Algorithm Analysis - Review

- Definition: An algorithm is a sequence of steps used to solve a problem in a finite amount of time

- Properties
  - Correctness: must provide the correct output for *every* input
  - Performance: Measured in terms of the resources used (time and space)
  - End: must finish in a *finite* amount of time

Input size

- Performance of an algorithm measured in terms of the input size
- Examples:
  - Number of elements in a list or array A: n

A

21	22	23	25	24	23	22
0	1	2	3	4	5	6

  - Number of cells in an mxn matrix: m, n
  - Number of bits in an integer: n
  - Number of nodes in a tree: n
  - Number of vertices and edges in a graph: $|V|$, $|E|$

Experimental vs Theoretical analysis

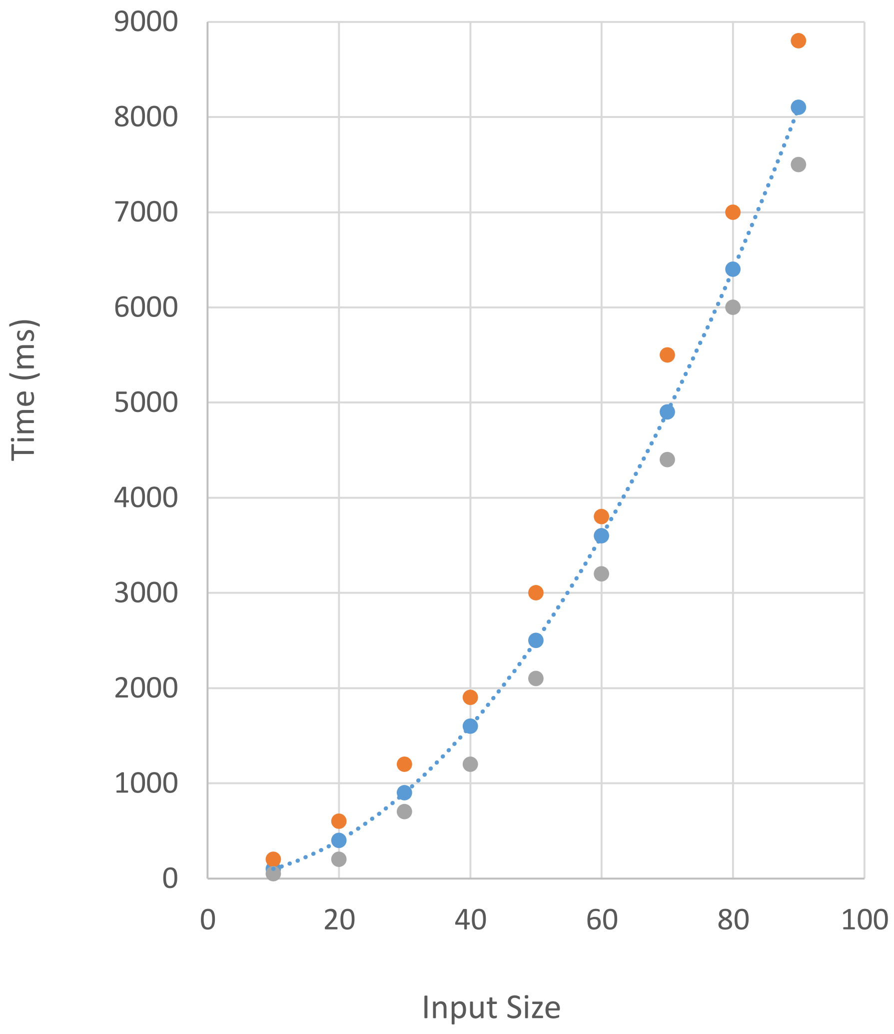
University of Windsor

Experimental analysis:

- Write a program that implements the algorithm
- Run the program with inputs of varying size and composition
- Keep track of the CPU time used by the program on each input size
- Plot the results on a two-dimensional plot

Limitations:

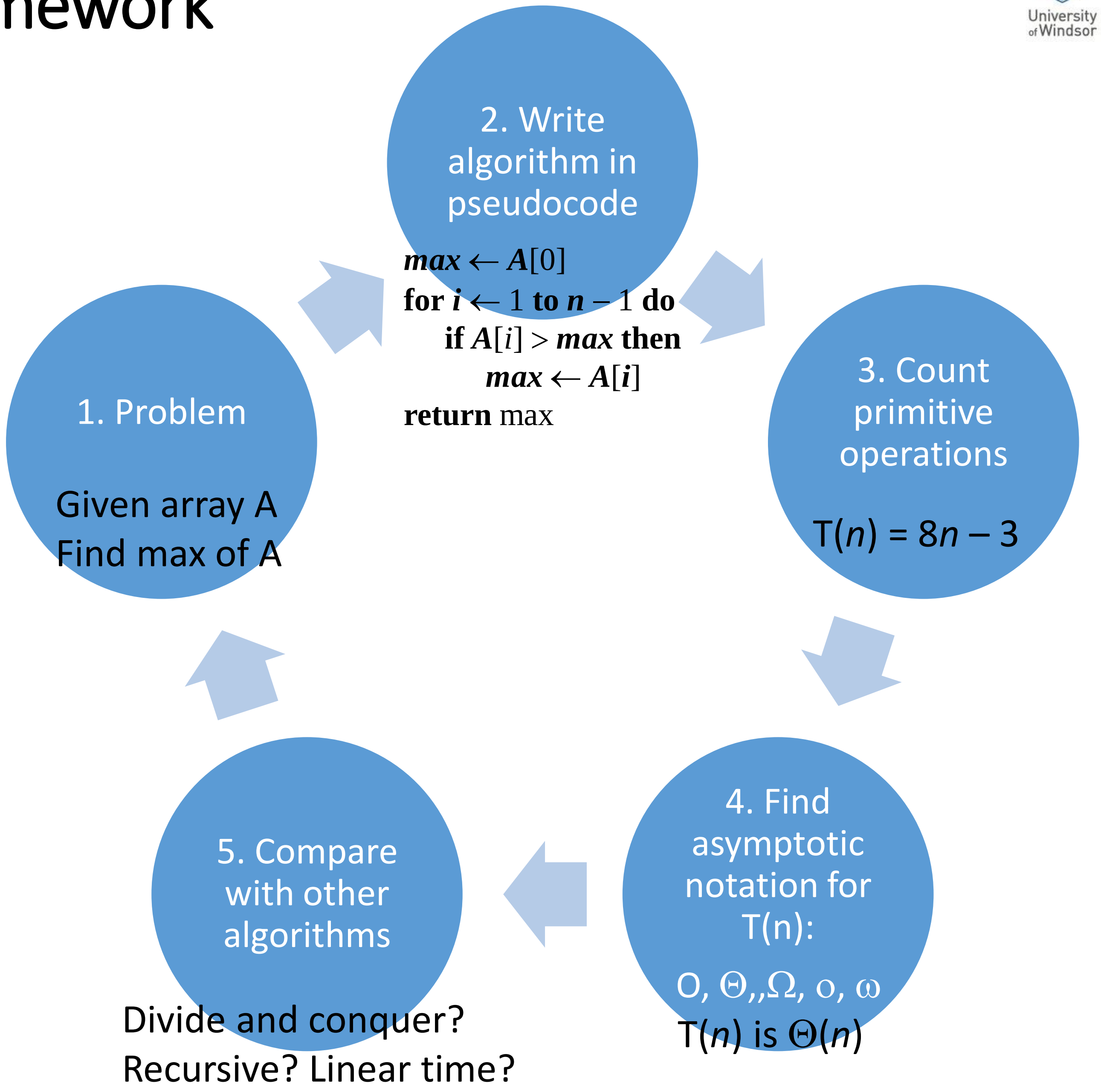
- Depends on hardware and programming language
- Need to implement the algorithm and debug the programs



