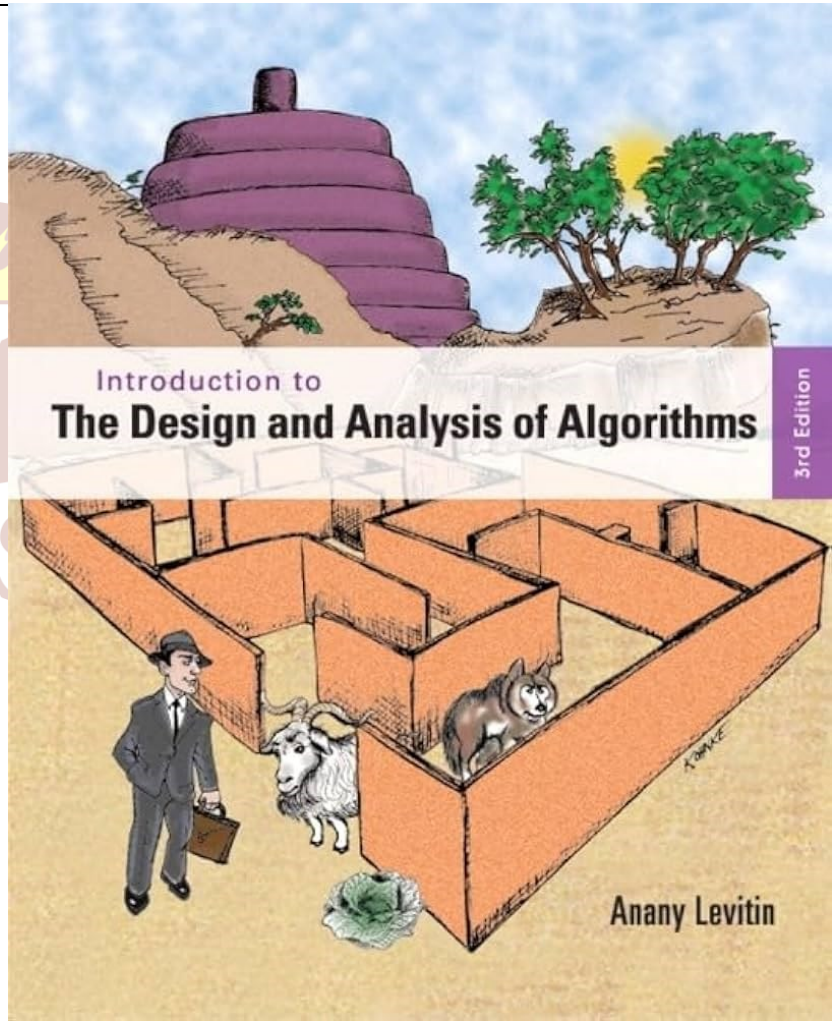
CS 07340 Design and Analysis of Algorithms

Week 3 Overview

- Brute Force and Exhaustive Search
  - Brute Force / Exhaustive Search
  - Depth-First Search
  - Breadth-First Search



Brute Force and Exhaustive Search

Brute-force search or ***exhaustive search*** is a problem-solving technique which consists of systematically checking all possible solution candidates for whether they satisfy the desired properties.

Examples of exhaustive search algorithms we will look at are:

- Selection Sort (done before)
- Bubble Sort
- Traveling Salesman Problem
- Knapsack Problem
- Depth-First Search
- Breadth-First Search

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 nondecreasing 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]
```


As we have already determined earlier, *Selection Sort* is $\Theta(n^2)$.

The analysis of selection sort is straightforward. The input size is given by the number of elements n; the basic operation is the key comparison $A[j] < A[\min]$. The number of times it is executed depends only on the array size and is given by the following sum (where $C(n)$ is the runtime cost function):

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

$$= \sum_{i=0}^{n-2} (n-1-i)$$

$$= \frac{1}{2}(n-1)n$$

BUBBLE SORT

Bubble Sort compares adjacent elements of our list and exchanges them if they are out of order. Repeated application lets the larger elements "bubble" to later positions, eventually ordering the list.

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 nondecreasing order
for i ← 0 to n − 2 do
    for j ← 0 to n − 2 − i do
        if A[j + 1] < A[j] swap A[j] and A[j + 1]
```


We can do an analysis similar to *Selection Sort*.

$$
\begin{aligned}
C(n) &= \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 \\
&= \sum_{i=0}^{n-2} (n-1-i) \\
&= \frac{1}{2}(n-1)n
\end{aligned}
$$

In worst case, we have to do $\Theta(n^2)$ swaps, but even in best case, we need $\Theta(n^2)$ iterations in our double loop. Hence *Bubble Sort* is $O(n^2)$ and $\Theta(n^2)$.

