $i \leftarrow 0$ **to** $n-2$ **do**

  for $j \leftarrow 0$ **to** $n-2-i$ **do**

    if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

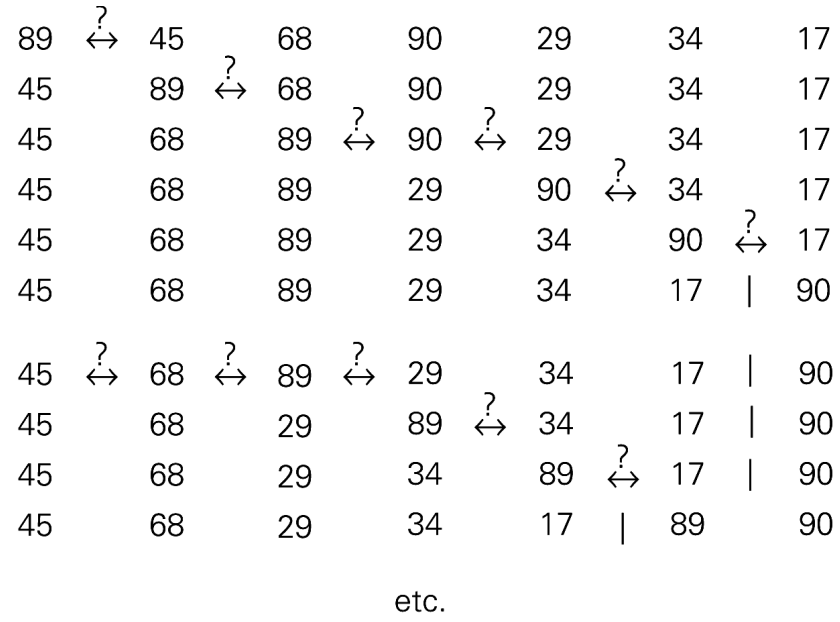


FIGURE 3.2 First two passes of bubble sort on the list 89, 45, 68, 90, 29, 34, 17. A new line is shown after a swap of two elements is done. The elements to the right of the vertical bar are in their final positions and are not considered in subsequent iterations of the algorithm.

Analysis of Bubble Sort

- Input size: number of elements *n*
- Basic operation: *key comparison*
  - Number of times it is executed depends only on array's size

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2} \in \Theta(n^2)$$

- Number of key swaps depends on the input
  - Same as number of key comparisons for the worst case of decreasing arrays

$$S_{\text{worst}}(n) = C(n) = \frac{(n-1)n}{2} \in \Theta(n^2)$$

Sequential Search

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element in A that matches K
//         or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

- Compare successive elements of list with a given search key until there's a match (success) or list is exhausted without a match (unsuccessful)

Sequential Search

ALGORITHM *SequentialSearch2*$(A[0..n], K)$

```
//Implements sequential search with a search key as a sentinel
//Input: An array A of n elements and a search key K
//Output: The index of the first element in A[0..n − 1] whose value is
//        equal to K or −1 if no such element is found
A[n] ← K
i ← 0
while A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

- Improvement
  - Trick: Append search key to end of list
    - Eliminates check for list's end on each iteration
  - If list is known to be sorted, search can stop as soon as an element greater than or equal to search key is encountered
- Complexity: $\Theta(n)$ in both worst and average cases

Brute-Force String Matching

- *Pattern*: a string of *m* characters to search for
- *Text*: a (longer) string of *n* characters to search in
- Problem: find a substring in the text that matches the pattern

Brute-force algorithm

Step 1 Align pattern at beginning of text

Step 2 Moving from left to right, compare each character of pattern to the corresponding character in text until
- all characters are found to match (successful search); or
- a mismatch is detected

Step 3 While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

An Example

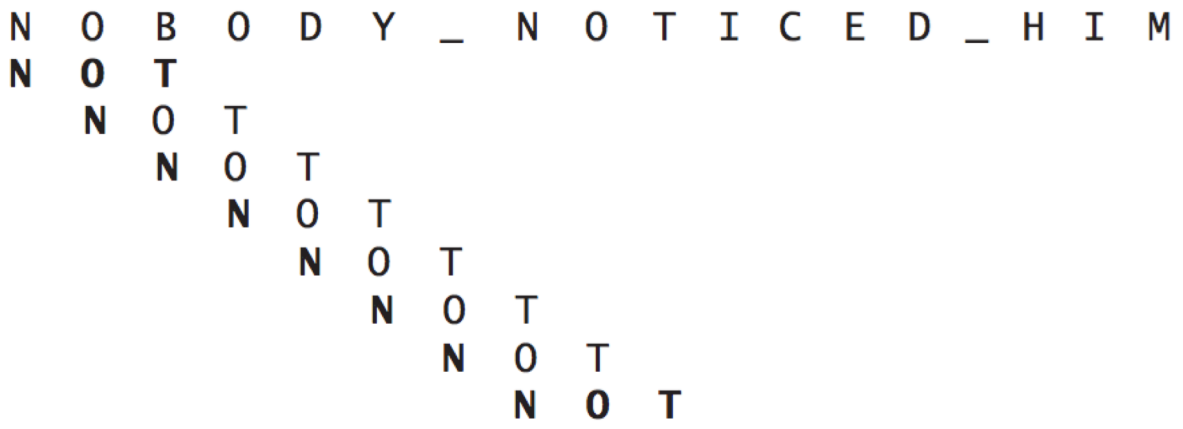


FIGURE 3.3 Example of brute-force string matching. The pattern's characters that are compared with their text counterparts are in bold type.

Pseudocode and Efficiency

ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

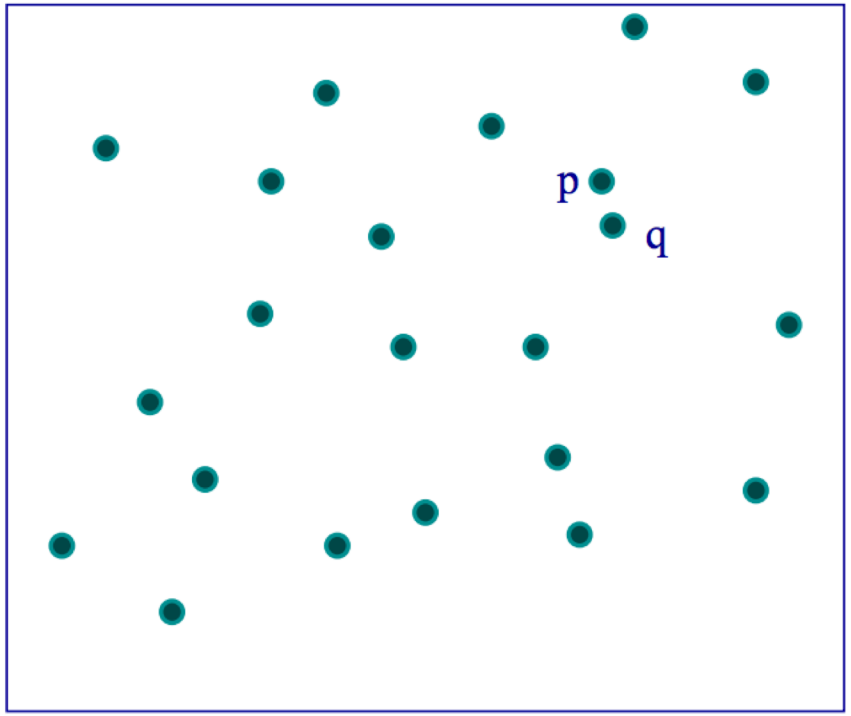
```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//       an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//        matching substring or −1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return −1
```

Efficiency:

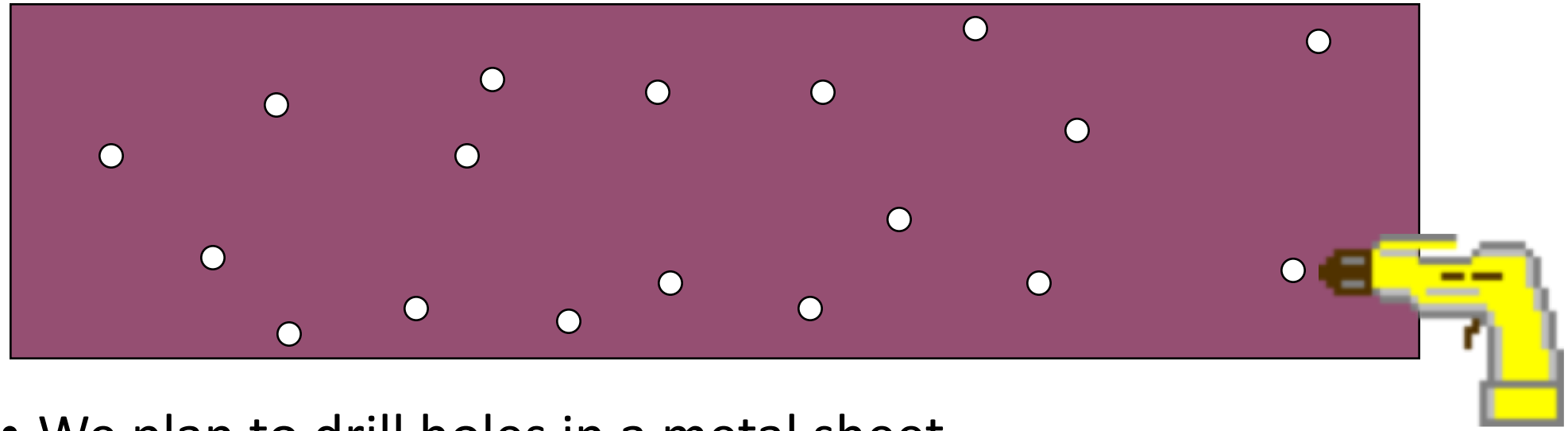
- Worst case: make all *m* comparisons before shifting the pattern
  - This can happen for each of the *n*-*m*+1 tries → Θ(*nm*)

Closest-Pair Problem

- Given n points in d-dimensions, find two whose mutual distance is smallest.
- Fundamental problem in many applications as well as a key step

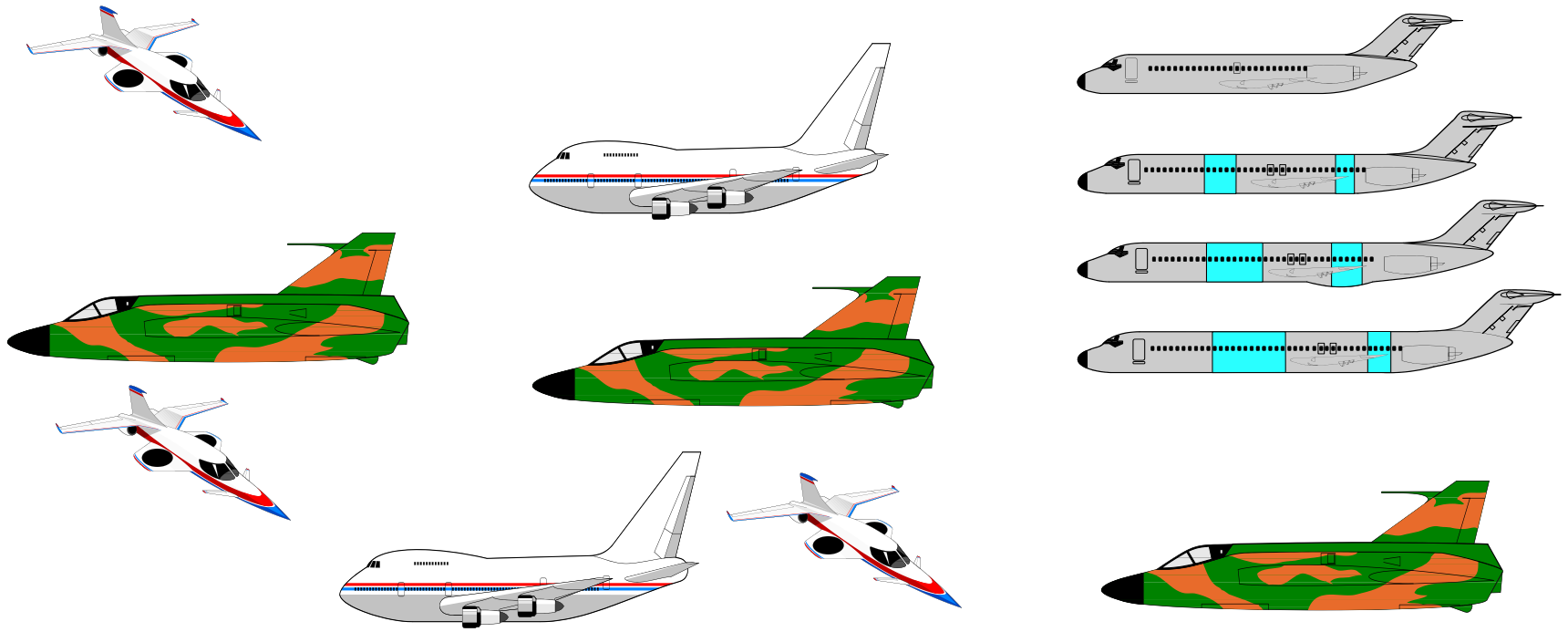


Applications



- We plan to drill holes in a metal sheet.
- If the holes are too close, the sheet will tear during drilling.
- Verify that no two holes are closer than a threshold distance (e.g., holes are at least 1 inch apart).

Air Traffic Control



- 3D -- Locations of airplanes flying in the neighborhood of a busy airport are known.
- Want to be sure that no two planes get closer than a given threshold distance.

2D Closest-Pair Problem

How to find the two closest points in a set of *n* points (in the two-dimensional Cartesian plane)?

Brute-force algorithm

- Compute the distance between every pair of distinct points
- and return the indexes of the points for which the distance is the smallest.

Closest-Pair Brute-Force Algorithm (cont.)

ALGORITHM *BruteForceClosestPoints*(P)

//Input: A list P of n $(n \geq 2)$ points $P_1 = (x_1, y_1), \ldots, P_n = (x_n, y_n)$
//Output: Indices *index*1 and *index*2 of the closest pair of points
$dmin \leftarrow \infty$
for $i \leftarrow 1$ **to** $n - 1$ **do**
    for $j \leftarrow i + 1$ **to** n **do**
        $d \leftarrow sqrt((x_i - x_j)^2 + (y_i - y_j)^2)$ //*sqrt* is the square root function
        if $d < dmin$
            $dmin \leftarrow d$; $index1 \leftarrow i$; $index2 \leftarrow j$
return *index*1, *index*2

Analysis of Closest-Pair Algorithm

- Basic op: *computing the Euclidean distance between two points*

- How to make it faster? Note – *sqrt function is strictly increasing*
  - Basic operation now: *squaring a number*

- Efficiency:

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2 = 2\sum_{i=1}^{n-1} (n-i) = 2\left[(n-1)+(n-2)+\ldots+1\right] = (n-1)n \in \Theta(n^2)$$

Review homework 1 - 1.1.5

- Design an algorithm to find all the common elements in two sorted lists of numbers. For example, for the lists 2, 5, 5, 5 and 2, 2, 3, 5, 5, 7, the output should be 2, 5, 5. What is the maximum number of comparisons your make?

Solution

```
Let a and b be sorted arrays
Let i = 0 and j = 0
While i < length (a) and j < length (b)
        If a[i] = b[j]
                Print a[i]
                i++
                j++
        Else if a[i] > b[j]
                j++
        else
                i++
```

1.2.9

- Consider the following algorithm for finding the distance between the two closest elements in an array of numbers.

ALGORITHM *MinDistance*$(A[0..n-1])$
//Input: Array $A[0..n-1]$ of numbers
//Output: Minimum distance between two of its elements
$dmin \leftarrow \infty$
for $i \leftarrow 0$ **to** $n-1$ **do**
for $j \leftarrow 0$ **to** $n-1$ **do**
if $i \neq j$ **and** $|A[i]-A[j]| < dmin$
$dmin \leftarrow |A[i]-A[j]|$
return $dmin$

- Make as many improvements as you can in this algorithmic solution to the problem. If you need to, you may change the algorithm altogether; if not, improve the implementation given.

Solution

Algorithm *MinDistance2*$(A[0..n-1])$
//Input: An array $A[0..n-1]$ of numbers
//Output: The minimum distance d between two of its elements

$dmin \leftarrow \infty$
for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        $temp \leftarrow |A[i] - A[j]|$
        if $temp < dmin$
            $dmin \leftarrow temp$
return $dmin$

1.3.1

- Consider the algorithm for the sorting problem that sorts an array by counting, for each of its elements, the number of smaller elements and then uses this information to put the element in its appropriate position in the sorted array:
  a) Apply this algorithm to sorting the list 60, 35, 81, 98, 14, 47.
  b) Is this algorithm stable?
  c) Is it in-place?

ALGORITHM *ComparisonCountingSort*$(A[0..n-1])$