Theoretical analysis – main framework

Advantages:

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, *n*.
- Takes into account *all* possible inputs
- Allows us to evaluate the speed of an algorithm independently of the hardware/software environment



Pseudocode

- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces
- Method declaration

  Algorithm *method* (*arg* [, *arg*…])
  Input …
  Output …
  return …

- Method call

  var.method (*arg* [, *arg*…])

- Return value

  return *expression*

- Expressions

  $\leftarrow$ Assignment (like $=$ in Java)

  $=$ Equality testing (like $==$ in Java)

  n^2 Superscripts and other mathematical formatting allowed

Example: find max element of an array

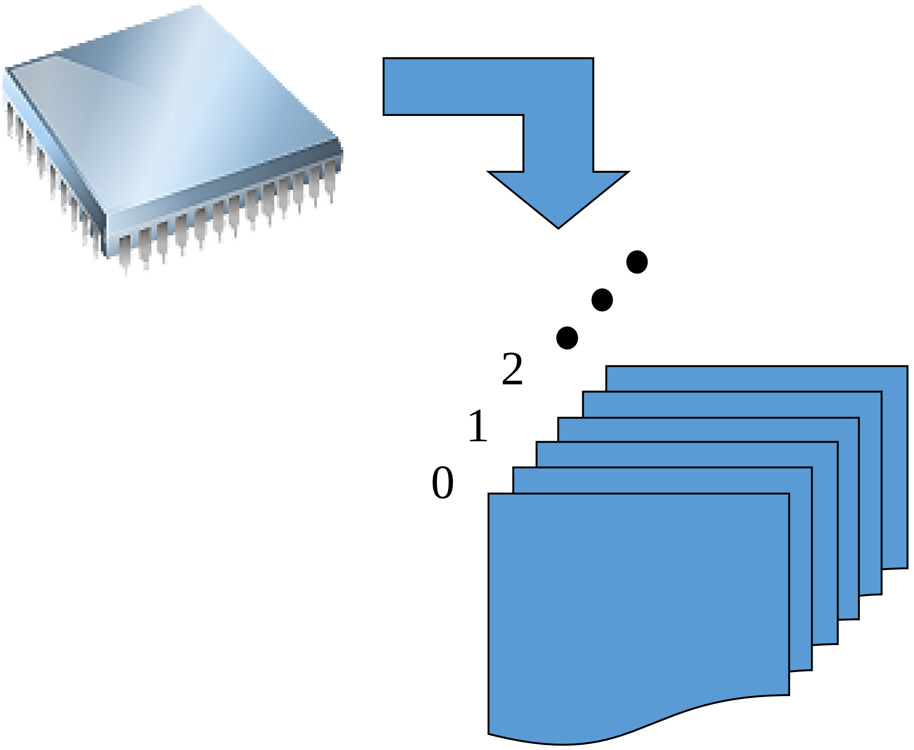
Algorithm *arrayMax*(A, n)
  Input array A of n integers
  Output maximum element of A

  $currentMax \leftarrow A[0]$
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    if $A[i] > currentMax$ **then**
      $currentMax \leftarrow A[i]$
  return $currentMax$

Pseudocode provides a high-level description of an algorithm and avoids to show details that are unnecessary for the analysis.

Random Access Machine (RAM)

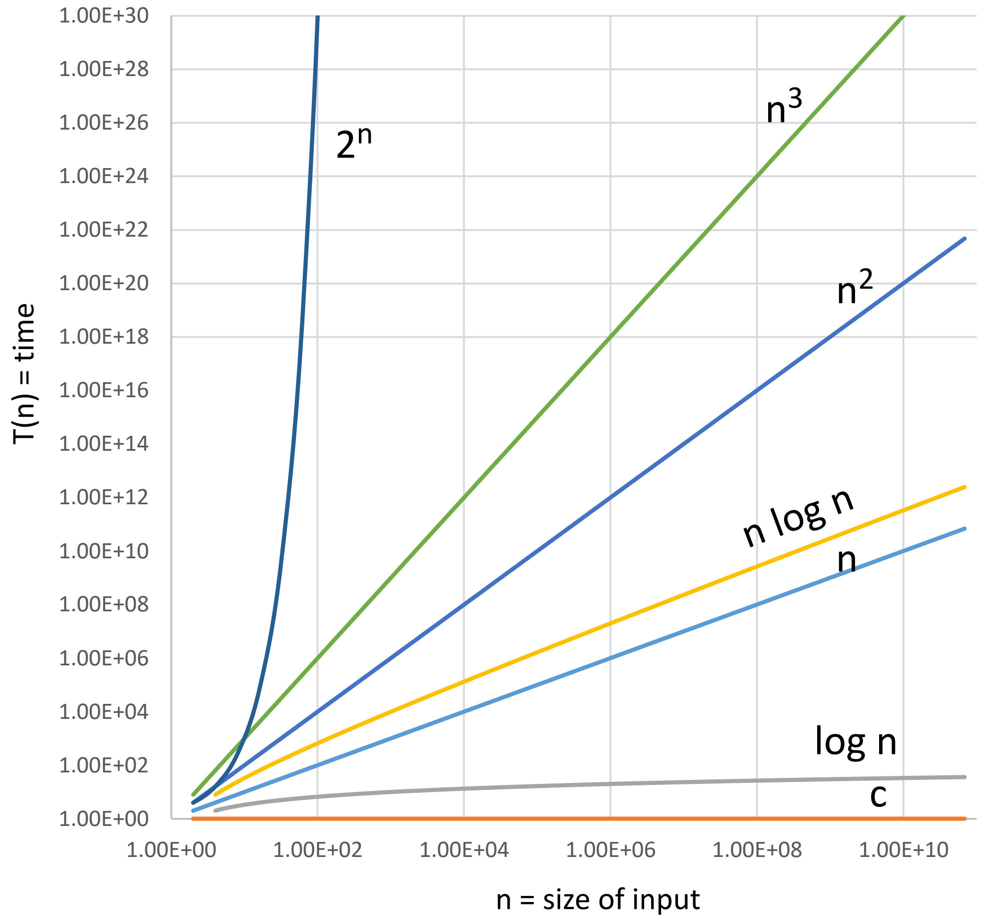
- It has a **CPU** – equivalent to that of a conventional computer
- A potentially unbounded bank of **memory** cells
- Each memory cell:
  - can hold an arbitrary number or character
  - is referenced by a number or index
  - Can be accessed in unit time