Example: Design a brute-force algorithm for computing the value of a polynomial $p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$ at a given point x_0 and determine its worst-case efficiency class.

Note that an optimal solution to this problem is Horner's Method, which is $\Theta(n)$. Its main idea is also used in the ***Discrete Fast Fourier Transform (DFFT)*** and ***fast exponentiation*** (exponentiation by squaring). We will see these algorithms in ***Transform-and-Conquer***.

Example: Show that brute force pattern matching is $O(nm)$ for strings of length n and patterns of length m (where $m \ll n$).

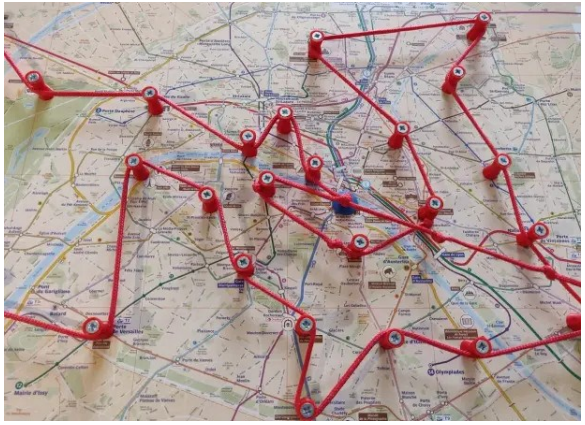
ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

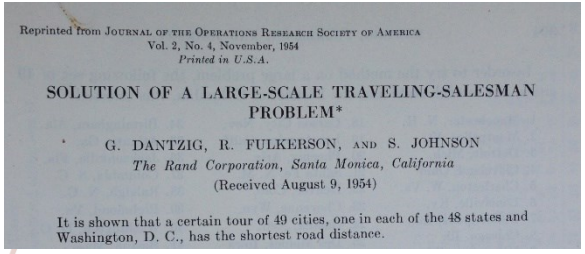
```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//       an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//        matching substring or −1 if the search 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
```

Exhaustive search is simply a brute-force approach to combinatorial problems. It suggests generating each element of the problem domain, selecting those of them that satisfy all the constraints, and then finding a desired element (the one that optimizes some objective function). Note that although the idea of exhaustive search is quite straightforward, its implementation typically requires an algorithm for generating certain combinatorial objects. We illustrate exhaustive search by applying it to three important problems: the traveling salesman problem (NP hard), the knapsack problem (NP hard), and the assignment problem ($O(mn + n^2 \log(n))$).

Traveling Salesman Problem

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?





Reprinted from JOURNAL OF THE OPERATIONS RESEARCH SOCIETY OF AMERICA
Vol. 2, No. 4, November, 1954
Printed in U.S.A.

SOLUTION OF A LARGE-SCALE TRAVELING-SALESMAN PROBLEM*

G. DANTZIG, R. FULKERSON, AND S. JOHNSON
The Rand Corporation, Santa Monica, California
(Received August 9, 1954)

It is shown that a certain tour of 49 cities, one in each of the 48 states and Washington, D. C., has the shortest road distance.

This problem is surprisingly hard, but it is an important logistical problem to solve.

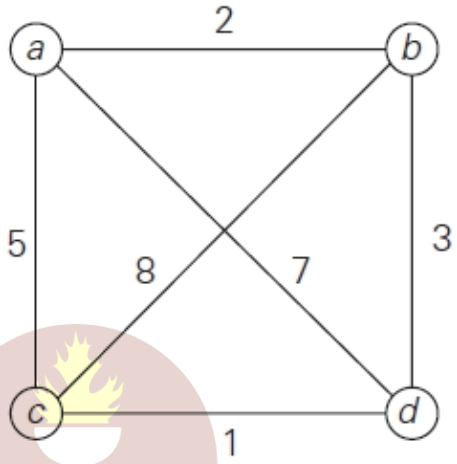
The ***Traveling Salesman Problem (TSP)*** can be modeled as a weighted graph. It is a minimization problem starting and finishing at a specified vertex after having visited each other vertex exactly once.

Newsweek—Bensi

FINDING the shortest route for a traveling salesman—starting from a given city, visiting each of a series of other cities, and then returning to his original point of departure—is more than an after-dinner teaser. For years it has baffled not only goods- and salesmen-routing businessmen but mathematicians as well. If a drummer visits 50 cities, for example, he has 10^{62} (62 zeros) possible itineraries. No electronic computer in existence could sort out such a large number of routes and find the shortest.