```
//Sorts an array by comparison counting
//Input: Array A[0..n − 1] of orderable values
//Output: Array S[0..n − 1] of A's elements sorted
//    in nondecreasing order
for i ← 0 to n − 1 do
    Count[i] ← 0
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] < A[j]
            Count[j] ← Count[j] + 1
        else Count[i] ← Count[i] + 1
for i ← 0 to n − 1 do
    S[Count[i]] ← A[i]
return S
```

Quiz 1: Consider the following recursive algorithm

ALGORITHM $Q(n)$

//Input: A positive integer n
if $n = 1$ **return** 1
else return $Q(n-1) + 2 * n - 1$

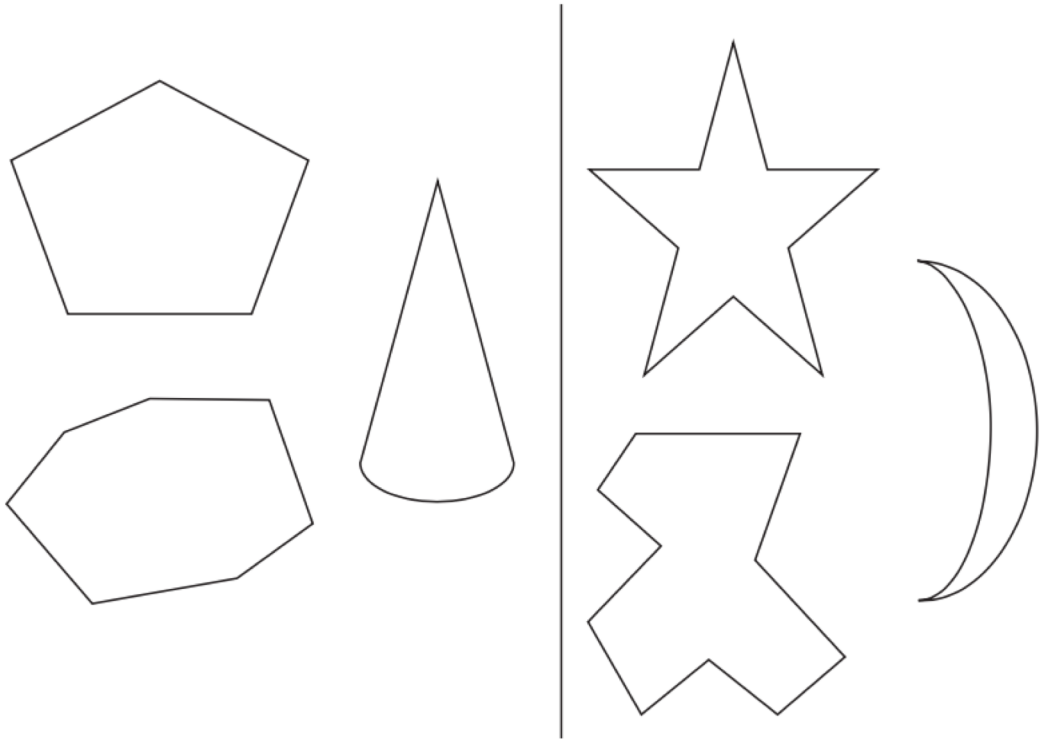
a) Set up a recurrence relation to count the basic operation? (3 points)

b) What is the efficiency class of this algorithm? (2 points)

Convex Hull Problem

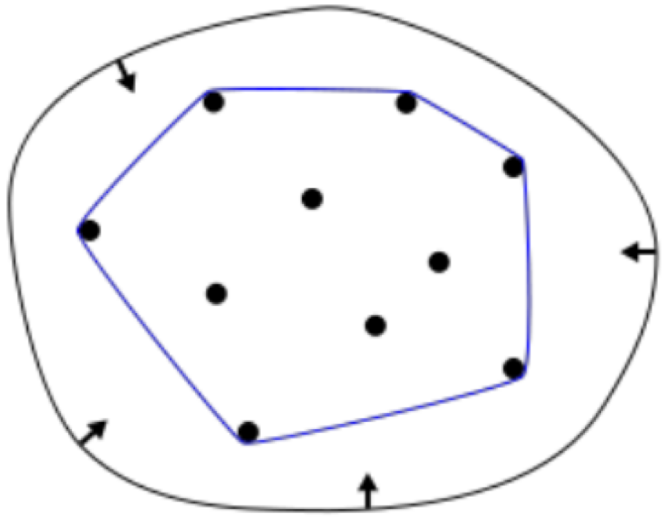
- A set of points in the plane is called ***convex*** if for any two points p and q in the set, the entire line segment with the endpoints at p and q belongs to the set.

Which sets below are convex?



Convex hull

- The ***convex hull*** of a set S of points is the smallest convex set containing S. (The "smallest" requirement means that the convex hull of S must be a subset of any convex set containing S.)



[Wikipedia]

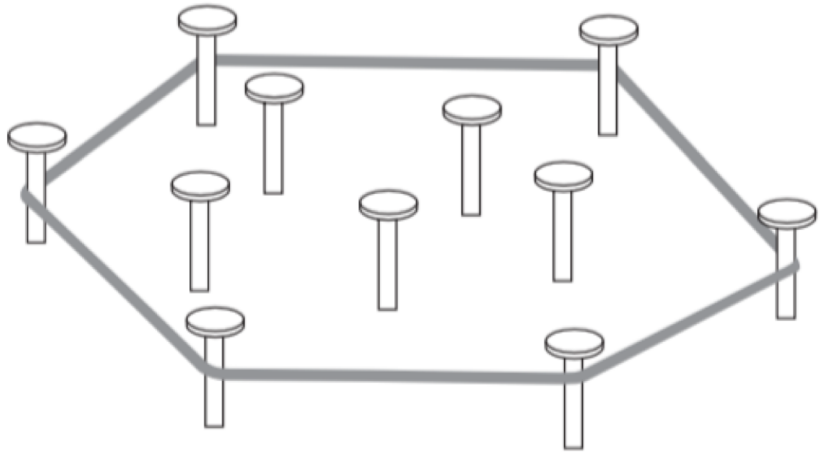


FIGURE 3.5 Rubber-band interpretation of the convex hull.

What is convex hull for the following set S?

- S is a set of two points
- S is a set of three points not on the same line
- S is a set of > 3 points

Convex Hull Problem

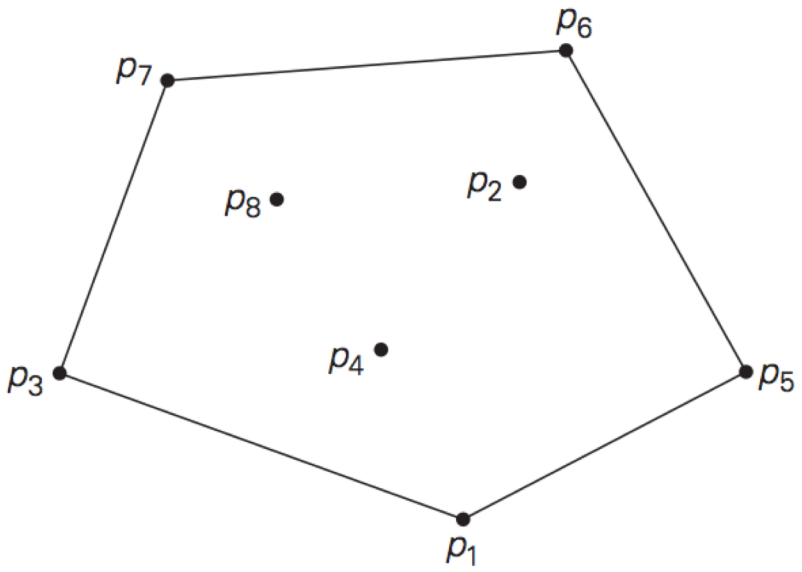
- The ***convex-hull problem*** is the problem of constructing the convex hull for a given set S of n points.

- One of the most important problems in computational geometry. The problem of finding convex hulls finds its practical applications in pattern recognition, image processing, statistics, geographic information system, game theory, construction of phase diagrams, and static code analysis by abstract interpretation (from Wikipedia)

- In computer animation, replacing objects by their convex hulls speeds up collision detection;

Use Brute Force to Solve Convex-Hull Problem

- How to construct the convex hull for a given set S of n points?
- Observation: a line segment connecting two points p_i and p_j of a set of n points is a part of the convex hull's boundary if and only if all the other points of the set lie on the same side of the straight line through these two points.



Brute Force (cont.)

- The straight line through two points (x_1, y_1), (x_2, y_2) in the coordinate plane can be defined by the equation

$$ax + by = c,$$

- Such a line divides the plane into two half-planes: for all the points in one of them, ax+by >c, while for all the points in the other, ax+by <c.