Most important functions used in Algorithm Analysis

- The following functions often appear in algorithm analysis:
  - Constant $\approx \boldsymbol{1}$ (or $\boldsymbol{c}$)
  - Logarithmic $\approx \log \boldsymbol{n}$
  - Linear $\approx \boldsymbol{n}$
  - N-Log-N $\approx \boldsymbol{n} \log \boldsymbol{n}$
  - Quadratic $\approx \boldsymbol{n}^2$
  - Cubic $\approx \boldsymbol{n}^3$
  - Exponential $\approx \mathbf{2}^n$

- In a log-log chart, the slope of the line corresponds to the growth rate of the function (except exponential)



Primitive operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent of the programming language
- Exact definition not important (we will see why later)
- Take a constant amount of time in the RAM model (one unit of time or constant time)

Examples:

- Evaluating an expression

e.g. $a - 5 + c\sqrt{b}$

- Assigning a value to a variable

e.g. $a \leftarrow 23$

- Indexing into an array

e.g. $A[i]$

- Calling a method

e.g. *v.method()*

- Returning from a method

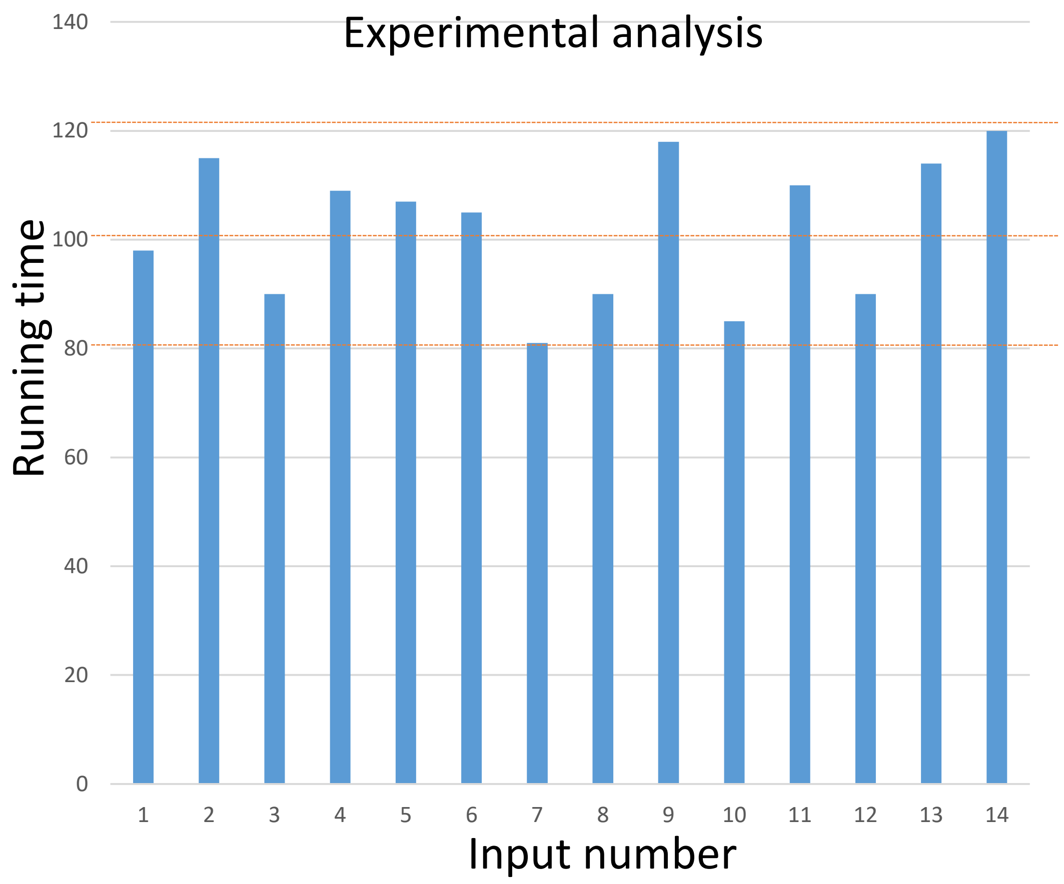
e.g. *return a*

Counting primitive operations:

Algorithm ***arrayMax***(***A***, ***n***)	# operations
currentMax ← ***A***[0]	2
for *i* ← 1 **to** *n* – 1 **do**	2***n***
if *A*[*i*] > *currentMax* **then**	2(***n*** – 1)
currentMax ← *A*[*i*]	2(***n*** – 1)
{ increment counter *i* }	2(***n*** – 1)
return *currentMax*	1

Total: $T(\boldsymbol{n}) = 8\boldsymbol{n} - 3$

Case analysis



Three cases

- Worst case:
  - among all possible inputs, the one which takes the largest amount of time.
- Best case:
  - The input for which the algorithm runs the fastest
- Average case:
  - The average is over all possible inputs
  - Can be considered as the expected value of T(n), which is a random variable