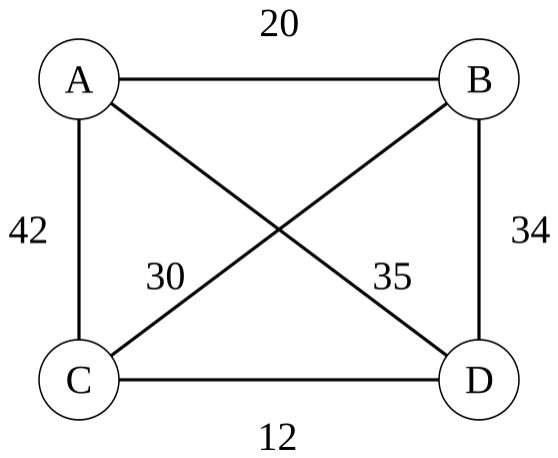
Three Rand Corp. mathematicians, using Rand McNally road-map distances between the District of Columbia and major cities in each of the 48 states, have finally produced a solution (see above). By an ingenious application of linear programming—a mathematical tool recently used to solve production-scheduling problems—it took only a few weeks for the California experts to calculate "by hand" the shortest route to cover the 49 cities: 12,345 miles.

An example from our textbook.

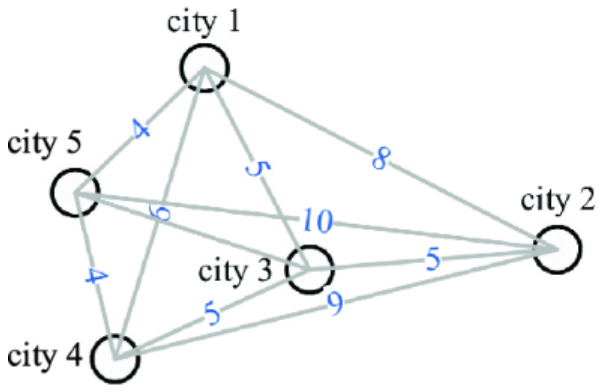


Tour	Length	
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	$l = 2 + 8 + 1 + 7 = 18$	
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	$l = 2 + 3 + 1 + 5 = 11$	optimal
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	$l = 5 + 8 + 3 + 7 = 23$	
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	$l = 5 + 1 + 3 + 2 = 11$	optimal
$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	$l = 7 + 3 + 8 + 5 = 23$	
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	$l = 7 + 1 + 8 + 2 = 18$	

A brute force technique for the TSP is to try all possible sequences of cities. In the first example, we have four cities. That makes for $4! = 24$ different travels. We will have to test all 24 travel routes, beginning with A→B→C→D→A and find the shortest travel as sum of the edge weights.



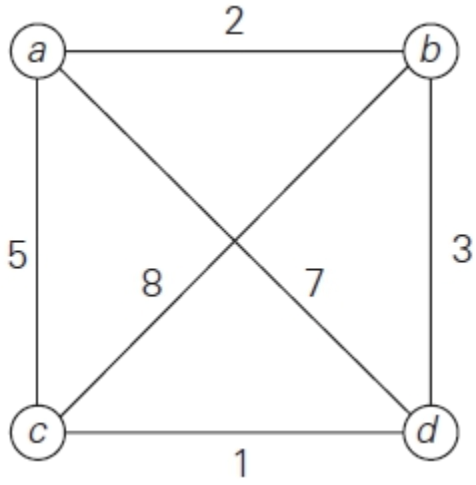
The second example is directed, so the associated matrix is not symmetric.



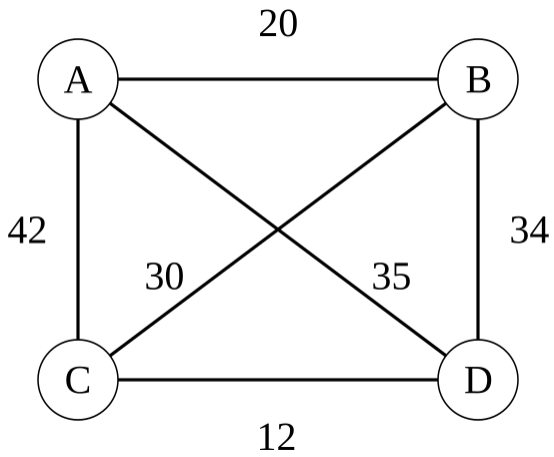
	city 1	city 2	city 3	city 4	city 5
city 1	0	8	5	6	4
city 2	8	0	5	9	10
city 3	5	5	0	5	5
city 4	6	9	5	0	4
city 5	4	10	5	4	0