Brute Force (cont.)

- Thus, to check whether certain points lie on the same side of the line, we can simply check whether the expression ax + by − c has the same sign for each of these points.

- Implementation of algorithm?

Algorithm

The 2D Convex-Hull Problem

for i ← 1 to n – 1 do
    for j ← i + 1 to n do
        L ← line through P_i and P_j
        flag ← do all other points lie on the same side of L ?
        if (flag)
        then add P_i and P_j to the boundary

Efficiency of the Brute Force Algorithm

- For each of n(n − 1)/2 pairs of distinct points, we may need to find the sign of ax + by − c for each of the other n − 2 points.

- Time efficiency:
  - worst case: $\Theta(n^3)$

Brute-Force Strengths and Weaknesses

- Strengths
  - wide applicability
  - simplicity
  - yields reasonable algorithms for some important problems (e.g., matrix multiplication, sorting, searching, string matching)

- Weaknesses
  - rarely yields efficient algorithms
  - some brute-force algorithms are unacceptably slow
  - not as constructive as some other design techniques

Exhaustive Search

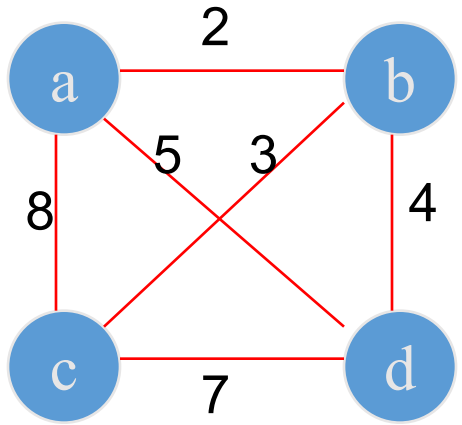
A brute force solution to a problem involving search for an element with a special property, usually among combinatorial objects such as permutations, combinations, or subsets of a set.

Method:

- generate a list of all potential solutions to the problem in a systematic manner (we'll see such algorithms in Sec. 4.3)
- evaluate potential solutions one by one, disqualifying infeasible ones and, for an optimization problem, keeping track of the best one found so far
- when search ends, announce the solution(s) found

Example 1: Traveling Salesman Problem

- Given *n* cities with known distances between each pair, find the shortest tour that passes through all the cities exactly once before returning to the starting city
- Alternatively: Find shortest *Hamiltonian circuit* in a weighted connected graph
- Example:



TSP by Exhaustive Search

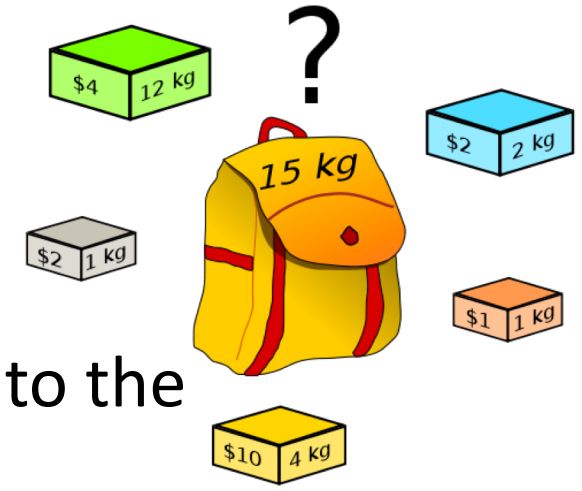
Tour	Cost
a→b→c→d→a	2+3+7+5 = 17
a→b→d→c→a	2+4+7+8 = 21
a→c→b→d→a	8+3+4+5 = 20
a→c→d→b→a	8+7+4+2 = 21
a→d→b→c→a	5+4+3+8 = 20
a→d→c→b→a	5+7+3+2 = 17

Efficiency: Θ(n!)

Example 2: Knapsack Problem

Given *n* items:

- weights: w_1 w_2 … w_n
- values: v_1 v_2 … v_n
- a knapsack of capacity *W*



Find most valuable subset of the items that fit into the knapsack

Example: Knapsack capacity W=10

item	weight	value
1	7	$42
2	3	$12
3	4	$40
4	5	$25

Knapsack Problem by Exhaustive Search

Subset	Total weight	Total value
∅	0	$ 0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$54