Asymptotic notation

Name	Notation /use	Informal name	Bound	Notes
Big-Oh	$O(n)$	order of	Upper bound – tight	The most commonly used notation for assessing the complexity of an algorithm
Big-Theta	$\Theta(n)$		Upper and lower bound – tight	The most accurate asymptotic notation
Big-Omega	$\Omega(n)$		Lower bound – tight	Mostly used for determining lower bounds on problems rather than algorithms (e.g., sorting)
Little-Oh	$o(n)$		Upper bound – loose	Used when it is difficult to obtain a tight upper bound
Little-Omega	$\omega(n)$		Lower bound – loose	Used when it is difficult to obtain a tight lower bound

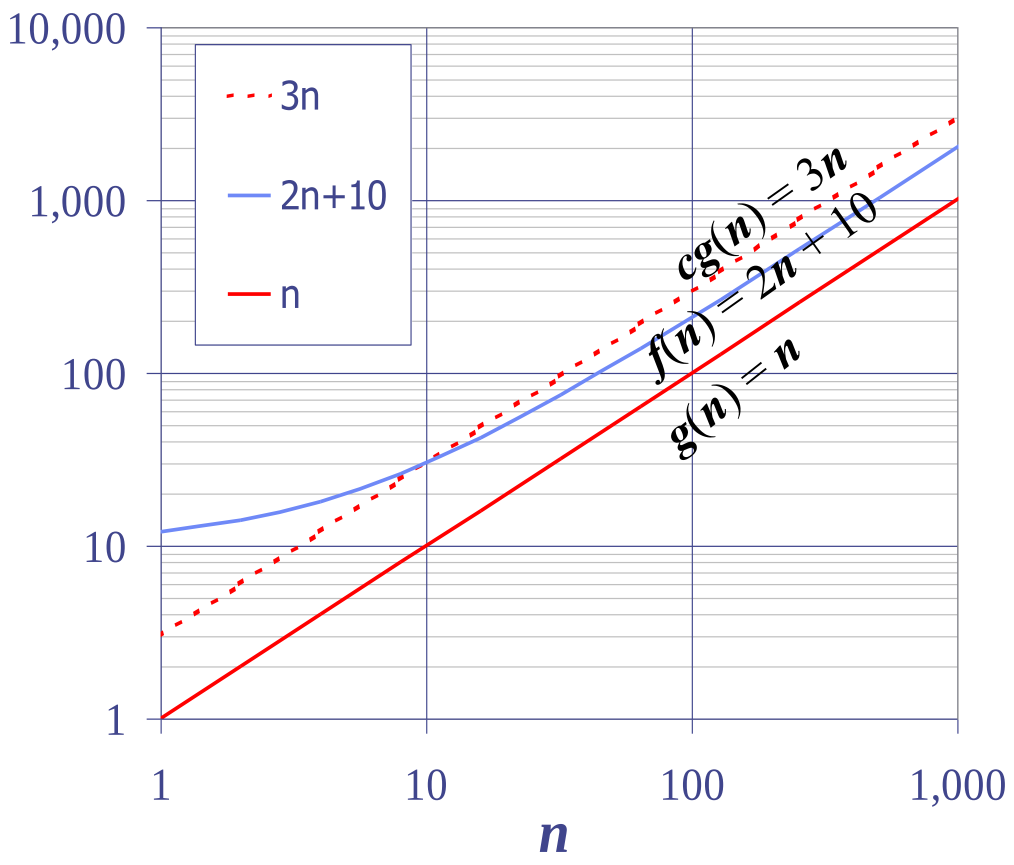
Big-Oh notation

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

  $$f(n) \le cg(n) \quad \text{for } n \ge n_0$$

- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \le cn$
  - $(c - 2)\, n \ge 10$
  - $n \ge 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$
  - It follows that

  $$2n + 10 \le 3n \text{ for } n \ge 10$$



Big-Oh rules - properties

- The statement "***f***(***n***) is ***O***(***g***(***n***))" means that the growth rate of ***f***(***n***) is no more than the growth rate of ***g***(***n***)
- ***O***(***g***(***n***)) is a *set* or *class* of functions: it contains all the functions that have the *same growth rate*
- If ***f***(***n***) is a polynomial of degree ***d***, then ***f***(***n***) is ***O***(***n^d^***)
  - If ***d*** **= 0**, then ***f***(***n***) is ***O***(**1**)
  - Example: $n^2 + 3n - 1$ is $\boldsymbol{O(n^2)}$
- We always use the *simplest* expression of the class/set
  - E.g., we state 2n + 3 is O(n) instead of O(4n) or O(3n+1)
- We always use the *smallest* possible class/set of functions
  - E.g., we state 2n is O(n) instead of $O(n^2)$ or $O(n^3)$
- Linearity of asymptotic notation
  - O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(max{f(n),g(n)}), where "max" is wrt "growth rate"
  - Example: $O(n) + O(n^2) = O(n + n^2) = O(n^2)$

Big-Omega and Big-Theta notations

- **big-Omega**
  - f(n) is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c\, g(n)$ for $n \geq n_0$

  Example: $3n^3 - 2n + 1$ is $\Omega(n^3)$
- **big-Theta**
  - f(n) is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that $c'\, g(n) \leq f(n) \leq c''\, g(n)$ for $n \geq n_0$
  - Example: $5n \log n - 2n$ is $\Theta(n \log n)$

- Important axiom:
  - f(n) is $O(g(n))$ and $\Omega(g(n)) \Leftrightarrow$ f(n) is $\Theta(g(n)$
  - Example: $5n^2$ is $O(n^2)$ and $\Omega(n^2) \Leftrightarrow 5n^2$ is $\Theta(n^2)$

Asymptotic notation – graphical comparison

Big-Oh

- f(n) is O(g(n)) if f(n) is asymptotically **less than or equal** to g(n)