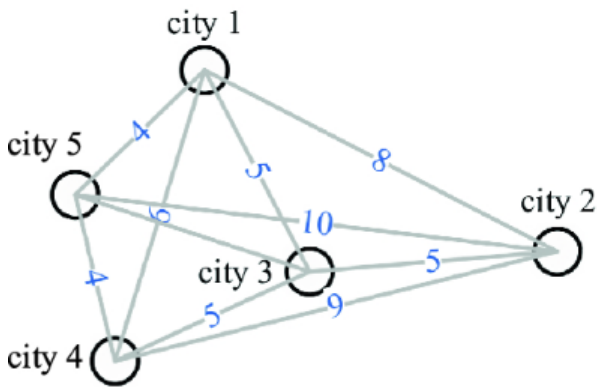
distance matrix

Example: Find an optimal solution to the TSP for the given graph by hand. Note that knowing it is optimal means checking all possible pathways.



Example (in-class): Program (brute force) an optimal solution to the TSP for the given graphs.





	city 1	city 2	city 3	city 4	city 5
city 1	0	8	5	6	4
city 2	8	0	5	9	10
city 3	5	5	0	5	5
city 4	6	9	5	0	4
city 5	4	10	5	4	0

distance matrix

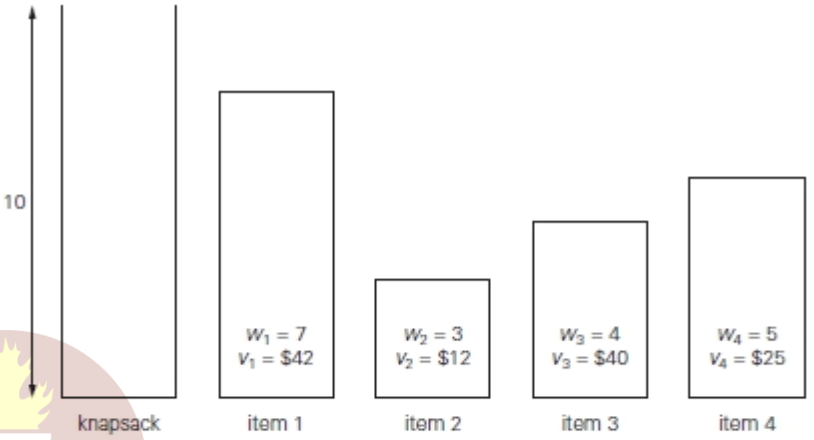
```java
import java.util.ArrayList;
import java.util.List;
import java.util.Arrays;
import java.util.stream.IntStream;

public class HeapsPermutation {
    public static void swap(int[] A, int i, int j) {
        A[i] = (A[i] + A[j]) - (A[j] = A[i]);
    }
    static List allPermutations(Integer n) {
        int[] A = IntStream.rangeClosed(0, n-1).toArray();
        List Permutations = new ArrayList<ArrayList<Integer>>();
        int[] C = new int[n];
        // initialize stack encoding recursion/position of next transposition
        Arrays.fill(C, 0);
        // store initial permutation as List
        // in typical complicated Java way
        Permutations.add(Arrays.stream(A).boxed().toList());
        int i = 1;
        while(i < n) {
            if(C[i] < i) {
                if(i%2 == 0)
                    swap(A, 0, i);
                else
                    swap(A, C[i], i);
                Permutations.add(Arrays.stream(A).boxed().toList());
                C[i] += 1;
                i = 1;
            }
            else {
                C[i] = 0;
                i += 1;
            }
        }
        return Permutations;
    }
}
```

KNAPSACK PROBLEM

Given n items of known weights $w_1, w_2, \dots, w_n$ with values $v_1, v_2, \dots, v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.

The exhaustive-search approach to this problem leads to generating all the subsets of the set of n items given, computing the total weight of each subset in order to identify feasible subsets, and finding a subset of the largest value among them. It is another optimization function. Since the number of subsets of an n-element set is 2^n, the exhaustive search leads to a $\Omega(2^n)$ algorithm, no matter how efficiently individual subsets are generated.

The *Traveling Salesman Problem* and the *Knapsack Problem* are the best-known examples of so-called NP-hard problems. No polynomial-time algorithm is known for any NP-hard problem. Moreover, most computer scientists believe that such algorithms do not exist, although this very important conjecture has never been proven.