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

Efficiency: ???

NP-Hard Problems

- Both TSP and Knapsack problems lead to algorithms that are extremely inefficient
- They are the best known examples of so-called NP-Hard problems
- No polynomial-time algorithm is known for any NP-Hard problem

Recaps

- Brute force
  - Closest-Pair Problem
  - Convex Hull Problem

- Exhaustive search
  - Traveling Salesman Problem
  - Knapsack Problem

Example 3: The Assignment Problem

- **Definition**: There are *n* people who need to be assigned to *n* jobs, one person per job. The cost of assigning person *i* to job *j* is C[*i*,*j*]. Find an assignment that minimizes the total cost. Here, C is called cost matrix.

Example:

	Job 1	**Job 2**	**Job 3**	**Job 4**
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

- Describe feasible solutions to the assignment problem as n-tuples $\langle j_1, \ldots, j_n \rangle$ in which the ith component, i = 1, . . . , n, indicates the column of the element selected in the ith row (i.e., the job number assigned to the ith person).

- For example, for the cost matrix above, $\langle 2, 3, 4, 1 \rangle$ indicates the assignment of Person 1 to Job 2, Person 2 to Job 3, Person 3 to Job 4, and Person 4 to Job 1.

Example 3: The Assignment Problem

Algorithmic Plan: Generate all legitimate assignments, compute their costs, and select the cheapest one.

Pose the problem as the one about a cost matrix: Generate all the permutations of integers 1, 2, …, n, computing the total cost of each assignment by summing up the corresponding elements of the cost matrix, and finally selecting the one with the smallest sum.

What is the number of permutations???

Assignment Problem by Exhaustive Search

$$C = \begin{matrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{matrix}$$

Assignment (col.#s)	Total Cost
1, 2, 3, 4	9+4+1+4=18
1, 2, 4, 3	9+4+8+9=30
1, 3, 2, 4	9+3+8+4=24
1, 3, 4, 2	9+3+8+6=26
1, 4, 2, 3	9+7+8+9=33
1, 4, 3, 2	9+7+1+6=23
etc.	

Comments on Exhaustive Search

- Exhaustive-search algorithms run in a realistic amount of time only on very small instances

- In some cases, there are much better alternatives!
  - Euler circuits
  - shortest paths
  - minimum spanning tree
  - assignment problem

- In many cases, exhaustive search or its variation is the only known way to get exact solution

Graph Traversal Algorithms

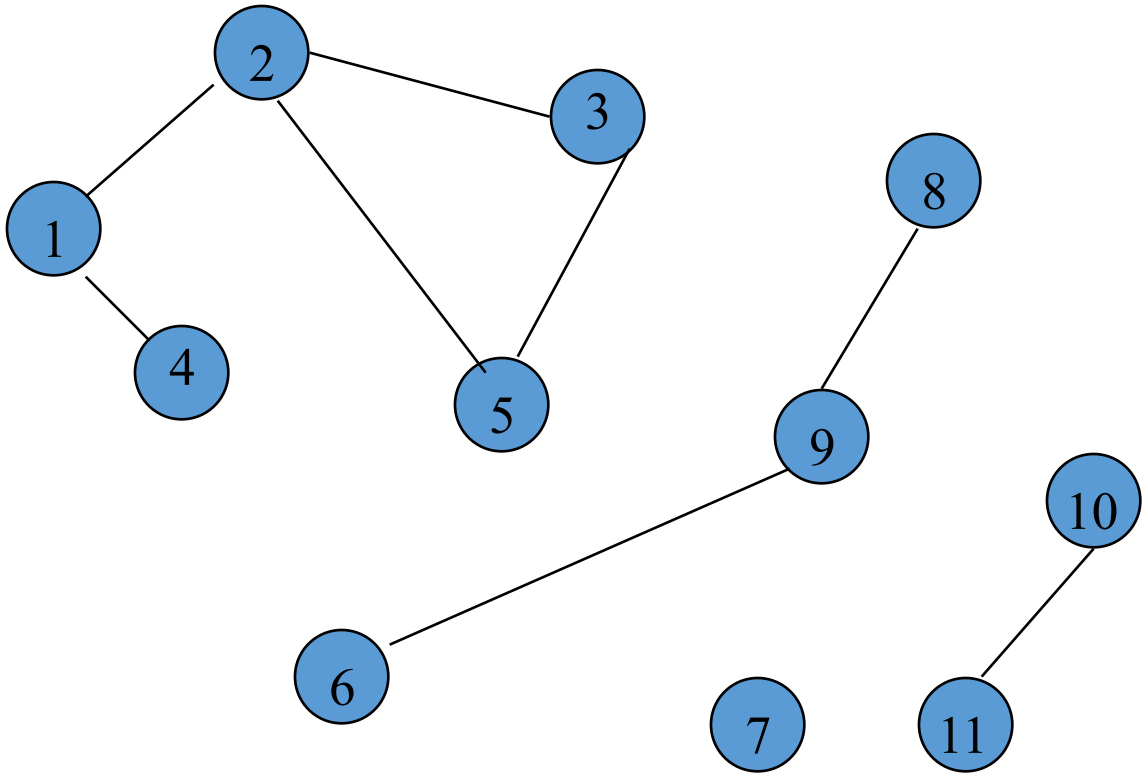
Many problems require processing all graph vertices (and edges) in systematic fashion

Graph traversal algorithms:

- Breadth-first search (BFS)
- Depth-first search (DFS)

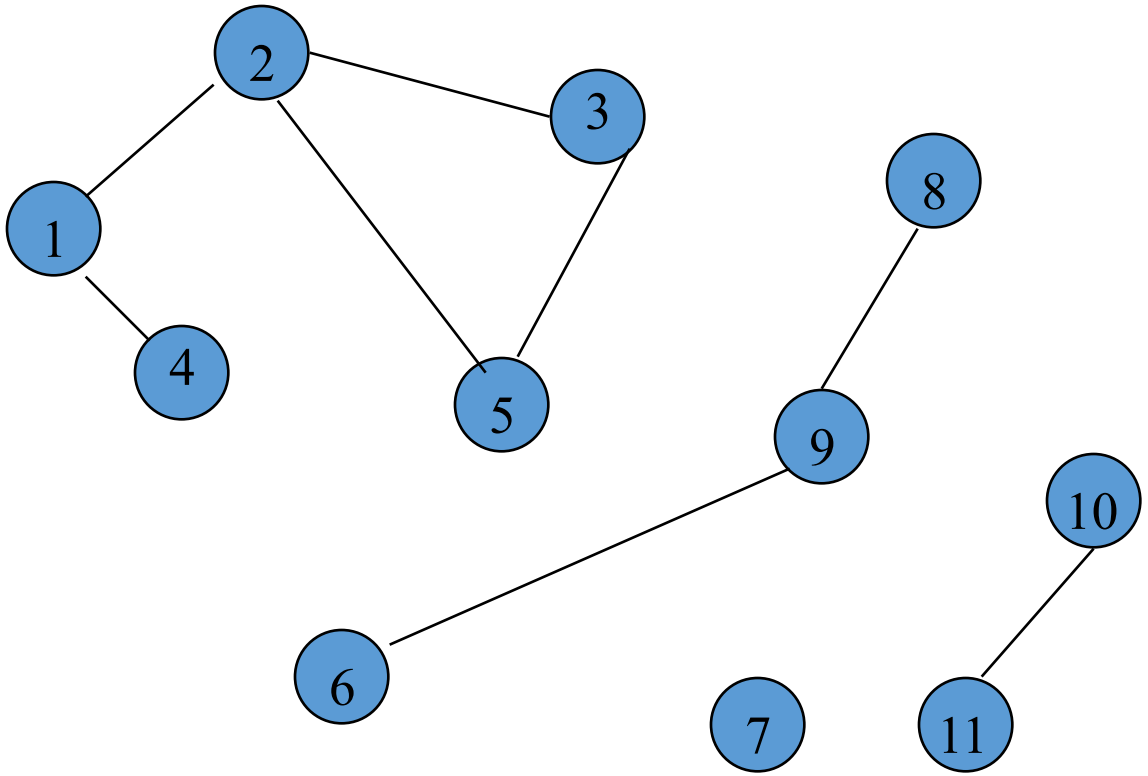
Graph Search Methods