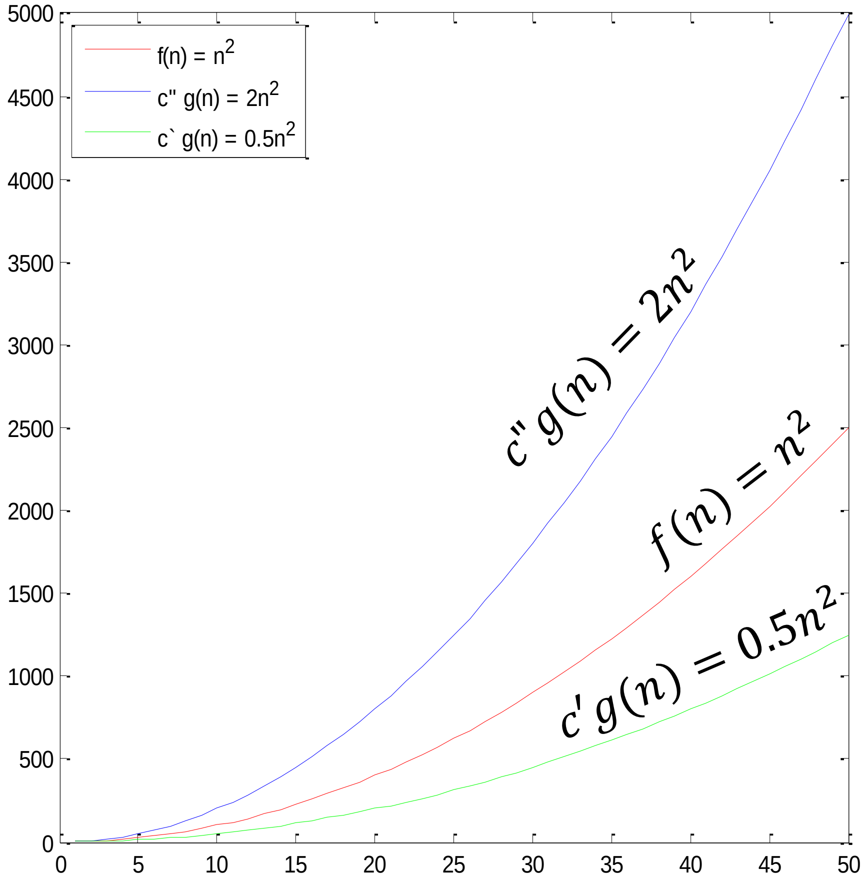
Big-Omega

- f(n) is Ω(g(n)) if f(n) is asymptotically **greater than or equal** to g(n)

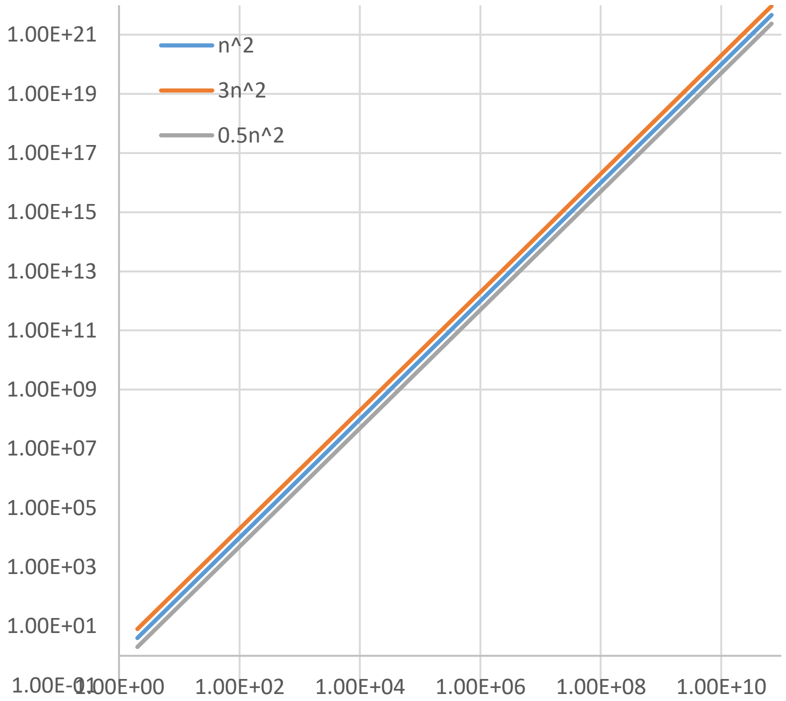
Big-Theta

- f(n) is Θ(g(n)) if f(n) is asymptotically **equal** to g(n)

Normal scale:



Log-log scale:



Little-Oh and Little-Omega notations

- **Little-Oh**
  - f(n) is o(g(n)) if *for any* constant $c > 0$, there is a constant $n_0 > 0$ such that $f(n) < c\, g(n)$ for $n \geq n_0$

  Example: $3n^2 - 2n + 1$ is $o(n^3)$, while $3n^2 - 2n + 1$ **is not** $o(n^2)$
- **Little-Omega**
  - f(n) is ω(g(n)) if for any constant $c > 0$, there is a constant $n_0 > 0$ such that $f(n) > c\, g(n)$ for $n \geq n_0$
  - Example: $3n^2 - 2n + 1$ is $\omega(n)$, while $3n^2 - 2n + 1$ **is not** $\omega(n^2)$

Important axiom:

$$f(n) \text{ is } o(g(n)) \Leftrightarrow g(n) \text{ is } \omega(f(n))$$

- Comparison with O and Ω
  - For O and Ω, the inequality holds if **there exists** a constant $c > 0$
  - For o and ω , the inequality holds **for all** constants $c > 0$

Case study 1: Search in a Map (sorted list)

- Problem: Given a sorted array S of integers (a map), find a key k in that map.
- One of the most important problems in computer science
- Solution 1: Linear search
  - Scan the elements in the list one by one
  - Until the key k is found
- Example:

S[i]	**8**	**12**	**19**	**22**	**23**	**34**	**41**	**48**
i	0	1	2	3	4	5	6	7

- Linear search runs in *linear* time.