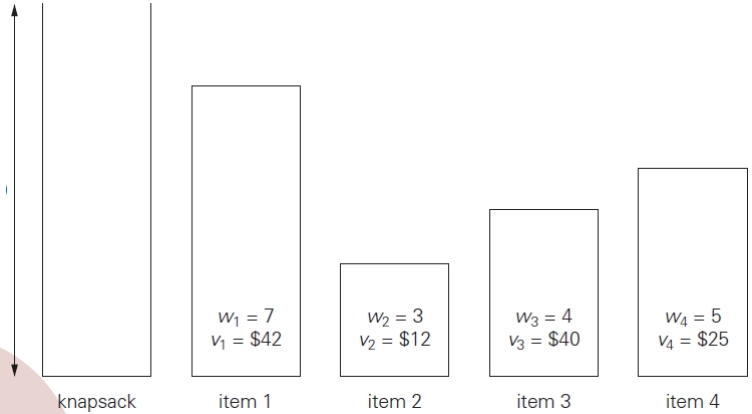


(a)

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

(b)

Example: Verify the solution to our book's knapsack problem manually.



If we want to use a brute-force approach, we need a way to create all subsets of items. How many are there? What is an efficient way to create all subsets of four given items?

```java
import java.util.ArrayList;
import java.util.List;

public class Subsets {
    public static List<List<Integer>> allSubsets(Integer n) {
        List<List<Integer>> Subsets = new ArrayList<>();
        for(int indicatorVector = 0; indicatorVector < (2<<n); indicatorVector++) {
            // check bits as means of membership in a set
            List<Integer> Subset = new ArrayList<>();
            for(Integer index = 0; index < n; index++) {
                int bitmask = (1<<index);
                if((indicatorVector & bitmask) != 0)
                    Subset.add(index);
            }
            Subsets.add(Subset);
        }
        return Subsets;
    }
}
```

This shows that creating subsets is $O(2^n)$. Evaluating the weight and then the objective function (sum) is at least $O(n)$ for each subset. The Knapsack problem is NP-complete.

The Assignment Problem

There are n people who need to be assigned to execute n jobs, one person per job. The cost that would accrue if the ith person is assigned to the jth job is a known quantity $C[i,j]$ for each pair $i,j = 1,2,\ldots,n$. The problem is to find an assignment with the minimum total cost. This problem can be solved in polynomial time.

A variation of this problem is the Stable Marriage Problem.

Example: Find an optimal solution to this assignment problem by hand.

	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

What tools do we need for a brute force/exhaustive search solution?

Since the number of permutations to be considered for the general case of the assignment problem is $n!$ exhaustive search is impractical for all but very small instances of the problem. There is a much more efficient algorithm for this problem called the Hungarian method after the Hungarian mathematicians König and Egerváry in $O(n^4)$, which later was improved to $O(n^3)$.

Depth-First Search

Depth-first search (***DFS***) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. Extra memory, usually a stack, is needed to keep track of the nodes discovered so far along a specified branch which helps in backtracking of the graph.

Representations of Graphs

There are two standard ways to represent a graph $G = (V, E)$:

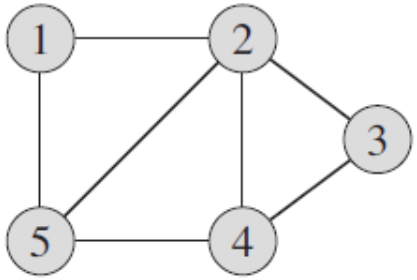
- a collection of adjacency lists
- an adjacency matrix.

The adjacency-list representation provides a compact way to represent sparse graphs. For denser graphs the matrix representation is better.

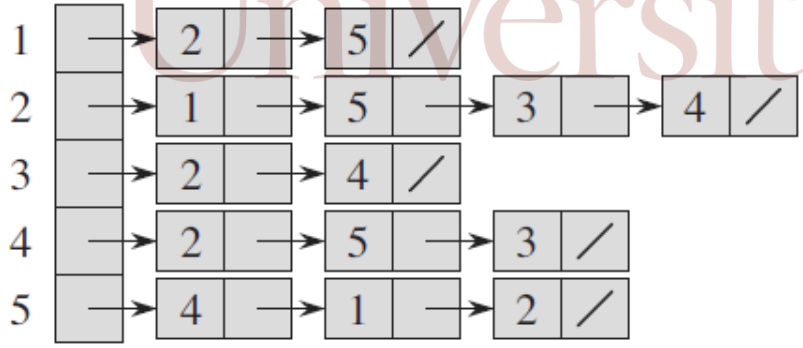
$$A[i,j] = \begin{cases} 0 & \text{if } (i,j) \in E \\ 1 & \text{if } (i,j) \notin E \end{cases}$$

We could use non-binary representations to implement “weight” or “capacity”, or even encode other properties. Another way of storing a graph would be as an edge list.