- A vertex *u* is **reachable** from vertex *v* iff there is a path from *v* to *u*.



Graph Search Methods

- A search method starts at a given vertex *v* and visits/labels/marks every vertex that is reachable from *v*.



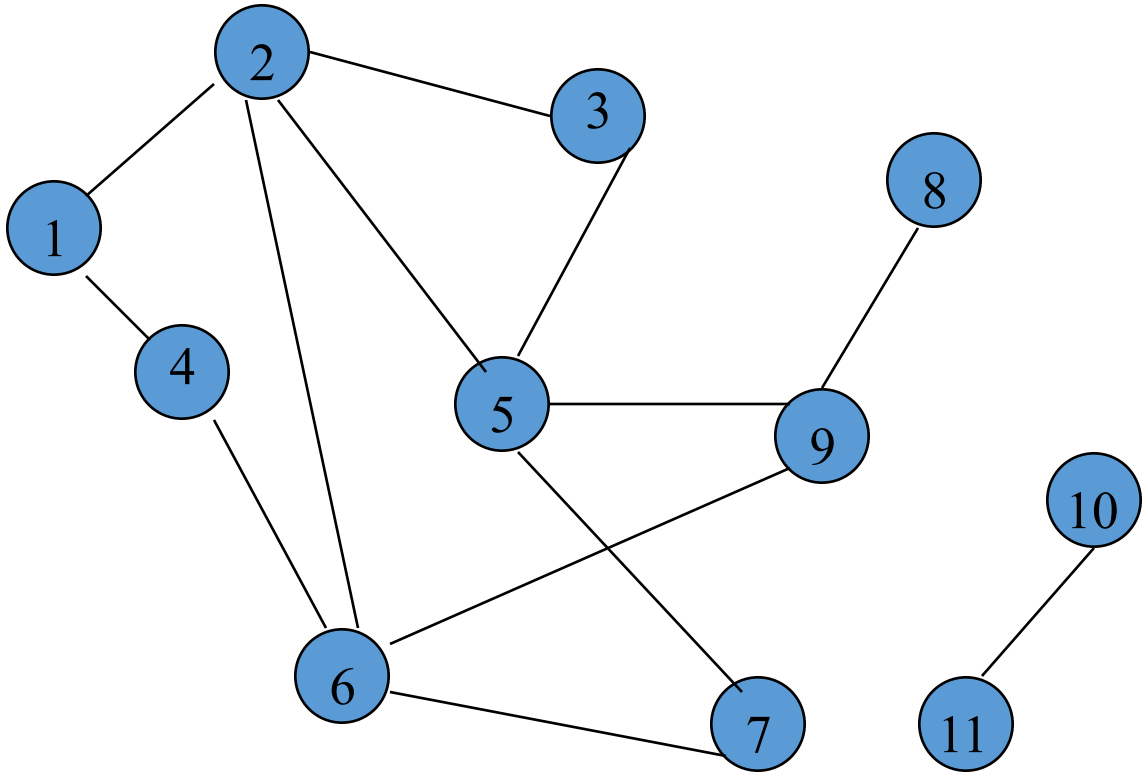
Graph Search Methods

- Many graph problems solved using a search method.
  - Path from one vertex to another.
  - Is the graph connected?
  - Find a spanning tree.
  - Etc.

Breadth-First Search

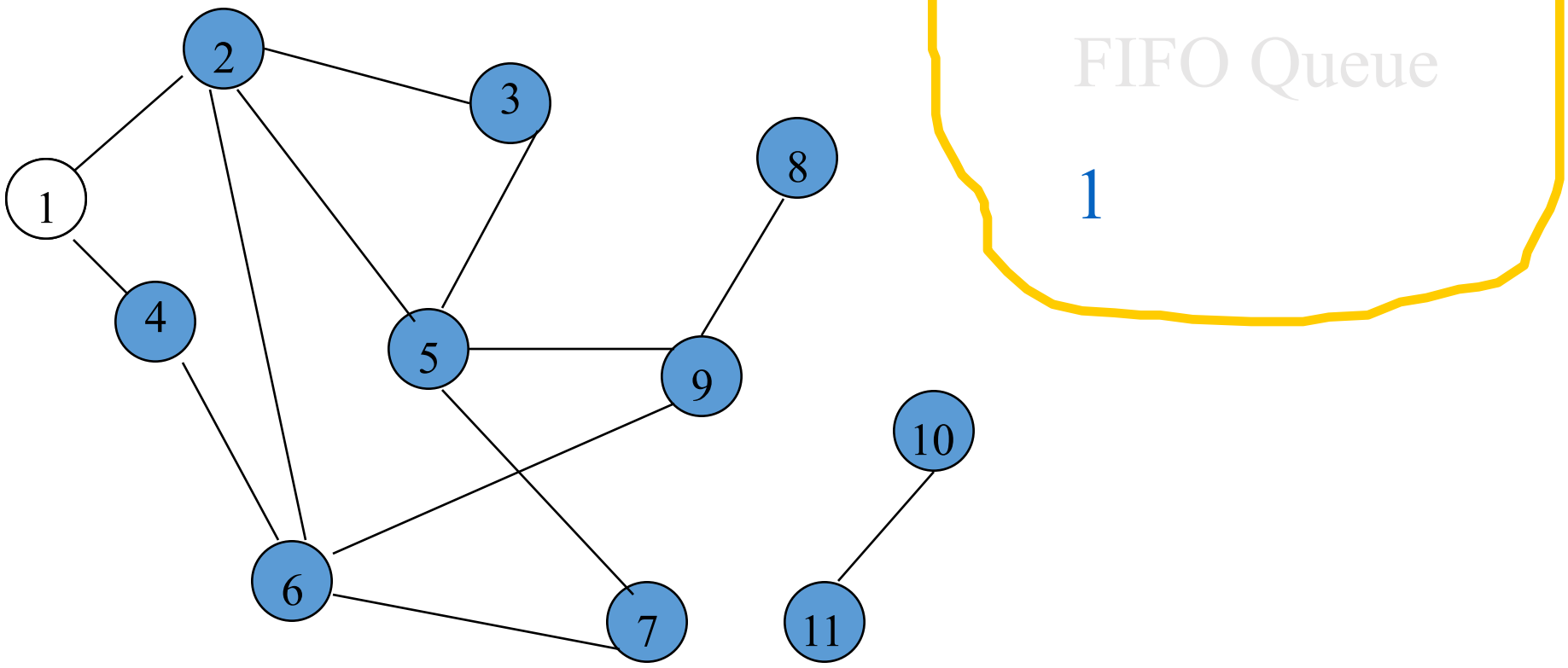
- Visit start vertex and put into a FIFO queue.
- Repeatedly remove a vertex from the queue, visit its unvisited adjacent vertices, put newly visited vertices into the queue.

Breadth-First Search Example



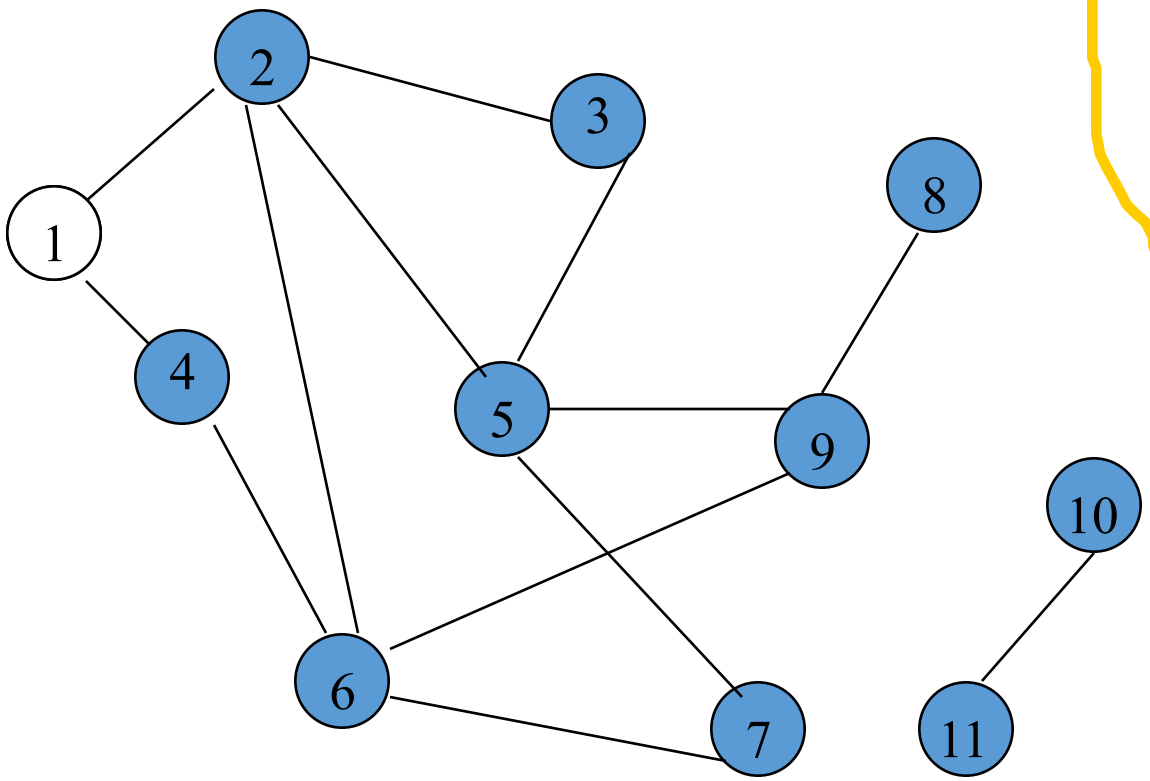
Start search at vertex 1.

Breadth-First Search Example



Visit/mark/label start vertex and put in a FIFO queue.

Breadth-First Search Example

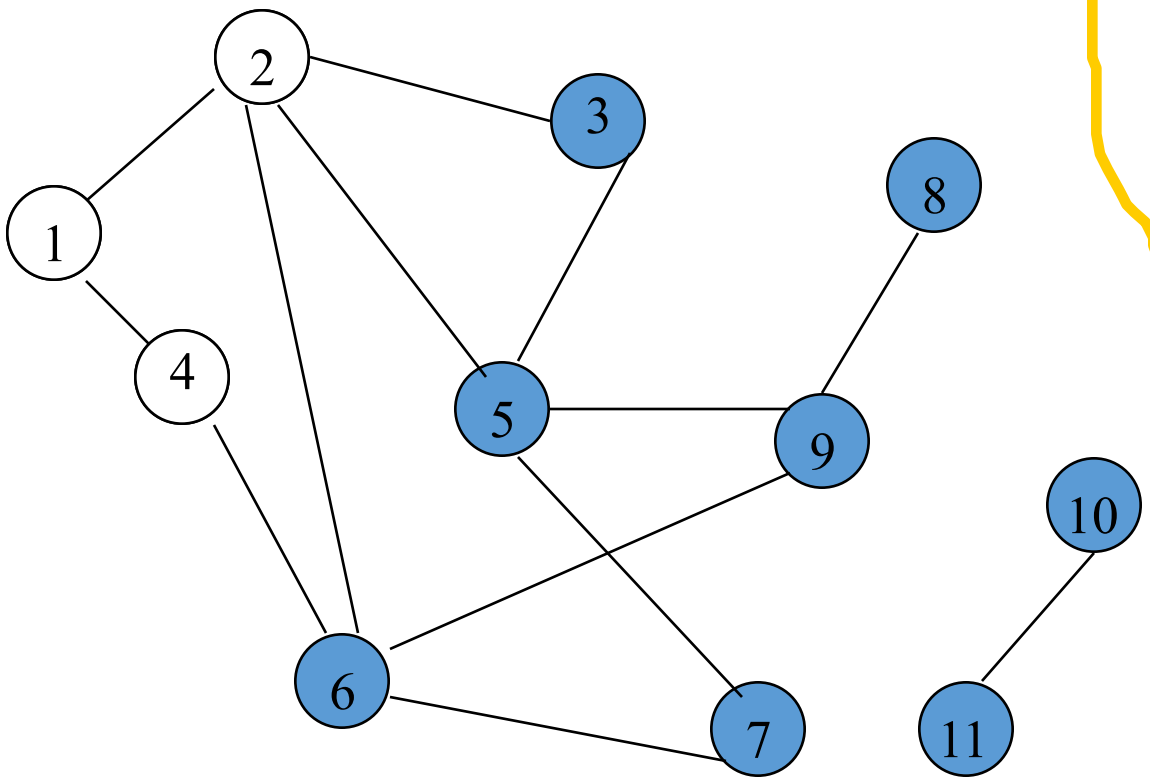


FIFO Queue

1

Remove 1 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example

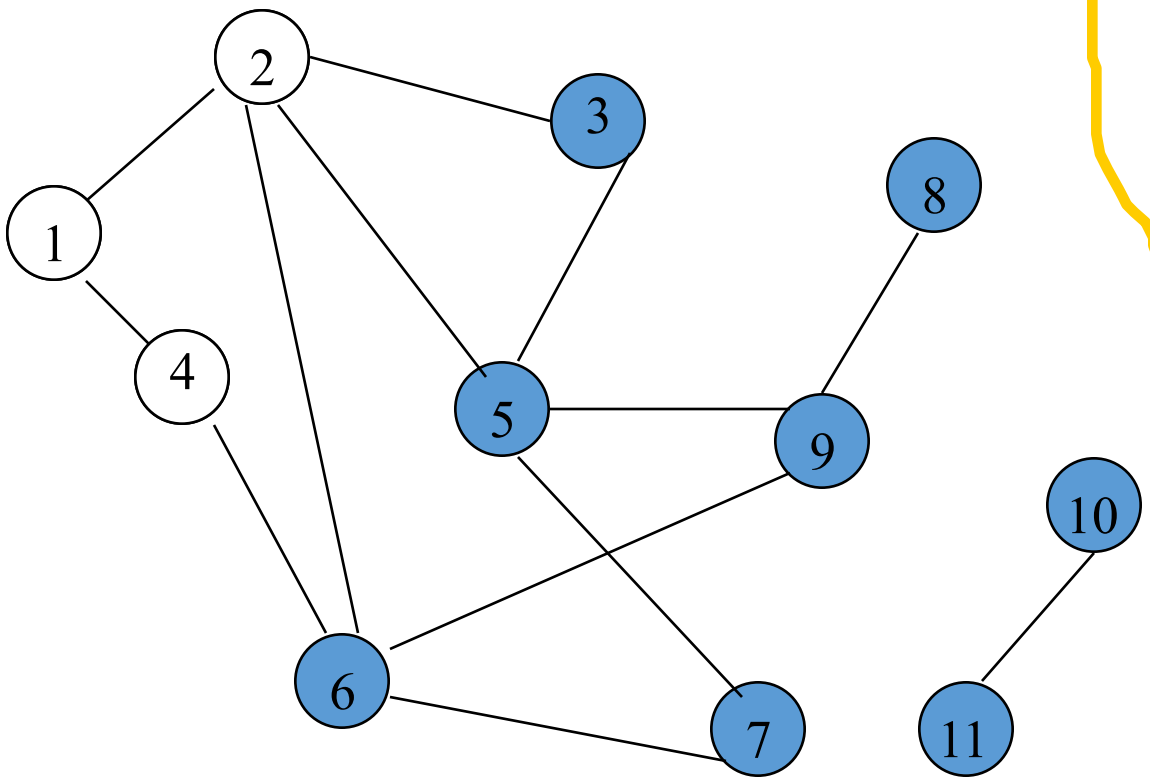


FIFO Queue

2 4

Remove 1 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example

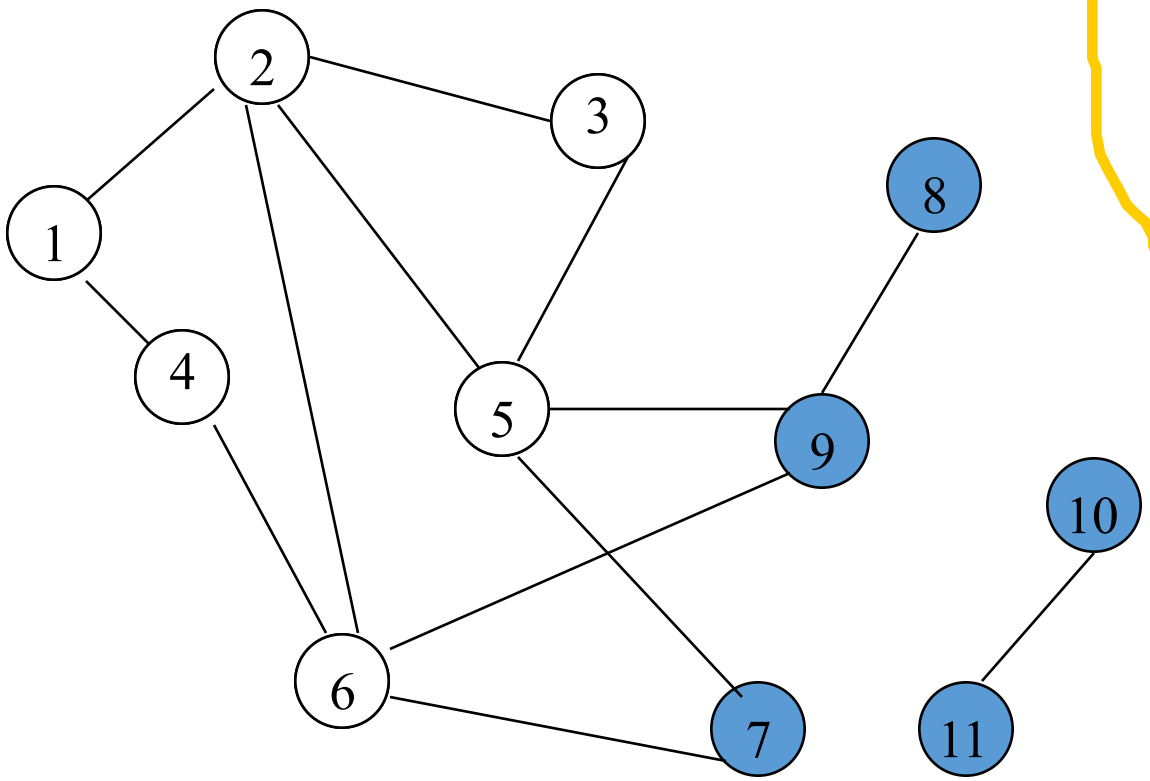


FIFO Queue

2 4

Remove 2 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example

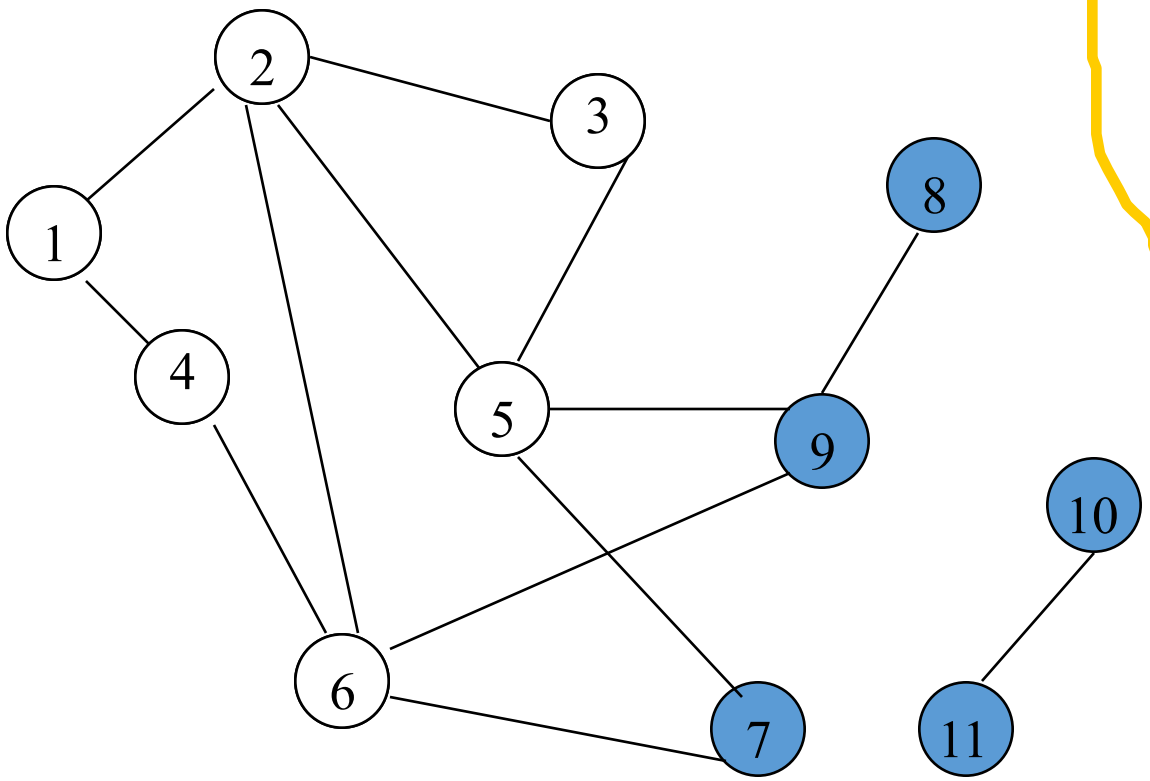


FIFO Queue

4 5 3 6

Remove 2 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example

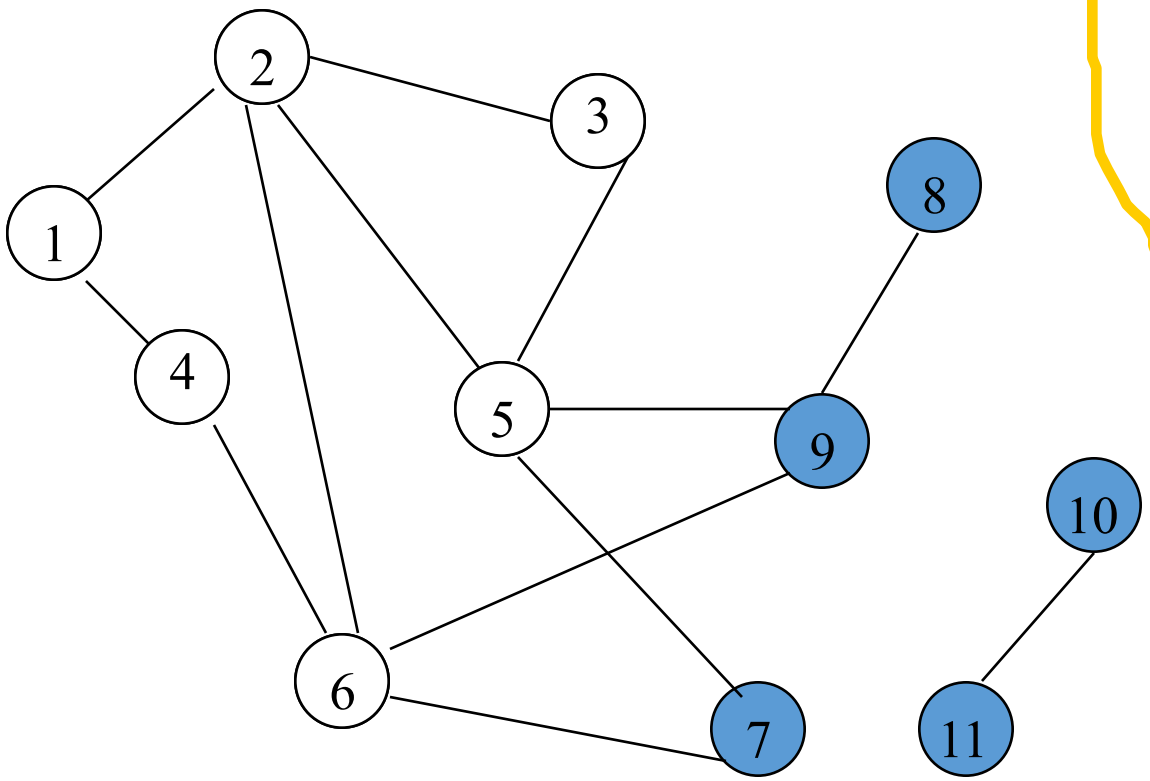


FIFO Queue

4 5 3 6

Remove 4 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example

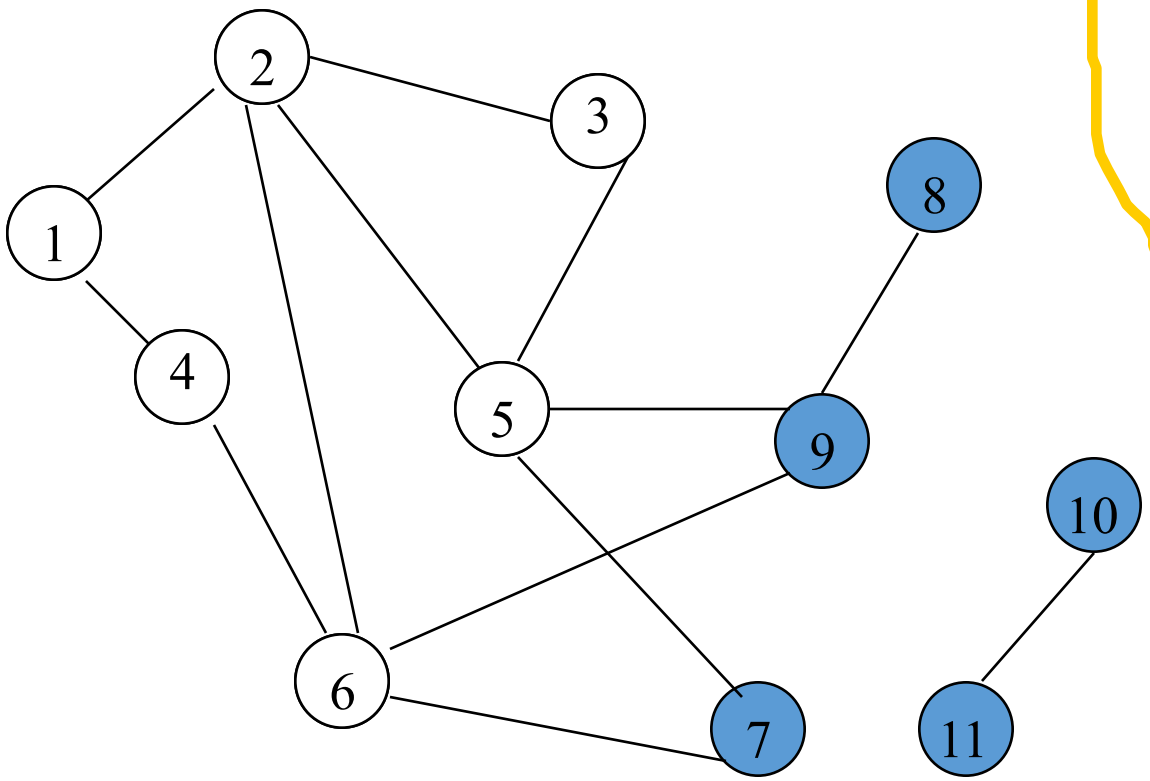


FIFO Queue

5 3 6

Remove 4 from Q; visit adjacent unvisited vertices;
put in Q.

Breadth-First Search Example

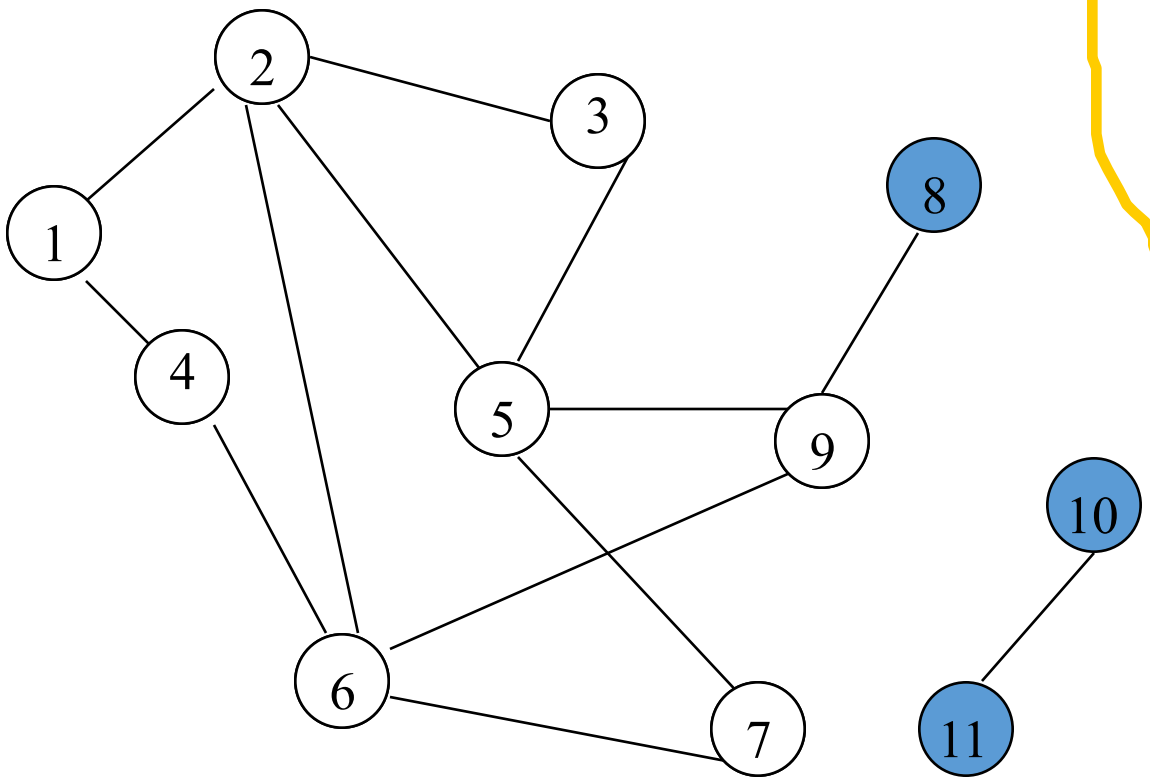


FIFO Queue

5 3 6

Remove 5 from Q; visit adjacent unvisited vertices;
put in Q.

Breadth-First Search Example

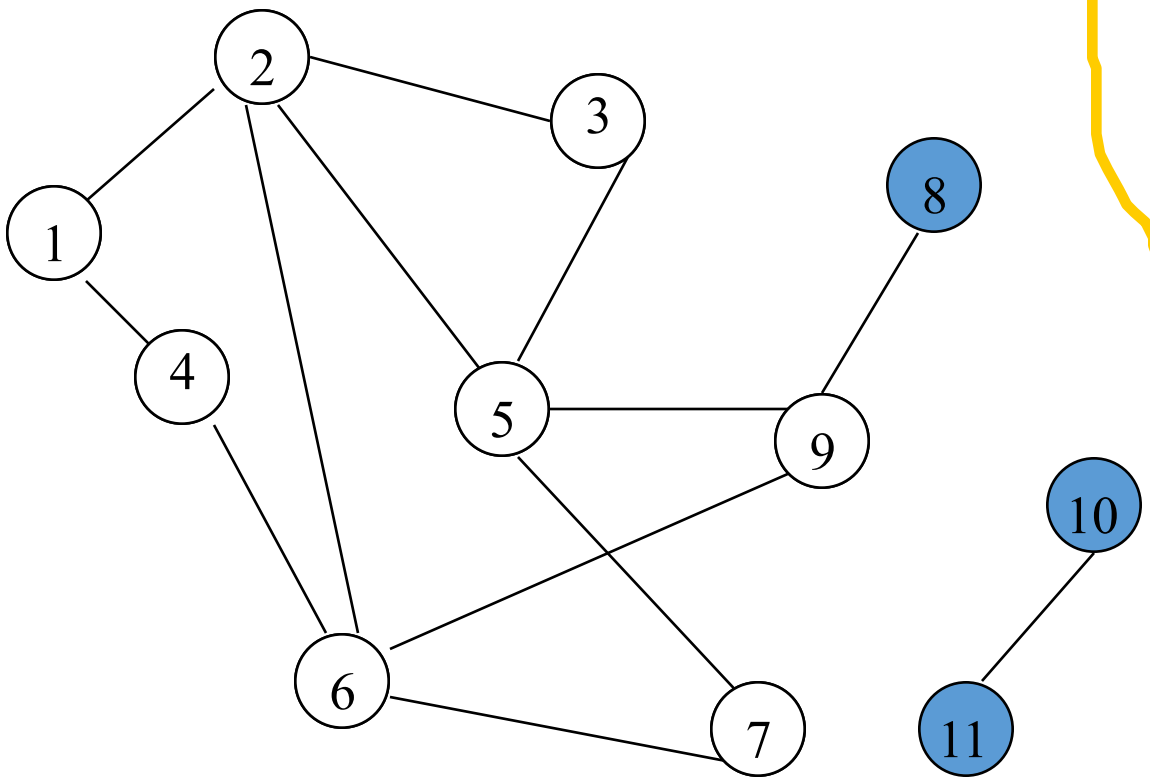


FIFO Queue

3 6 9 7

Remove 5 from Q; visit adjacent unvisited vertices;
put in Q.

Breadth-First Search Example

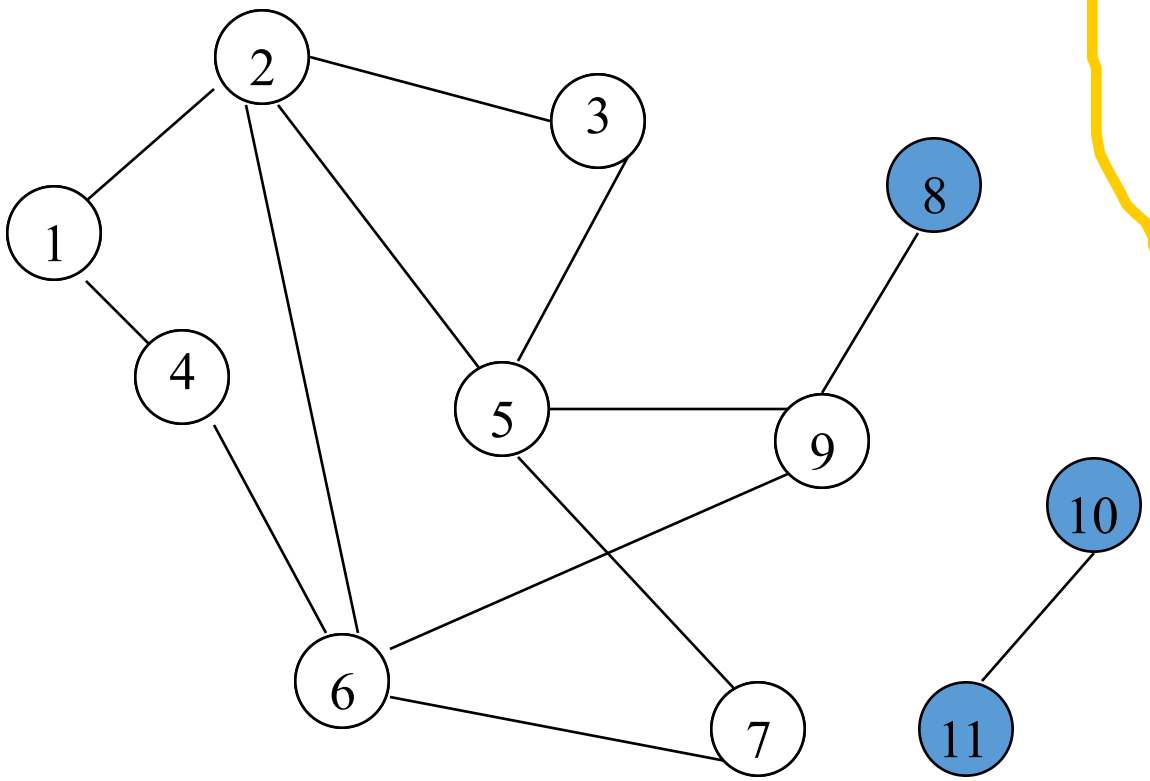


FIFO Queue

3 6 9 7

Remove 3 from Q; visit adjacent unvisited vertices;
put in Q.

Breadth-First Search Example

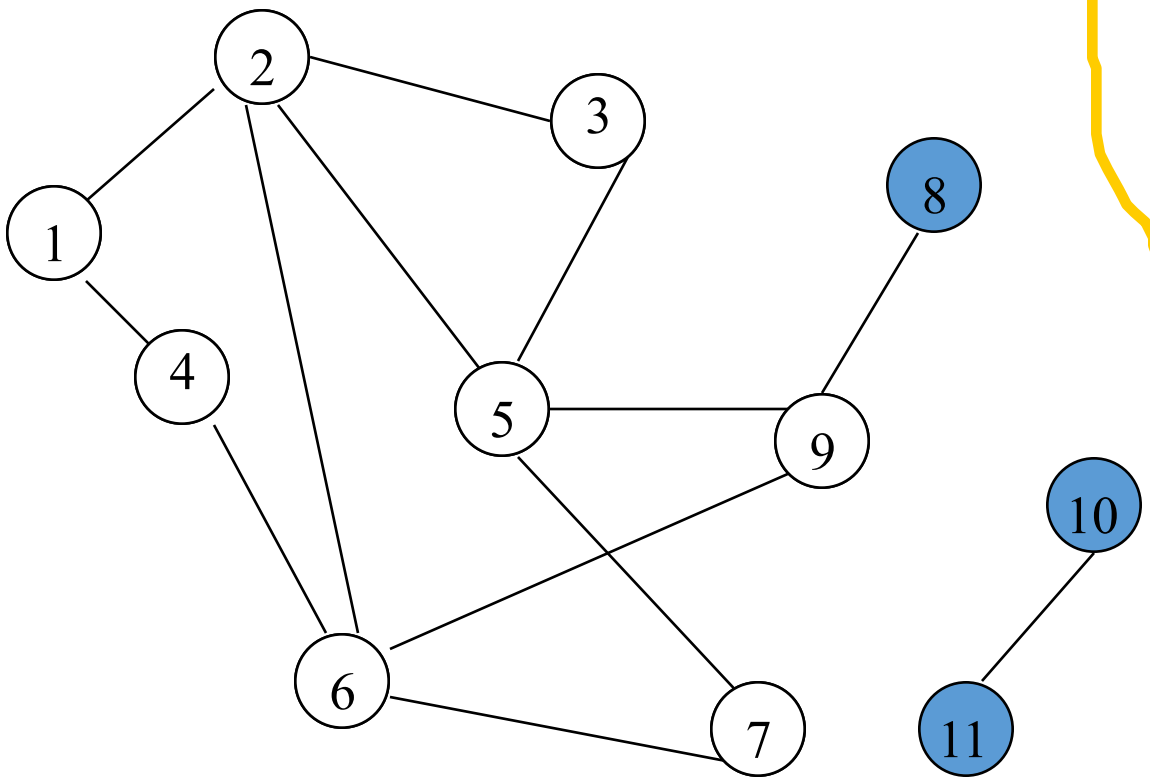


FIFO Queue

6 9 7

Remove 3 from Q; visit adjacent unvisited vertices;
put in Q.

Breadth-First Search Example

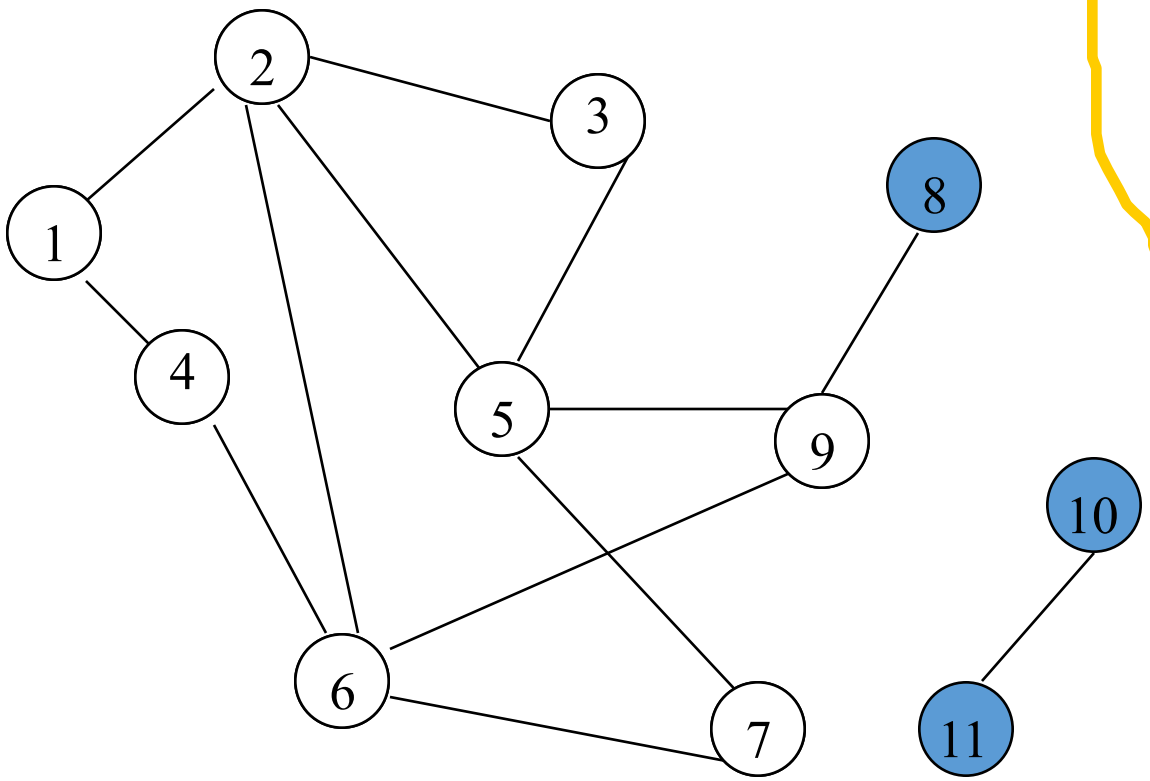


FIFO Queue

6 9 7

Remove 6 from Q; visit adjacent unvisited vertices;
put in Q.

Breadth-First Search Example

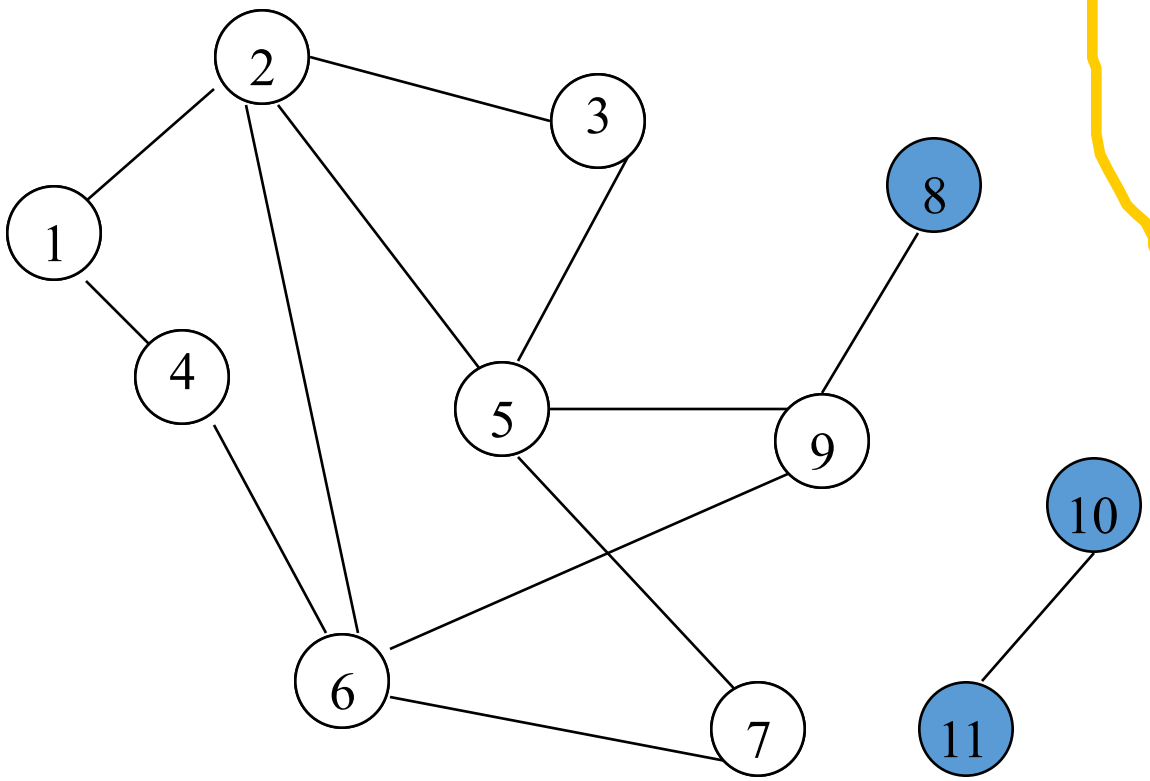


FIFO Queue

9 7

Remove 6 from Q; visit adjacent unvisited vertices;
put in Q.

Breadth-First Search Example

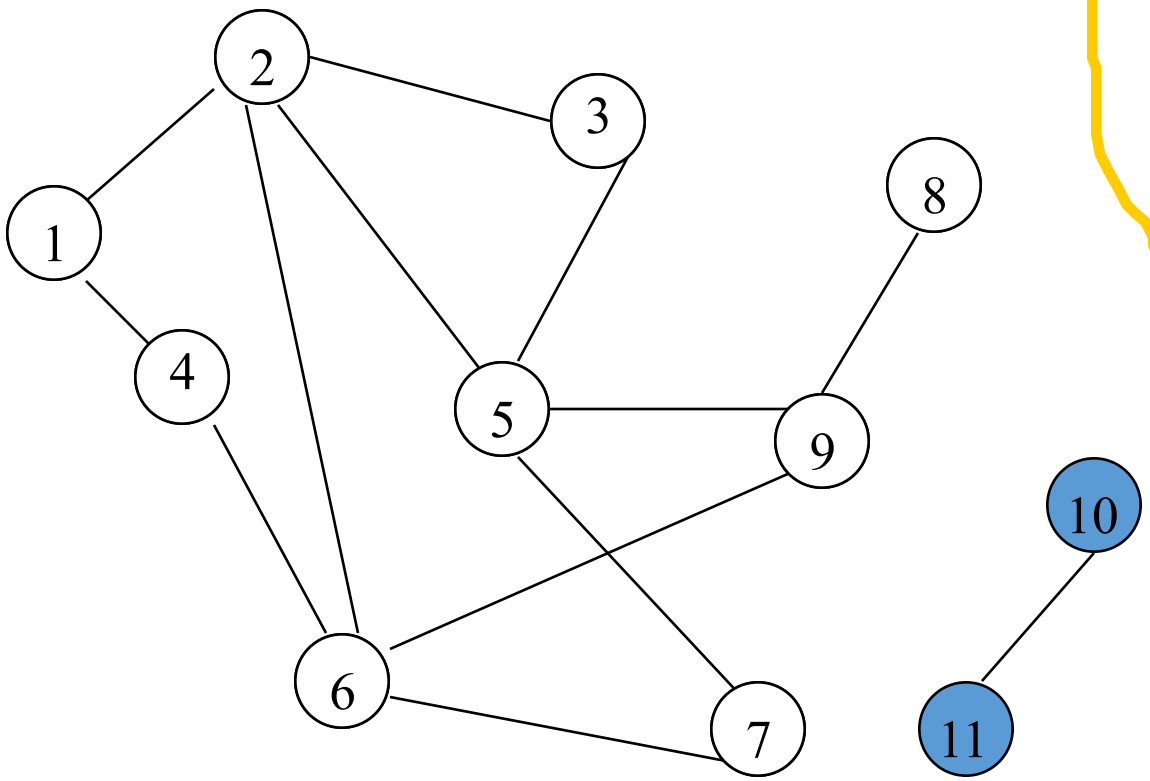


FIFO Queue