Algorithm linearSearch(S, *k*, *n*):

Input: Sorted array S of size *n*, and key *k*

Output: Null or the element found

$i \leftarrow 0$

while $i < n$ **and** $S[i] != k$

  $i \leftarrow i + 1$

if $i = n$ **then**

  return null

else

  $e \leftarrow S[i]$

  return *e*

Worst-case running time: T(n) = 3n + 4 ➔ T(n) is O(n)

Case study 1: Search in a Map (sorted list)

- Problem: Given a sorted array of integers (a map), find a key k in that map.
- Solution 2: Binary search
- Binary search runs in *logarithmic* time
- Same problem:
  - Two algorithms run in different times

Algorithm binarySearch(S, *k*, low, high):
Input: A key *k*
Output: Null or the element found
if low > high **then**
        return null
else
    mid $\leftarrow \lfloor$(low + high) / 2$\rfloor$
    e $\leftarrow$ S[mid]
    if *k* = *e*.getKey() **then**
        return *e*
    else if *k* < *e*.getKey() **then**
            return binarySearch(S, *k*, low, mid-1)
        else
            return binarySearch(S, *k*, mid+1, high)

S[*i*]	8	12	19	22	23	34	41	48
i	0	1	2	3	4	5	6	7

Worst-case running time: T(n) = T(n/2) + 1 ➔ T(n) is O(log n)

Case study 2: Prefix averages

- The $\boldsymbol{i}$-th prefix average of an array $\boldsymbol{S}$ is the average of the first $(\boldsymbol{i}+1)$ elements of $\boldsymbol{S}$:

$$\boldsymbol{A}[\boldsymbol{i}] = (\boldsymbol{S}[0] + \boldsymbol{S}[1] + \ldots + \boldsymbol{S}[\boldsymbol{i}])/(\boldsymbol{i}+1)$$

- Problem: Compute the array $\boldsymbol{A}$ of prefix averages of another array $\boldsymbol{S}$
- Has applications in financial analysis
- Solution 1: A quadratic-time algorithm: quadPrefixAve
- Example:

S	**21**	**23**	**25**	**31**	**20**	**18**	**16**
	0	1	2	3	4	5	6

A	**21**	**22**	**23**	**25**	**24**	**23**	**22**
	0	1	2	3	4	5	6

Algorithm quadPrefixAve(*S*, *n*)

Input: array *S* of *n* integers

Output: array *A* of prefix averages of *S*

	#operations
A ← new array of *n* integers	n
for *i* ← 0 **to** *n* - 1 **do**	n
s ← *S*[0]	n – 1
for *j* ← 1 **to** *i* do	1 + 2 + …+ (n - 1)
s ← *s* + *S*[*j*]	1 + 2 + …+ (n - 1)
A[*i*] ← *s* / (*i* + 1)	n - 1
return *A*	1

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

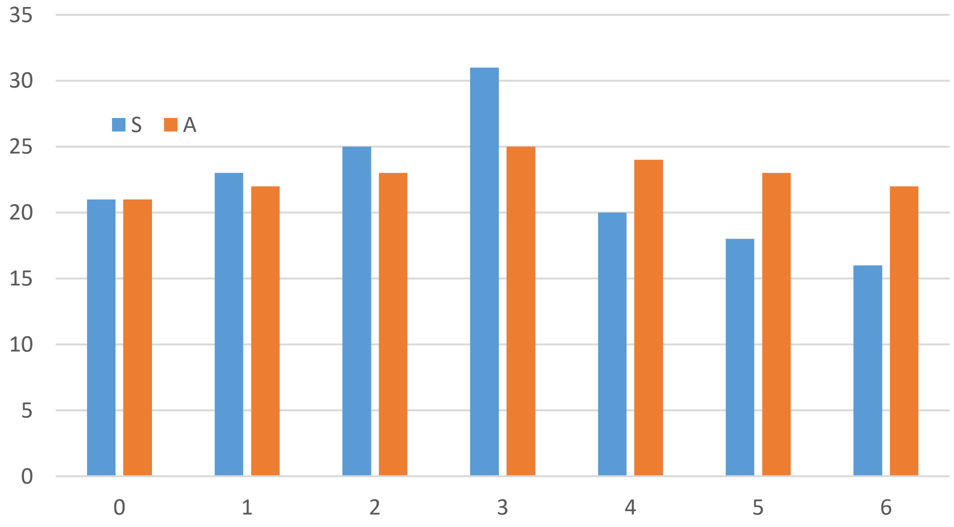
$T_2(n) = 2n + 2(n\text{-}1) + 2n(n\text{-}1)/2 + 1$ is $O(n^2)$

Case study 2: Prefix averages

- Solution 2: A linear-time algorithm: linearPrefixAve
- For each element being scanned, keep the running sum

S	**21**	**23**	**25**	**31**	**20**	**18**	**16**
	0	1	2	3	4	5	6

A	**21**	**22**	**23**	**25**	**24**	**23**	**22**
	0	1	2	3	4	5	6



Algorithm linearPrefixAve(*S*, *n*)

Input: array *S* of *n* integers

Output: array *A* of prefix averages of *S*

	#operations
$A \leftarrow$ new array of n integers	n
$s \leftarrow 0$	1
for $i \leftarrow 0$ **to** $n - 1$ **do**	n
$\quad s \leftarrow s + S[i]$	n – 1
$\quad A[i] \leftarrow s / (i + 1)$	n – 1
return A	1

$T_2(n) = 4n$ is $O(n)$

Case study 3: Maximum contiguous subsequence sum (MCSS)

- Problem:
  - Given: a sequence of integers (possibly negative) $A = A_1, A_2, \ldots, A_n$
  - Find: the maximum value of $\sum_{k=i}^{j} A_k$
  - If all integers are negative the MCSS is 0
- Example:
  - For A = -3, 10, -2, 11, -5, -2, 3 the MCSS is 19
  - For A = -7, -10, -1, -3 the MCSS is 0
  - For A = 12, -5, -6, -4, 3 the MCSS is 12
- Various algorithms solve the *same* problem
  - Cubic time
  - Quadratic time
  - Divide and conquer
  - Linear time

MCSS: Cubic vs quadratic time algorithms

Algorithm cubicMCSS(A,n)

Input: A sequence of integers A of length n

Output: The value of the MCSS

maxS ← 0

for i ← 0 **to** n – 1 **do**

  for j ← i **to** n – 1 **do**

    curS ← 0

    for k ← i **to** j **do**

      curS ← *curS* + $A[k]$

    if *curS* > *maxS*

      maxS ← *curS*

return *maxS*

$$\sum_{i=0}^{n-1} \sum_{j=i}^{n-1} \sum_{k=i}^{j} 1 = \frac{n^3 + 3n^2 + 2n}{6}$$

$T(n) = \frac{n^3+3n^2+2n}{6} + c$ is $O(n^3)$

Algorithm quadraticMCSS(A,n)