Here are three representations of the same graph as picture, list of adjacency lists, and adjacency matrix.



(a)



(b)

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

ALGORITHM *DFS*(G)

//Implements a depth-first search traversal of a given graph
//Input: Graph $G = \langle V, E \rangle$
//Output: Graph G with its vertices marked with consecutive integers
// in the order they are first encountered by the DFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
$count \leftarrow 0$
for each vertex v in V **do**
  if v is marked with 0
    dfs(v)

dfs(v)
//visits recursively all the unvisited vertices connected to vertex v
//by a path and numbers them in the order they are encountered
//via global variable *count*
$count \leftarrow count + 1$; mark v with *count*
for each vertex w in V adjacent to v **do**
  if w is marked with 0
    dfs(w)

DFS explore edges out of the most recently discovered vertex v that still has unexplored edges leaving it. Once all of v's edges have been explored, we "backtrack" to explore edges leaving the vertex from which v was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex.

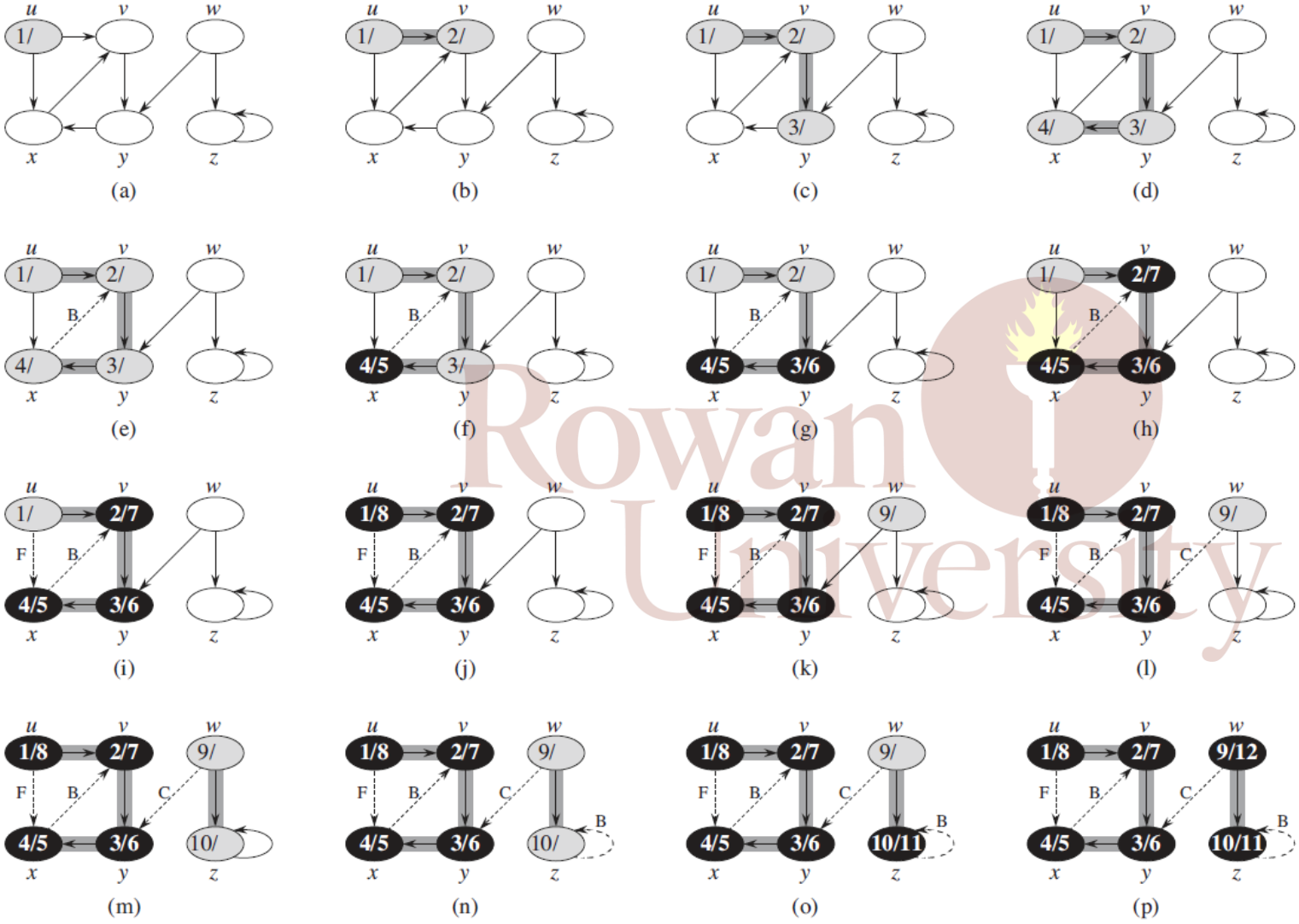
If any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source. The algorithm repeats this entire process until it has discovered every vertex.