9 7

Remove 9 from Q; visit adjacent unvisited vertices;
put in Q.

Breadth-First Search Example

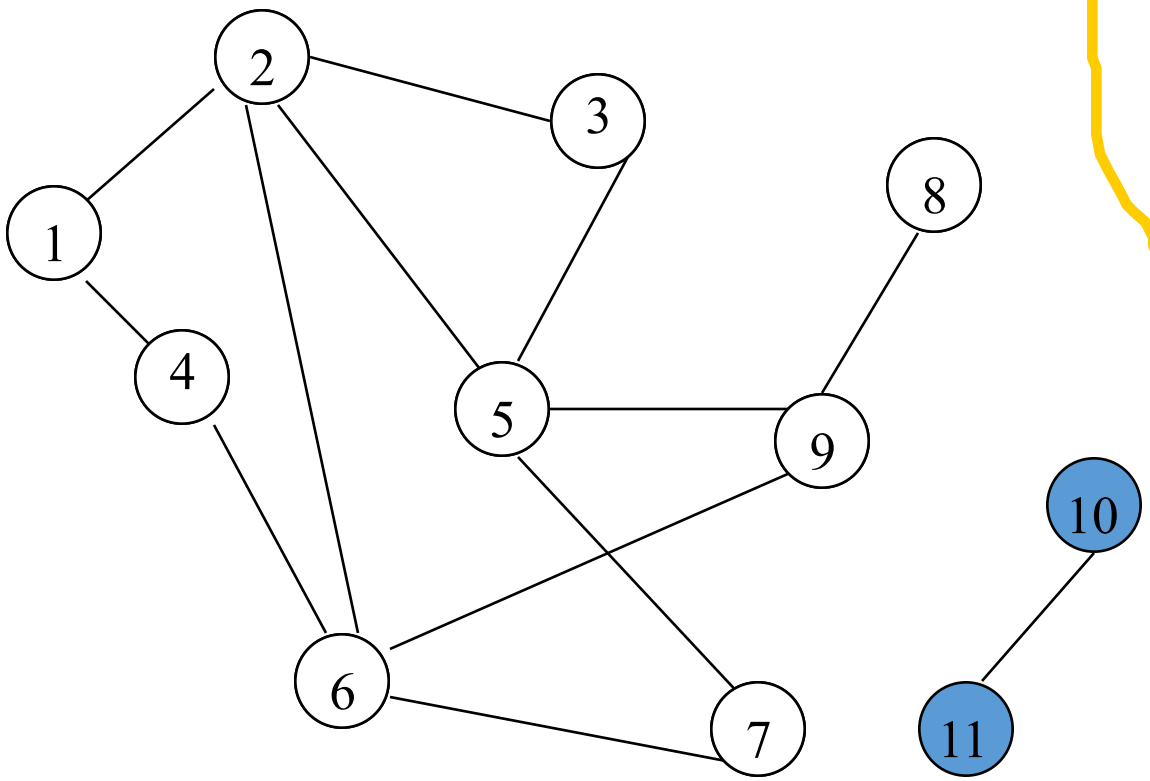


FIFO Queue

7 8

Remove 9 from Q; visit adjacent unvisited vertices;
put in Q.

Breadth-First Search Example

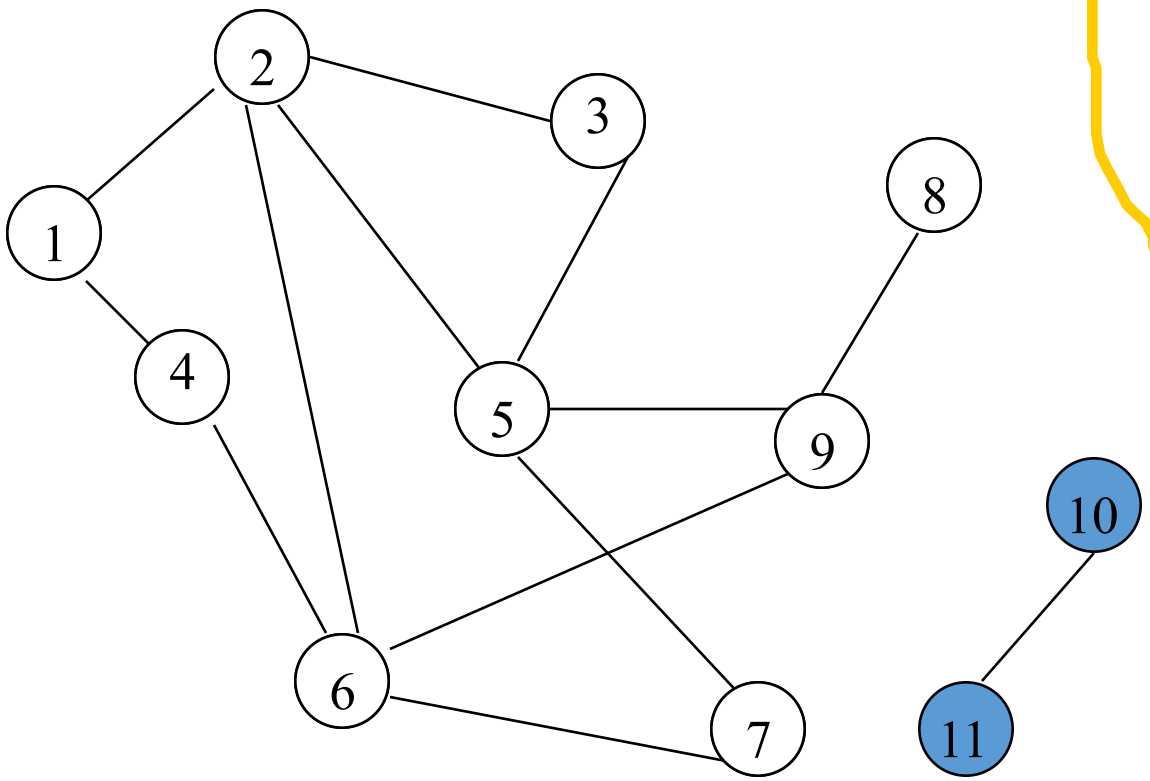


FIFO Queue

7 8

Remove 7 from Q; visit adjacent unvisited vertices;
put in Q.

Breadth-First Search Example

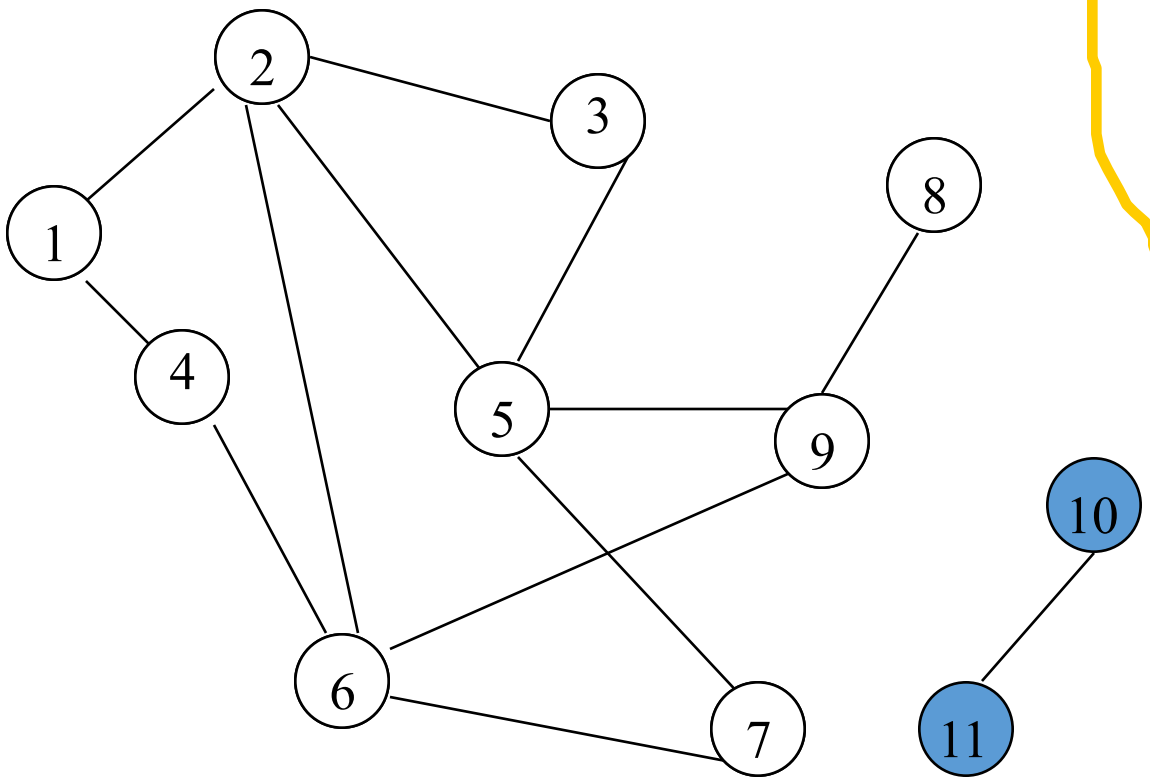


FIFO Queue

8

Remove 7 from Q; visit adjacent unvisited vertices;
put in Q.

Breadth-First Search Example

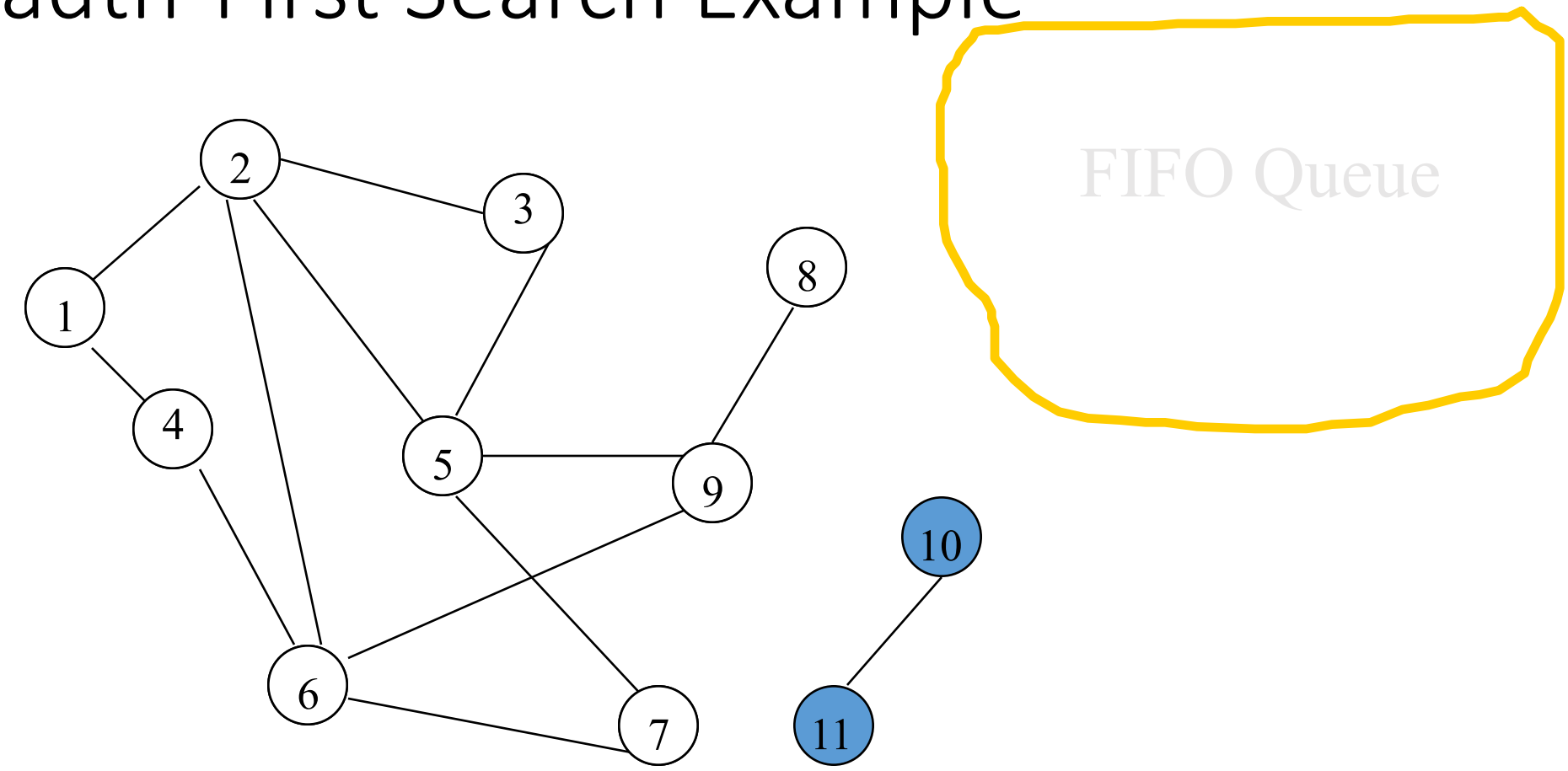


FIFO Queue

8

Remove 8 from Q; visit adjacent unvisited vertices;
put in Q.

Breadth-First Search Example



Queue is empty. Search terminates.

Breadth-First Search Property

- All vertices reachable from the start vertex (including the start vertex) are visited.

Time Complexity

- Each visited vertex is put on (and so removed from) the queue exactly once.

- When a vertex is removed from the queue, we examine its adjacent vertices.
  - Efficiency is proportional to size of data structure used
    - $\Theta(|V| + |E|)$ if adjacency list used
    - $\Theta(|V|)^2$ if adjacency matrix used

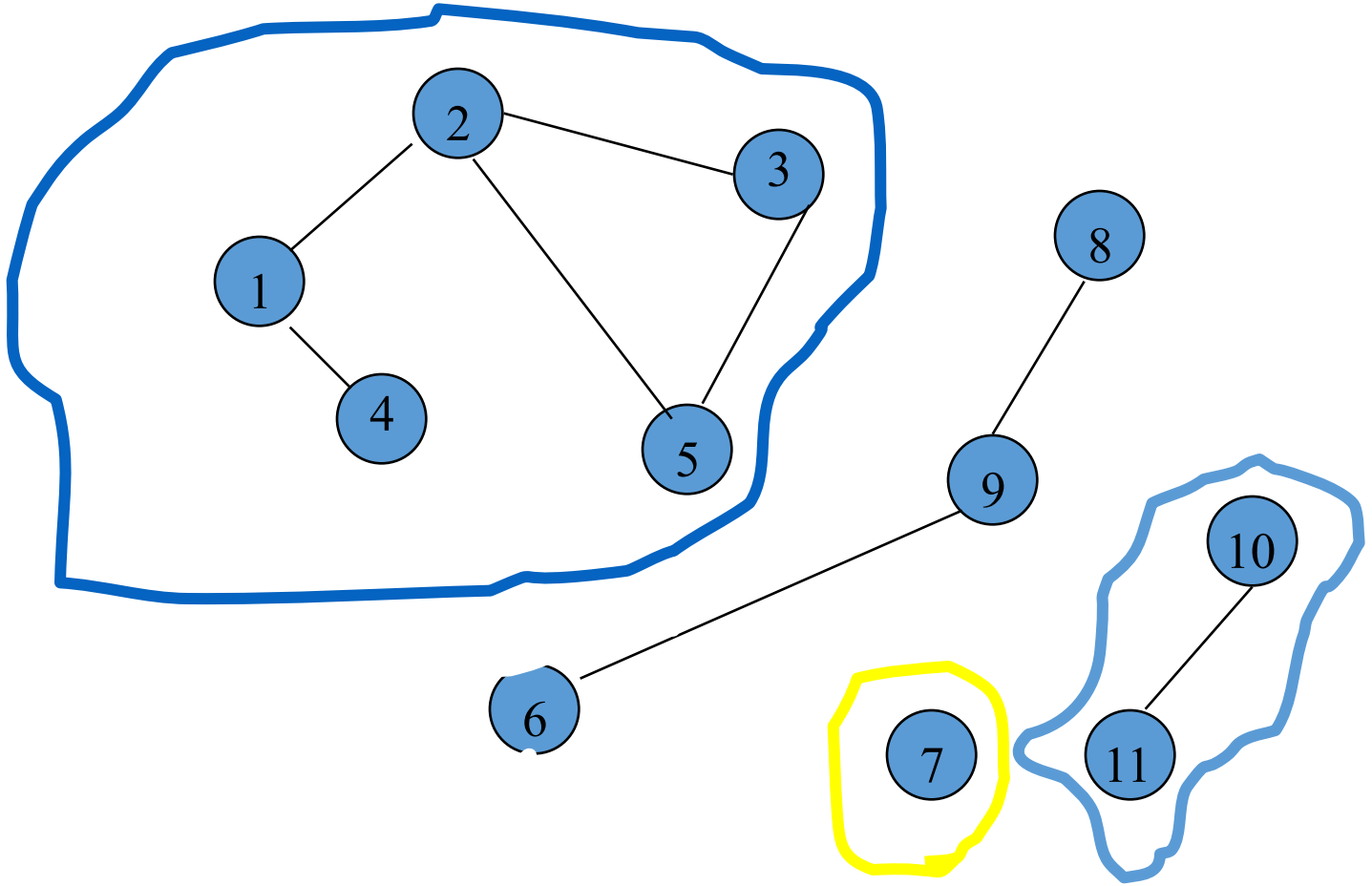
Path From Vertex *v* To Vertex *u*

- Start a breadth-first search at vertex *v*.
- Terminate when vertex *u* is visited or when *Q* becomes empty (whichever occurs first).
- Time
  - $\Theta(|V| + |E|)$ if adjacency list used
  - $\Theta(|V|)^2$ if adjacency matrix used

Is The Graph Connected?

- Start a breadth-first search at any vertex of the graph.
- Graph is connected iff all n vertices get visited.
- Time
  - $\Theta(|V|+|E|)$ if adjacency list used
  - $\Theta(|V|)^2$ if adjacency matrix used

Connected Components



Connected Components

- Start a breadth-first search at any as yet unvisited vertex of the graph.
- Newly visited vertices (plus edges between them) define a component.
- Repeat until all vertices are visited.

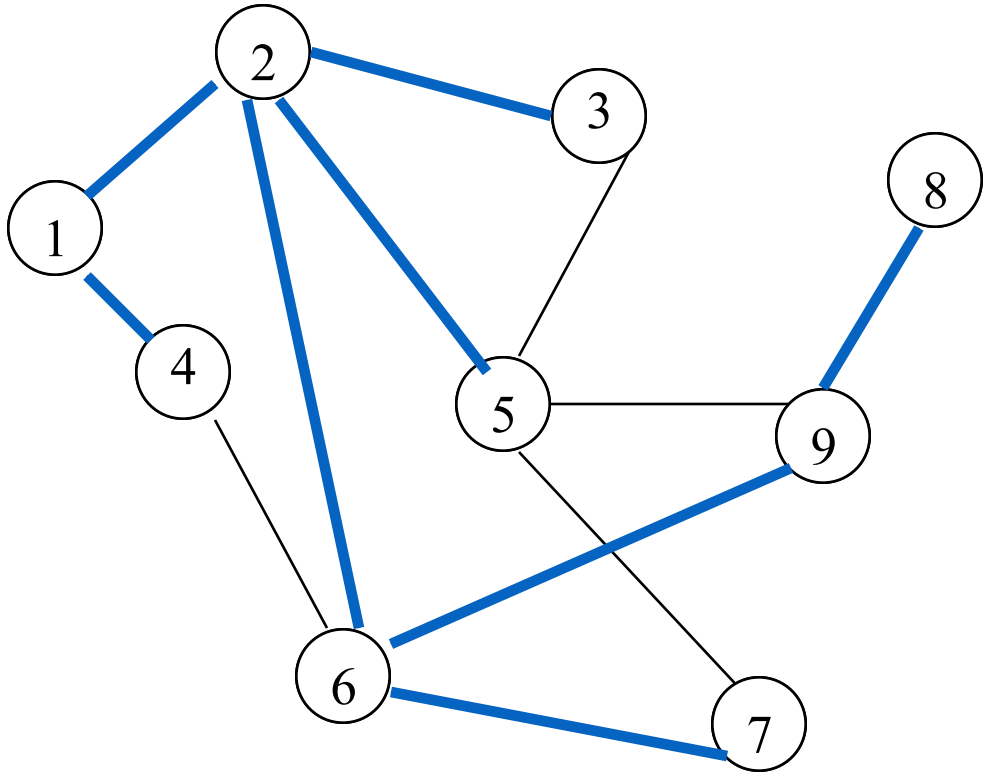
Time Complexity

- $\Theta(|V| + |E|)$ if adjacency list used
- $\Theta(|V|)^2$ if adjacency matrix used

Spanning Tree

- A subset of edges sufficient to keep a graph connected if all other edges are removed
- Start a breadth-first search at any vertex of the graph.
- If graph is connected, the edges used to get to ***unvisited*** vertices define a spanning tree (breadth-first spanning tree).
- Time
  - $\Theta(|V| + |E|)$ if adjacency list used
  - $\Theta(|V|)^2$ if adjacency matrix used

Breadth-First Spanning Tree



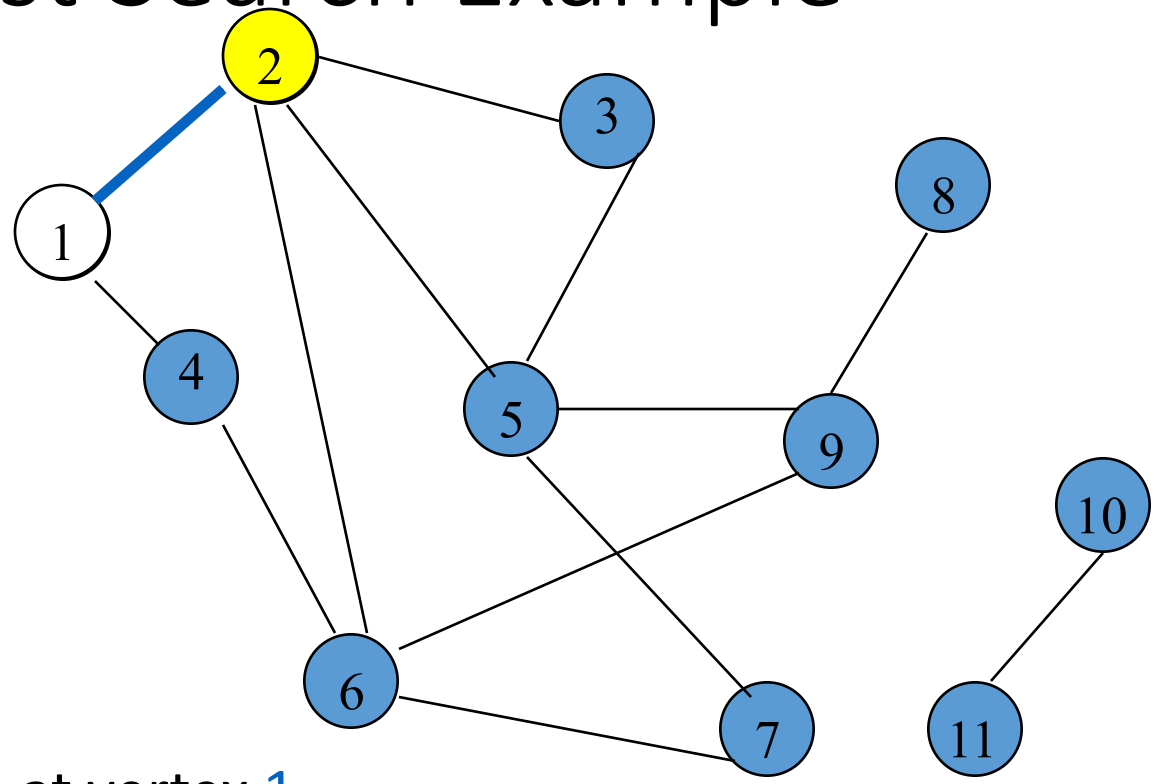
Breadth-first search from vertex 1.

Breadth-first spanning tree.

Depth-First Search