Input: A sequence of integers A of length n

Output: The value of the MCSS

maxS ← 0

for i ← 0 **to** n – 1 **do**

  for j ← i **to** n – 1 **do**

    curS ← *curS* + $A[j]$

    if *curS* > *maxS*

      maxS ← *curS*

return *maxS*

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

The double sum will give $O(n^2)$

Example: for A = -3, 10, -2, 11, -5, -2, 3 the MCSS is 19

MCSS: Divide and conquer

- Main features:
  - Rather lengthy
  - Split the sequence into two

First half Second half

A = -3, 10, -2, 11, -1, 2, -3

MCSS spans both halves

- Algorithm:
  - Divide subsequence into two halves
  - Find max left border sum (left arrow)
  - Find max right border sum (right arrow)
  - Return the sum of both maximums as the max sum
  - Do this recursively for each half
  - Complexity given by $T(n) = 2T(n/2) + n$, where $T(1) = 1$
  - Runs in $O(n \log n)$

Linear time algorithm

- Tricky parts of this algorithm are:
  - No MCSS will **start** or **end** with a negative number
  - We only find the value of the MCSS
  - But if we need the actual subsequence, we'll need to resort on at least divide and conquer

Example: for A = -3, 10, -2, 11, -5, -2, 3 the MCSS is 19

Algorithm linearMCSS(*A*,*n*)

Input: A sequence of integers *A* of length *n*

Output: The value of the MCSS

maxS ← 0; *curS* ← 0

for *j* ← 0 **to** *n* – 1 **do**

    curS ← *curS* + *A*[*j*]

    if *curS* > *maxS*

        maxS ← *curS*

    else

        if *curS* < *0*

            curS ← 0

return *maxS*

The single for loop gives $O(n)$

Example: Best vs worst case

Loops:

Worst Case: take maximum

Best Case: take minimum

	worst	**best**
$i \leftarrow 0$	1	1
while $i < n$ **and** $A[i]$!= 7	n	1
$\quad i \leftarrow i + 1$	n	0
	O(n)	**O(1)**

Worst-case input:

3	1	4	2	3	2	1	8	
0	1	2	3	4	5	6	7	**n**

Best-case input:

7	1	5	4	8	2	1	9	
0	1	2	3	4	5	6	7	**n**

Graphs

- **Definition:** A *graph* G is a pair (V, E), where
  - $V = \{v_1, v_2, \ldots, v_m\}$ is a set of vertices
  - $E \subseteq V \times V$ is a binary relation on V
- **Directed graph:**
  - E contains ordered pairs,

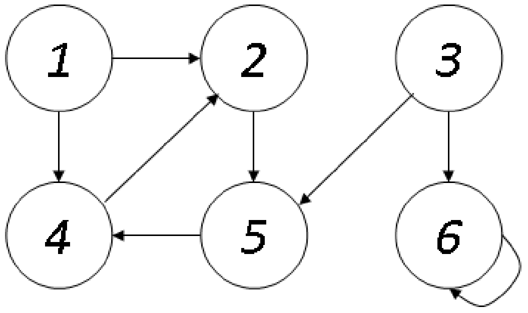
    i.e., pair $(u,v) \neq (v,u)$
- **Undirected graph:**
  - E contains unordered pairs,

    i.e., each pair (u,v) is a set {u,v}

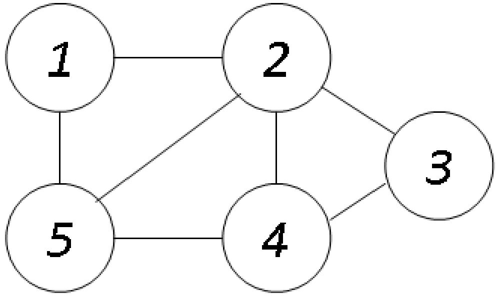
Definitions:

- **Path:** a sequence of vertices
- **Simple path:** all vertices are different
- **Cycle:** first and last vertices in path are equal
- **Acyclic graph:** It has no cycles
- **DAG:** Directed acyclic graph

Directed:



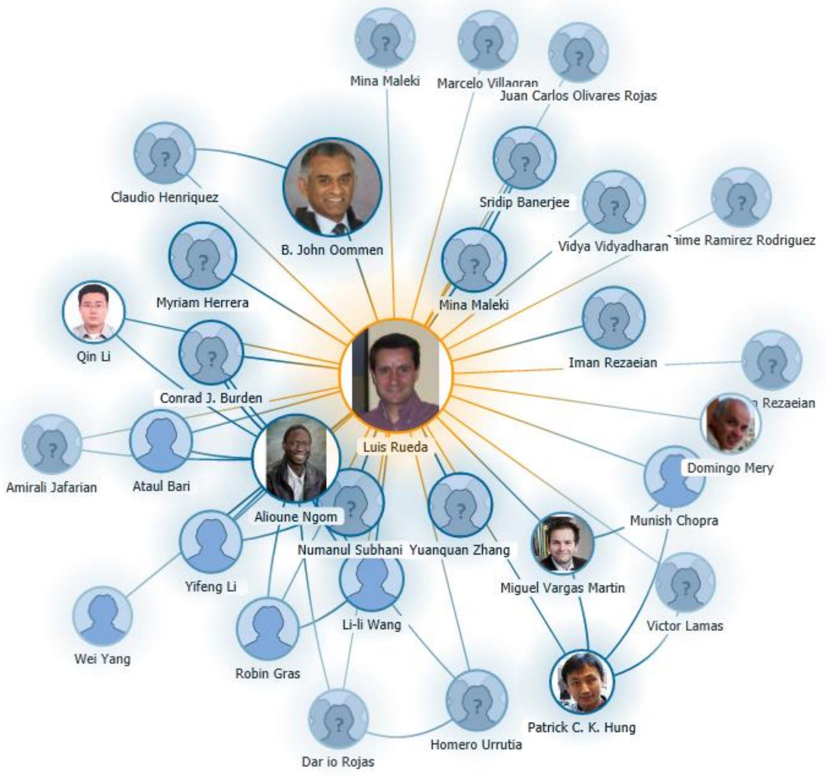
Undirected:



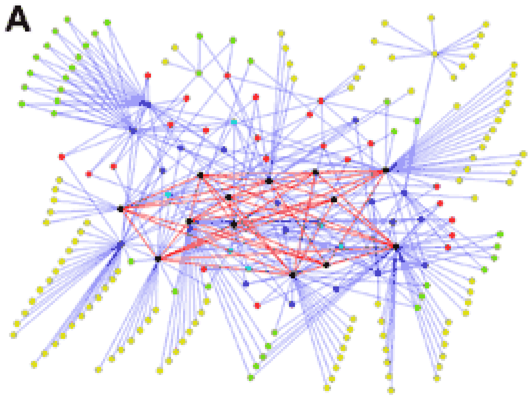
Graphs – sparseness + application examples