In depth-first search we may color vertices during the search to indicate their state. Each vertex is:

- White – initially
- Grey – when it is discovered
- Black – when it is finished, i.e., adjacency list has been examined completely.

An alternate, colored version of the algorithm highlights the visiting progress. It guarantees (visually and programmatically) that each vertex ends up in exactly one depth-first tree, so that these trees are disjoint.

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
 1  time = time + 1            // white vertex u has just been discovered
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]      // explore edge (u, v)
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK            // blacken u; it is finished
 9  time = time + 1
10  u.f = time
```



DFS Analysis

The loops over vertices take $\theta(V)$, excluding the time to execute the calls to DFS. The procedure DFS is called exactly once for each vertex v in V, since the vertex w on which DFS is invoked must be white and the first thing DFS does is paint vertex $w$gray. During an execution of DFS the loop over adjacent vertices w is executes $|Adj(v)|$ times. The sum over adjacent vertices on average the number of edges (double-counted).

$$\sum_{v \in V} |Adj[v]| = \Theta(E)$$

Hence the total time to execute adjacent vertices is $\theta(E)$ and the total running-time of DFS is $\theta(E + V)$.

Breadth-First Search

Breadth-First Search (BFS) proceeds in a concentric manner by visiting first all the vertices that are adjacent to a starting vertex, then all unvisited vertices two edges apart from it, and so on, until all the vertices in the same connected component as the starting vertex are visited.

If there remain unvisited vertices, the algorithm must be restarted at an arbitrary vertex of another connected component of the graph.

It is convenient to use a queue to trace the operation of breadth-first search.

Given a graph G and a distinguished source vertex s, breadth-first search (BFS) systematically explores the edges of G to "discover" every vertex that is reachable from s. Breadth-first search expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. The algorithm discovers all vertices at distance k from s before discovering any vertices at distance $k + 1$.

ALGORITHM *BFS*(G)

```
//Implements a breadth-first search traversal of a given graph
//Input: Graph G = ⟨V, E⟩
//Output: Graph G with its vertices marked with consecutive integers
//        in the order they are visited by the BFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each vertex v in V do
    if v is marked with 0
        bfs(v)