```
depthFirstSearch(v)
{
  Label vertex v as reached.
  for (each unreached vertex u adjacenct from v)
        depthFirstSearch(u);
}
```

Depth-First Search Example

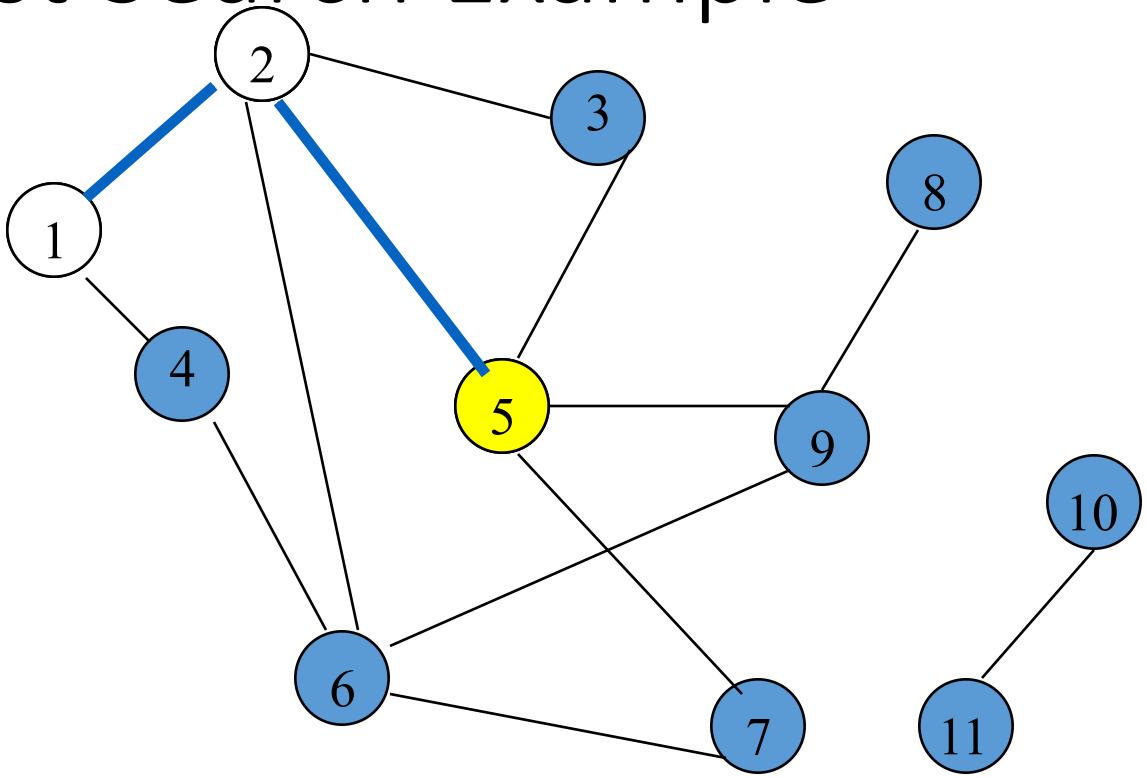


Start search at vertex 1.

Label vertex 1 and do a depth first search from either 2 or 4.

Suppose that vertex 2 is selected.

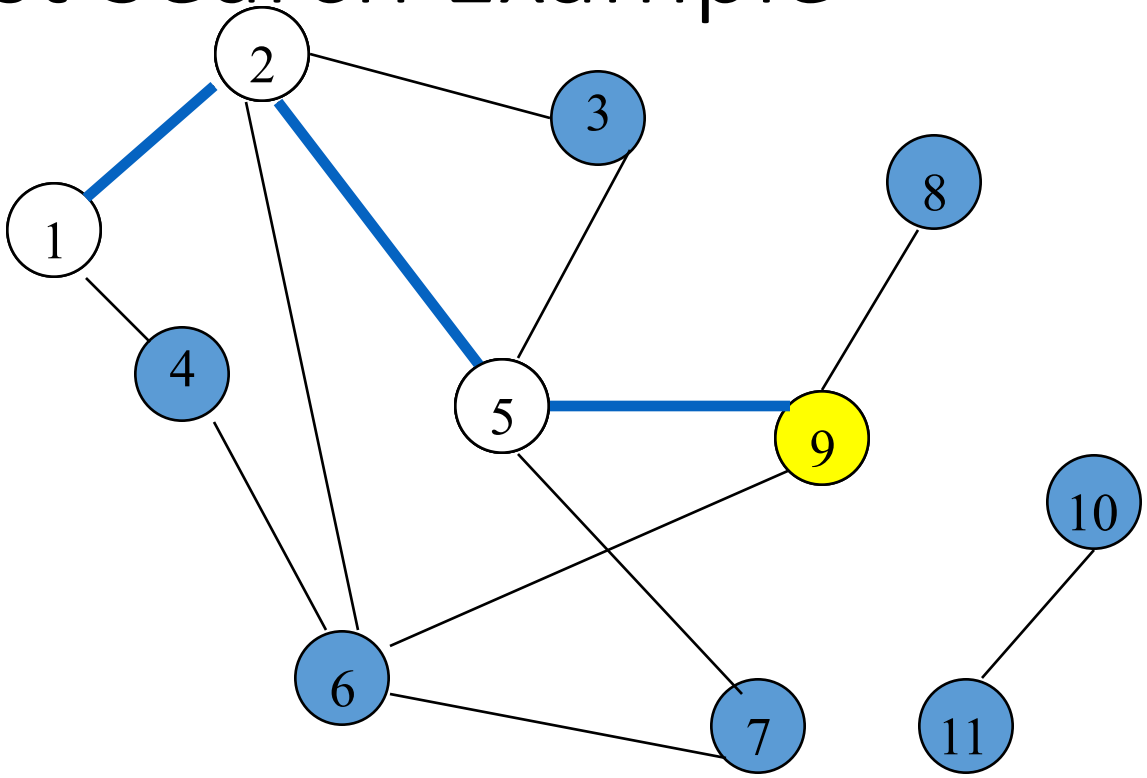
Depth-First Search Example



Label vertex 2 and do a depth first search from either 3, 5, or 6.

Suppose that vertex 5 is selected.

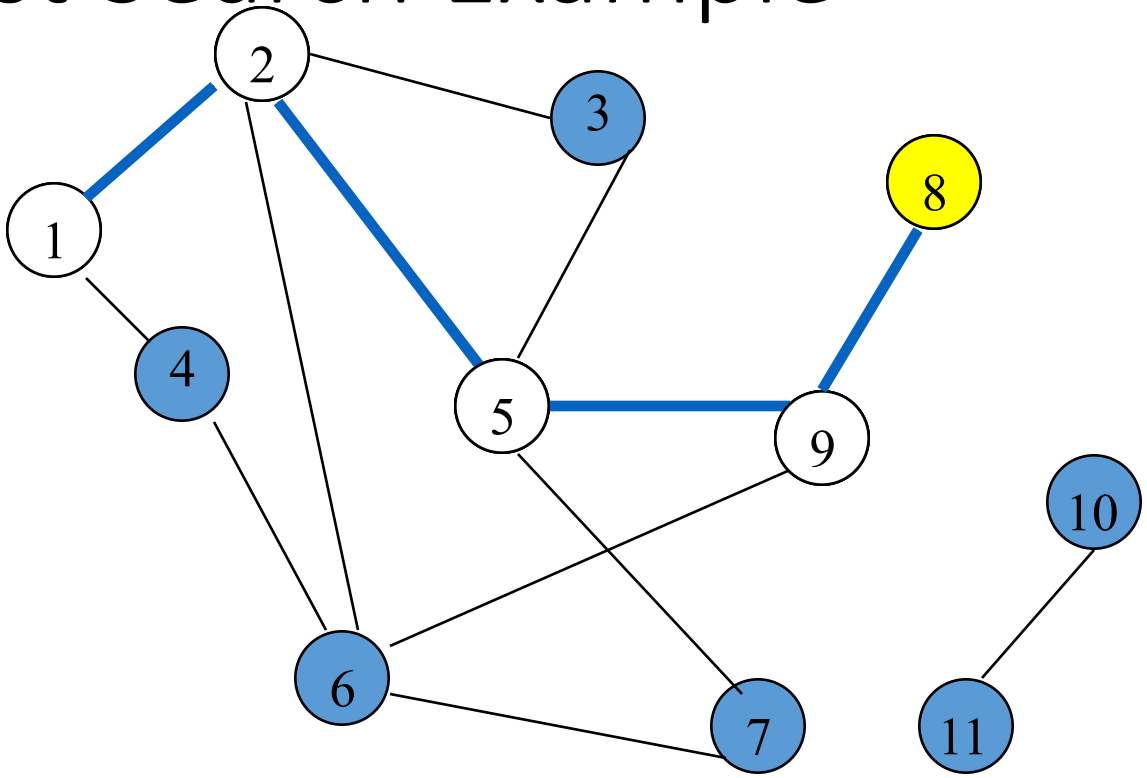
Depth-First Search Example



Label vertex 5 and do a depth first search from either 3, 7, or 9.

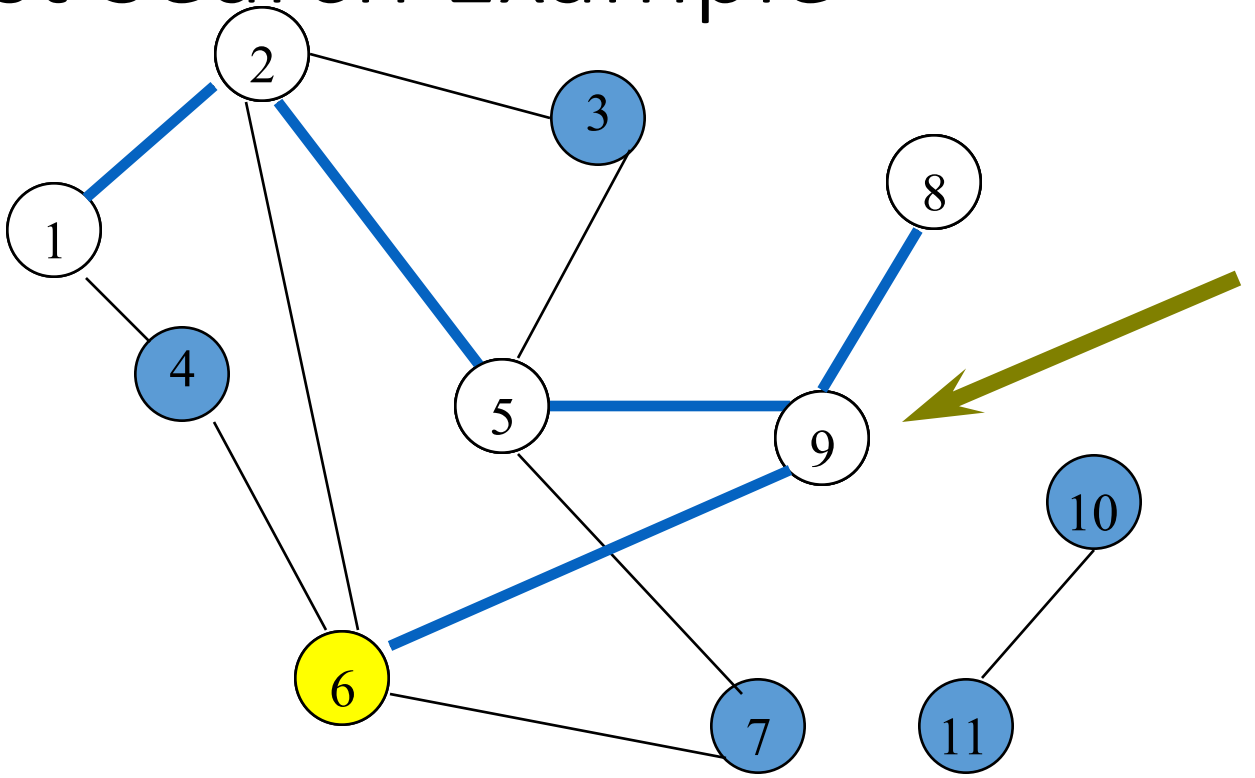
Suppose that vertex 9 is selected.

Depth-First Search Example



Label vertex 9 and do a depth first search from either 6 or 8.
Suppose that vertex 8 is selected.

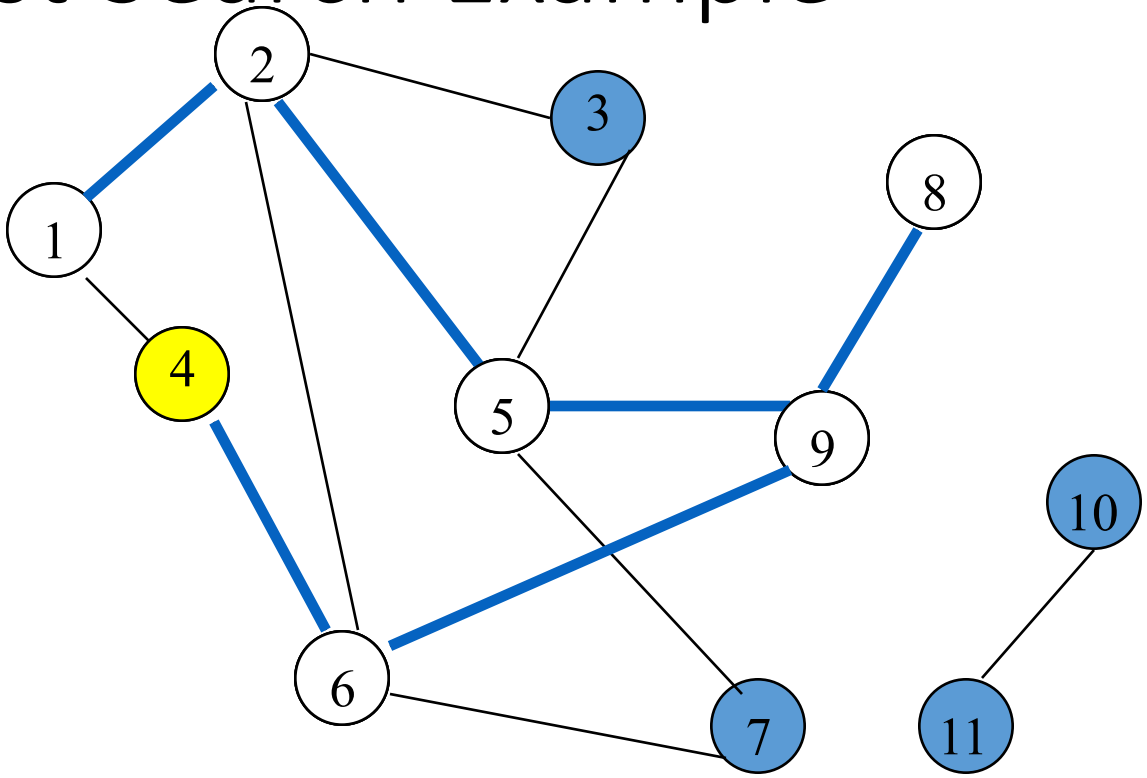
Depth-First Search Example



Label vertex 8 and return to vertex 9.

From vertex 9 do a dfs(6).

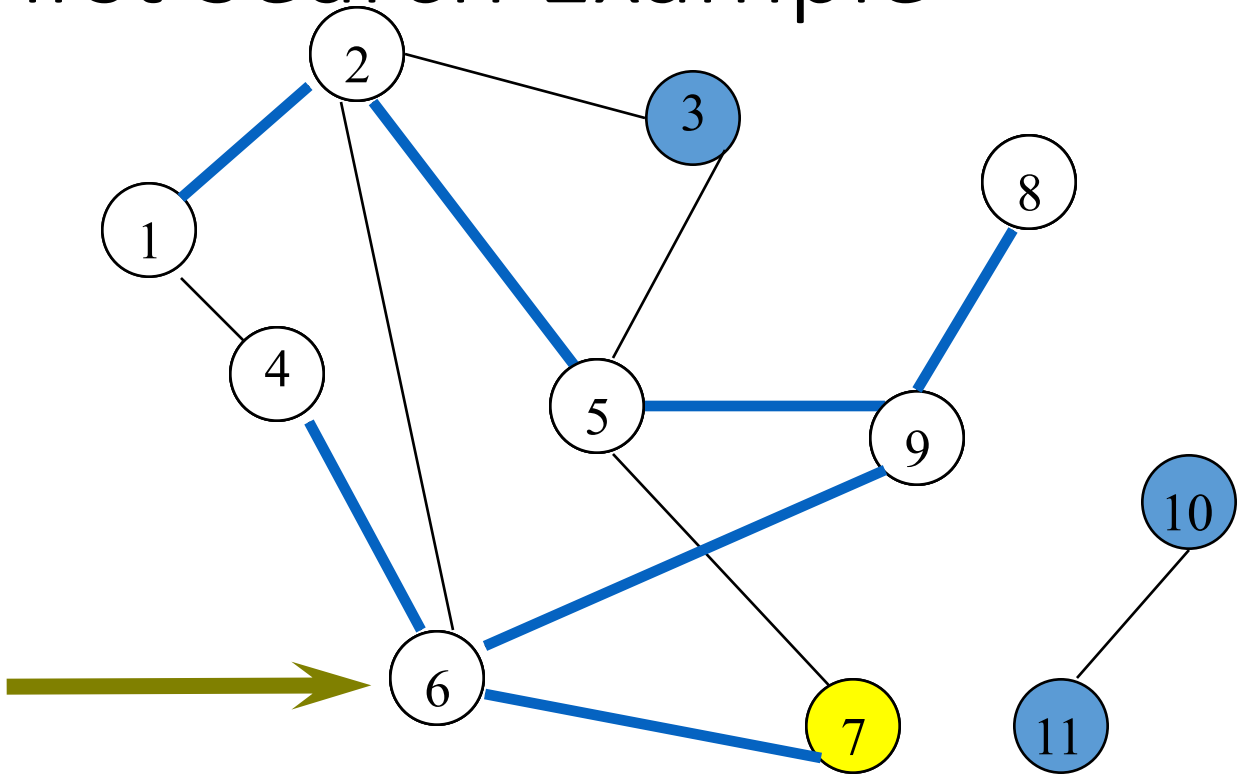
Depth-First Search Example



Label vertex 6 and do a depth first search from either 4 or 7.

Suppose that vertex 4 is selected.

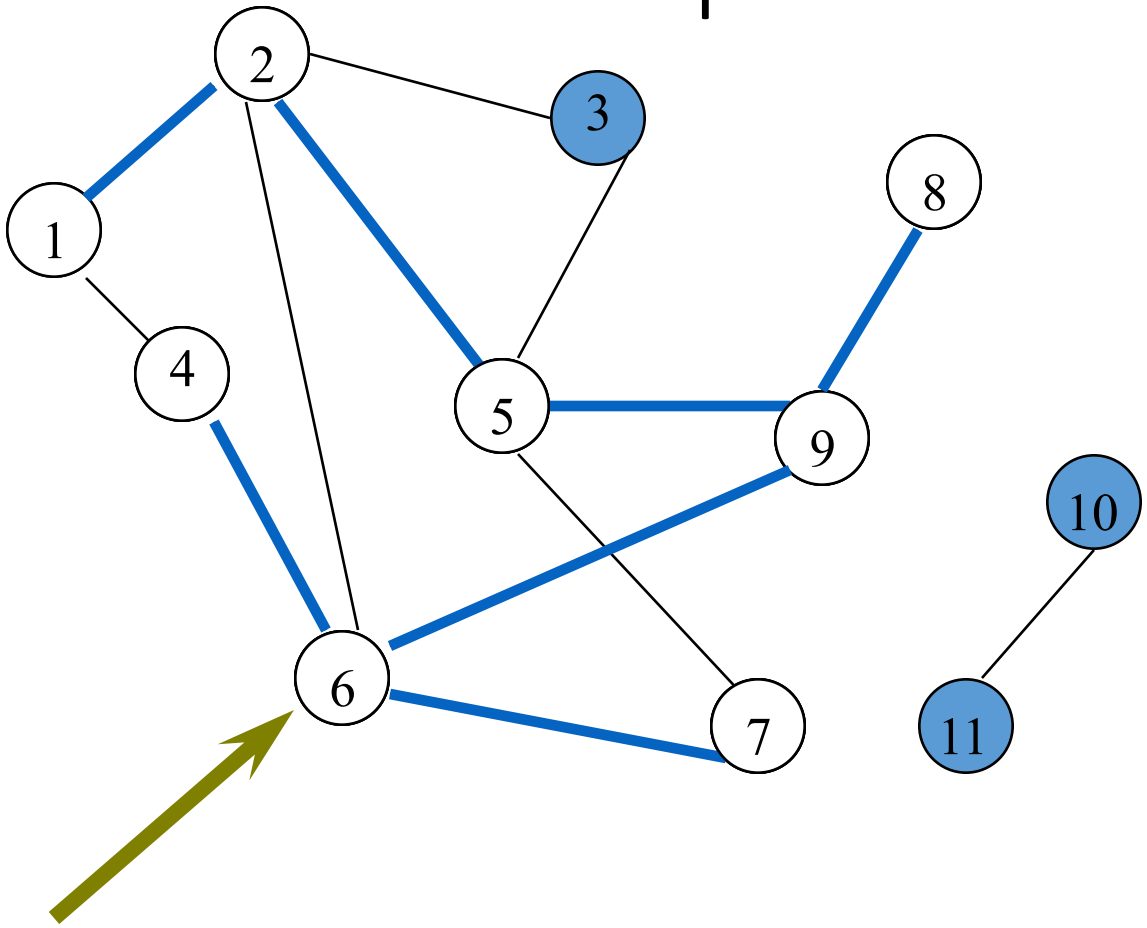
Depth-First Search Example



Label vertex 4 and return to 6.

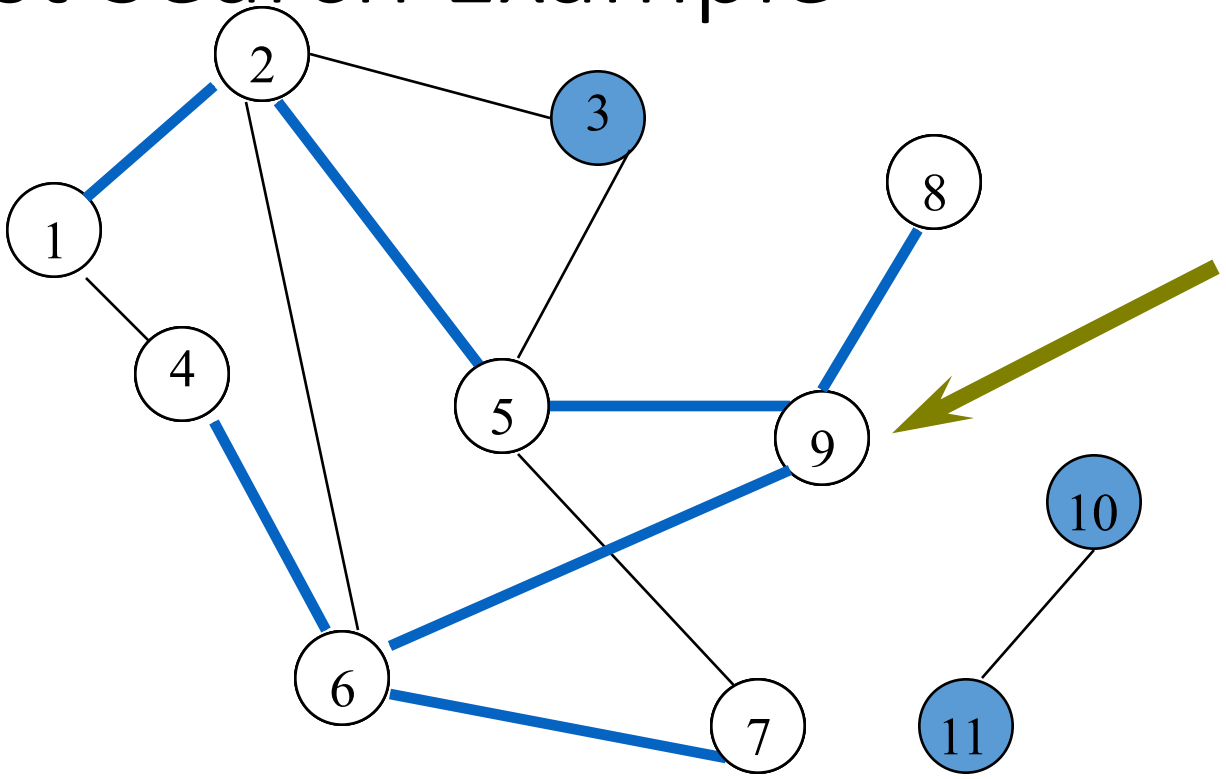