bfs(v)
//visits all the unvisited vertices connected to vertex v
//by a path and numbers them in the order they are visited
//via global variable count
count ← count + 1;   mark v with count and initialize a queue with v
while the queue is not empty do
    for each vertex w in V adjacent to the front vertex do
        if w is marked with 0
            count ← count + 1;   mark w with count
            add w to the queue
    remove the front vertex from the queue
```

BFS computes the distance (smallest number of edges) from s to each reachable vertex. It also produces a "breadth-first tree" with root s that contains all reachable vertices.

For any vertex reachable from s, the simple path in the breadth-first tree from s to v corresponds to a "shortest path" from s to v in G, that is, a path containing the smallest number of edges. The algorithm works on both directed and undirected graphs.

Again, we can use a BFS coloring algorithm to keep track of progress:

- White – undiscovered vertex
- Gray – discovered vertex connected to other discovered and undiscovered vertices (white or grey)
- Black – discovered vertex connected only to other discovered vertices (grey or black)

All vertices start out white and may later become gray and then black.

A vertex is discovered the first time it is encountered during the search, at which time it becomes nonwhite (i.e., grey or black).

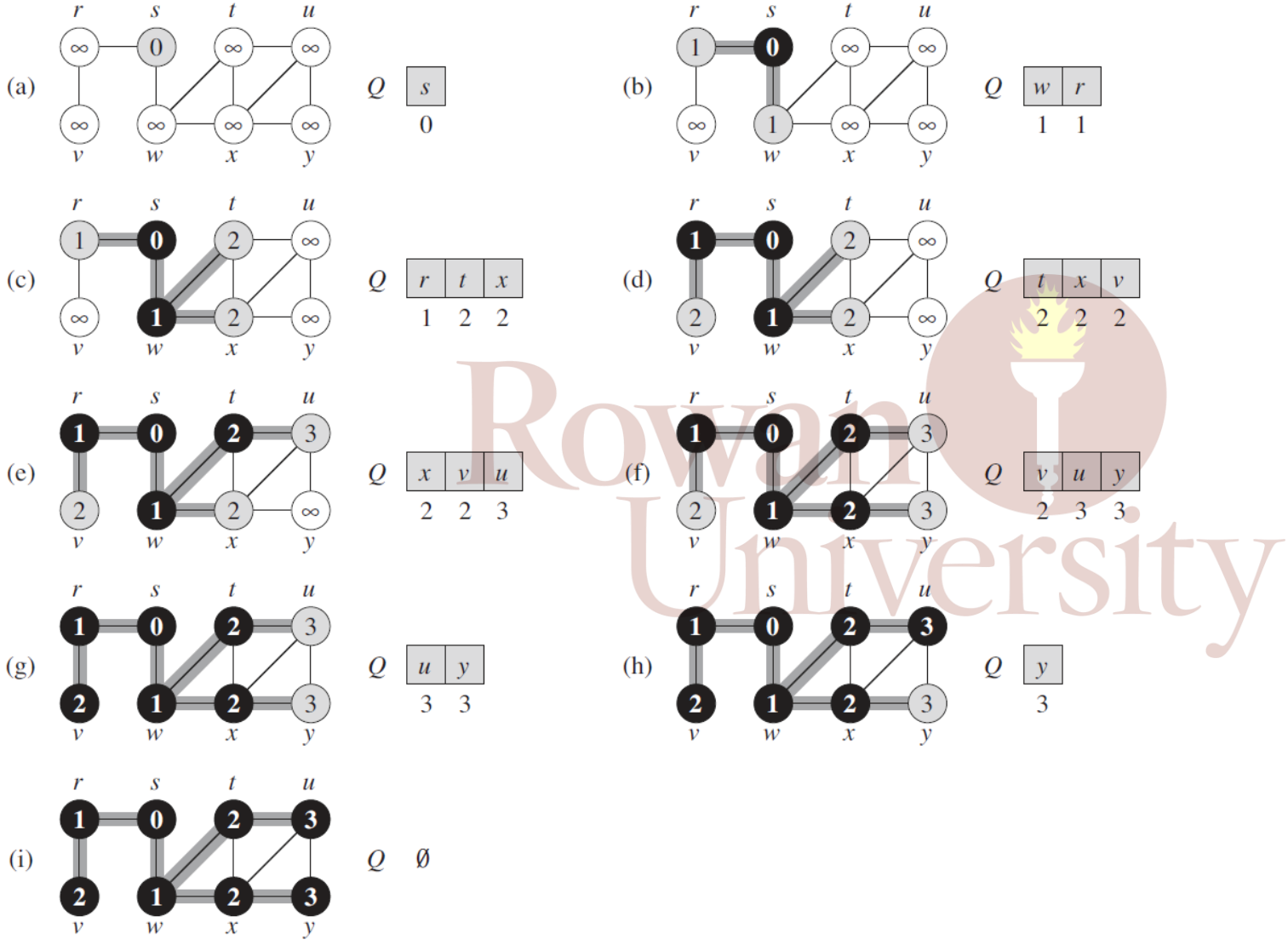
BFS constructs a breadth-first tree, starting with the root, which is the source vertex s.

When BFS discovers a white vertex v in the course of scanning the adjacency list of an already discovered vertex u, the vertex v and the edge (u, v) are added to the tree.

u is the predecessor or parent of v in the breadth-first tree. Since a vertex is discovered at most once, it has at most one parent.

Ancestor and descendant relationships in the breadth-first tree are defined relative to the root s as usual: if u is on the simple path in the tree from the root s to vertex v, then u is an ancestor of v and v is a descendant of u.

```
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```



BFS Analysis

After initialization, breadth-first search never whitens a vertex, and thus each vertex is enqueued at most once, and dequeued at most once.

The operations of enqueuing and dequeuing take $O(1)$ time, and so the total time devoted to queue operations is $O(V)$. Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once.

Since the sum of the lengths of all the adjacency lists is $\Theta(E)$, the total time spent in scanning adjacency lists is $O(E)$. The overhead for initialization is $O(V)$, and thus the total running time of the BFS procedure is $O(V + E)$. Thus, breadth-first search runs in time linear in the size of the adjacency-list representation of G.

During its execution, BFS discovers every vertex that is reachable from the source s. Upon termination, it has recorded the distance of every vertex from the source, and a shortest path can be reconstructed.

Quiz Topics

- Programming paradigms
- Algorithm complexity for standard/famous problems