From vertex 6 do a dfs(7).

Depth-First Search Example



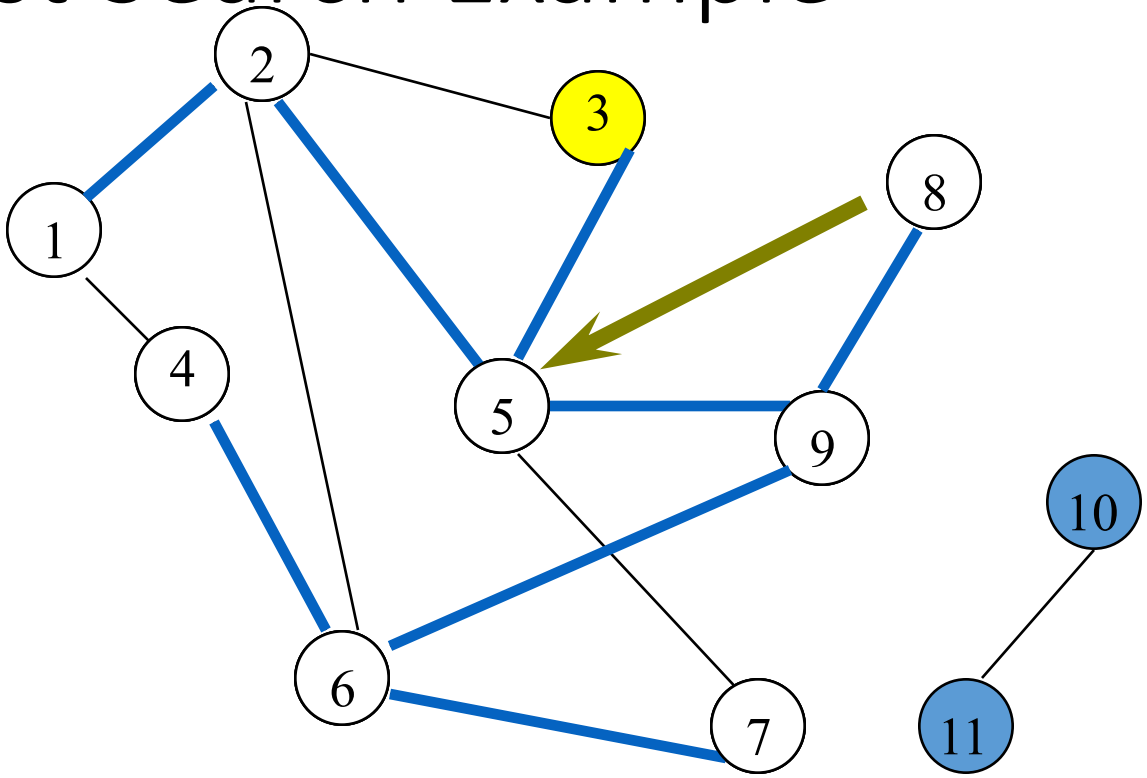
Label vertex 7 and return to 6.

Depth-First Search Example



Return to 9.

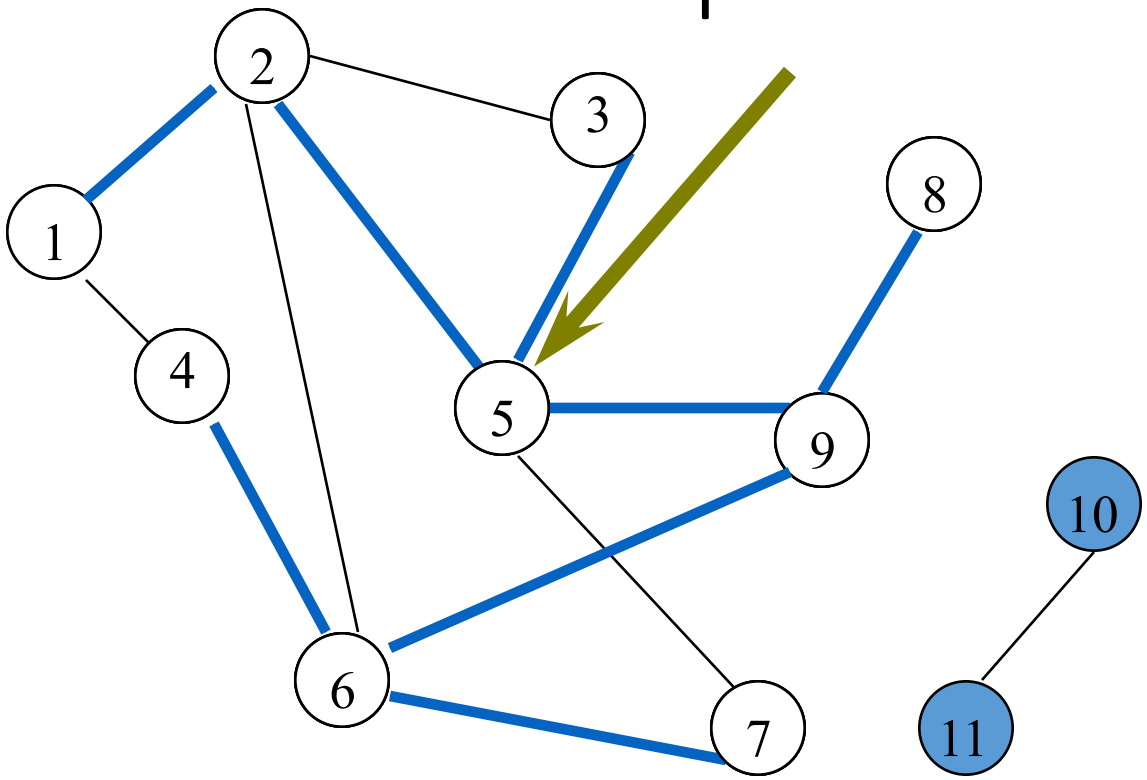
Depth-First Search Example



Return to 5.

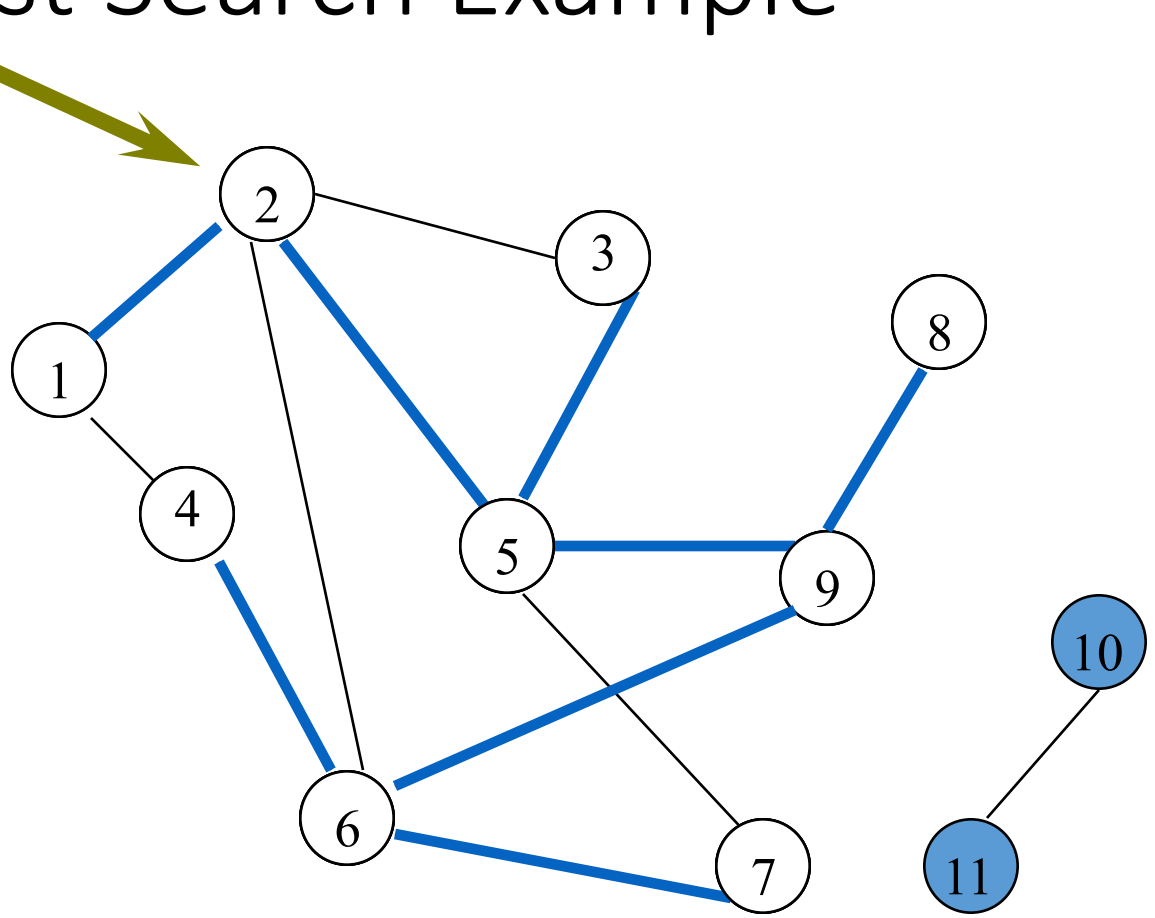
Do a dfs(3).

Depth-First Search Example



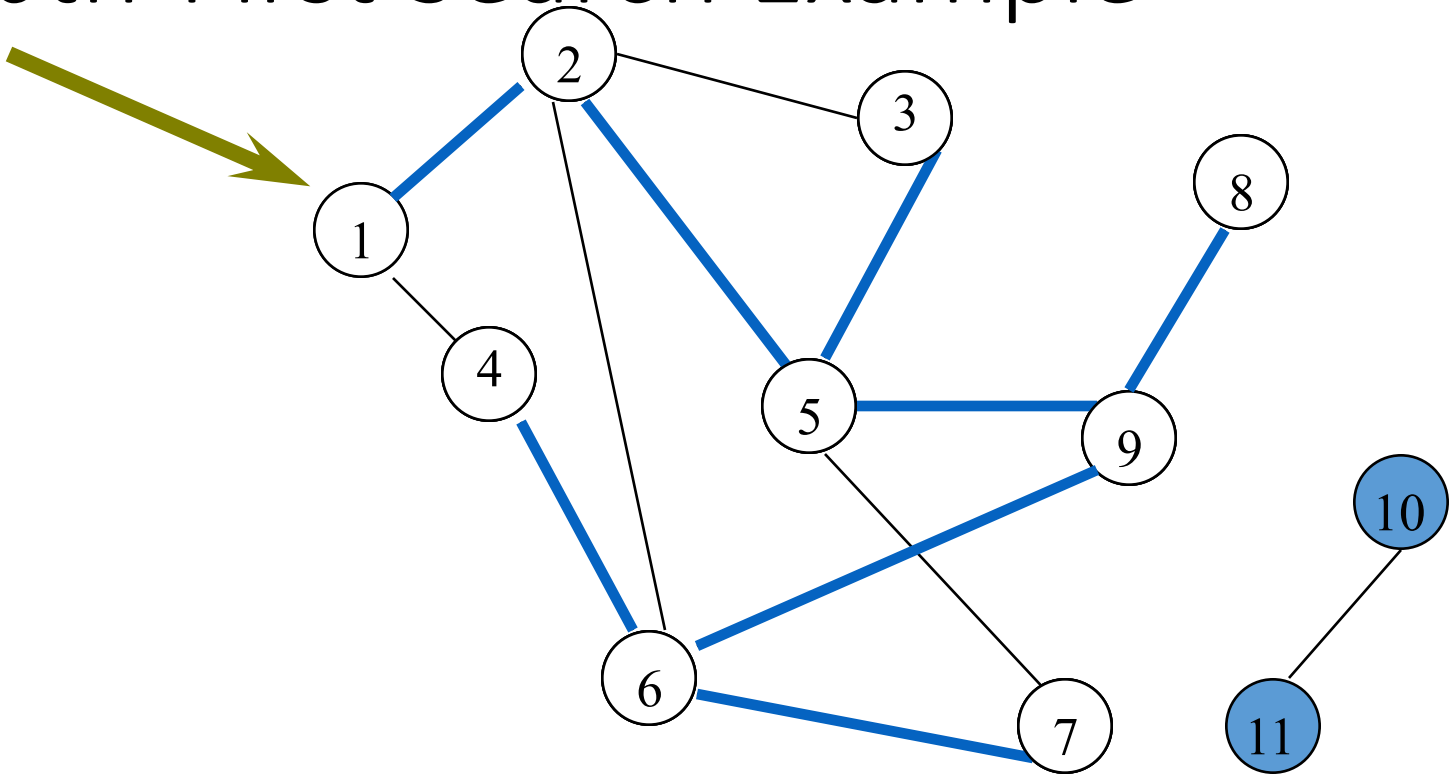
Label 3 and return to 5.

Depth-First Search Example



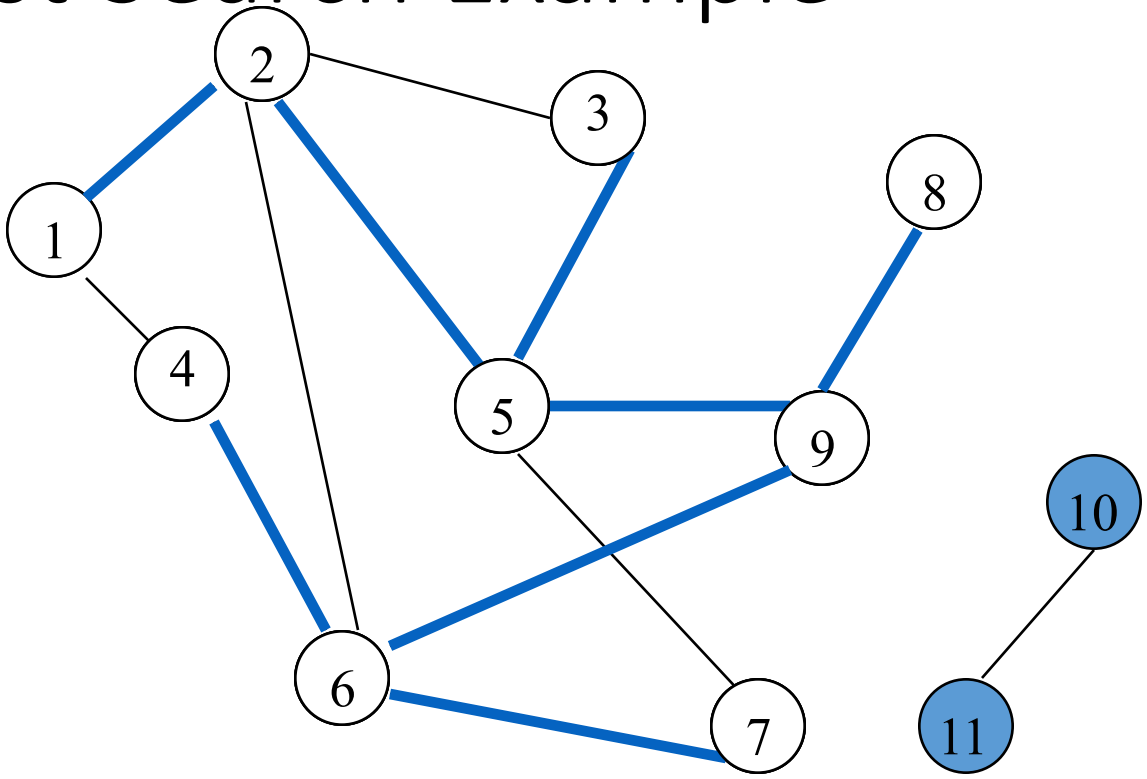
Return to 2.

Depth-First Search Example



Return to 1.

Depth-First Search Example



Return to invoking method.

Depth-First Search Properties

- Same complexity as BFS.
- Same properties with respect to path finding, connected components, and spanning trees.
- Edges used to reach unlabeled vertices define a depth-first spanning tree when the graph is connected.
- There are problems for which bfs is better than dfs and vice versa.

Exercise

a) Explain how one can check a graph's acyclicity by using breadth-first search.

b) Does either of the two traversals–DFS or BFS–always find a cycle faster than the other? If you answer yes, indicate which of them is better and explain why it is the case; if you answer no, give two examples supporting your answer.

Plan for Next Lecture: Decrease & Conquer