- Sparse graph:
  - G = (V, E) is *sparse*, if |E| <<< $|V|^2$
    Space used is O(|V|+|E|)
- Dense graph:
  - G = (V, E) is dense, if |E| ≅ $|V|^2$
    Space used is O(|V|+|E|)

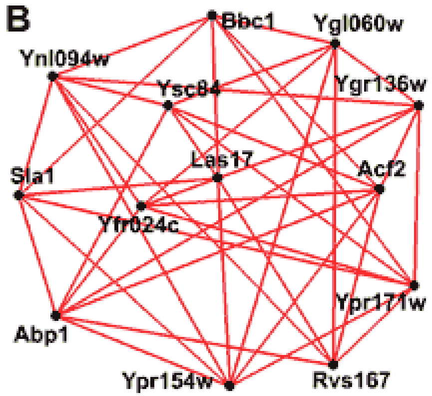
Sparse: Publication co-authorship network



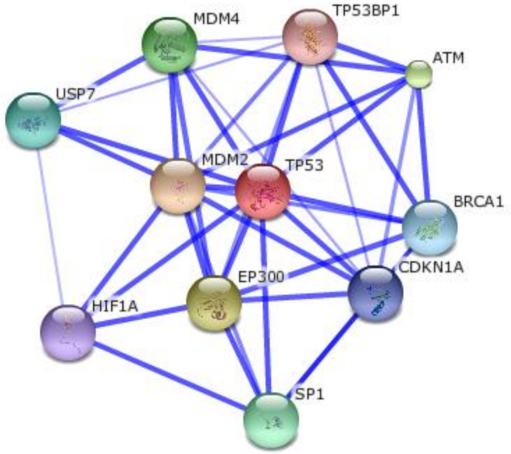
Sparse:



Dense:

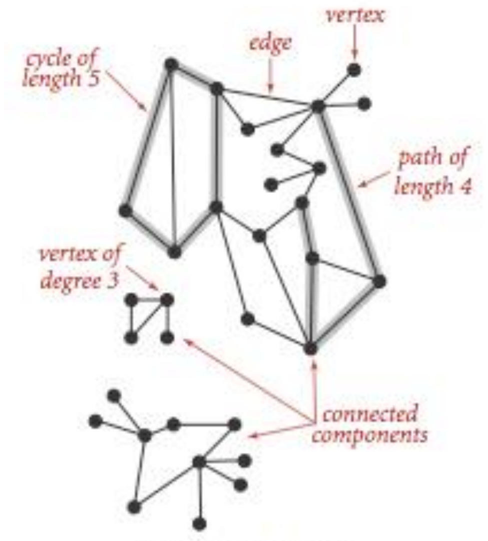


Dense: Protein interaction network

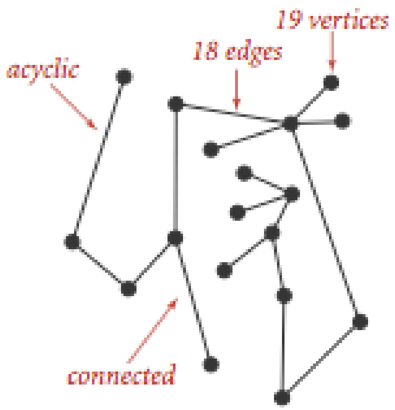


Graphs – more definitions, notation

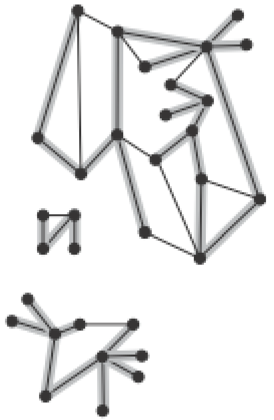
- Vertex w is adjacent to v iff (v,w) $\in$ E
  - Notation: Adj[v]
- Tree: Undirected acyclic graph
- Forest: a disjoint set of trees
- Reachable: Vertex u is reachable from v if there is a path from v to u
- A vertex v is connected to another vertex u if there is a path from v to u
- A graph is connected if there is a path from every vertex to every other vertex
- Spanning tree: A subgraph that is a tree and contains all vertices
- Spanning forest: Spanning trees of an unconnected graph



Anatomy of a graph



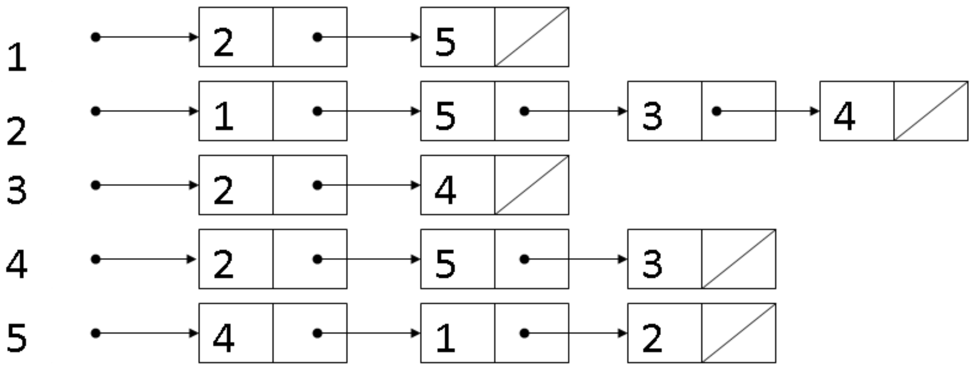
A tree

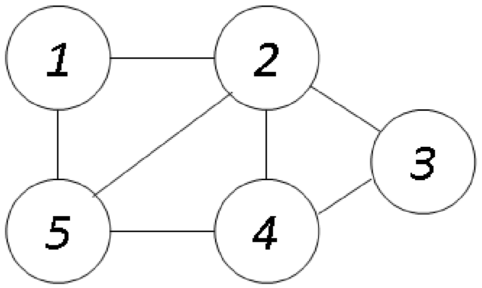


A spanning forest

Graphs - representations

- **Adjacency list:**
  - An array of |V| singly linked lists: Adj[u]
  - $\forall$ u $\in$ V, Adj[u] contains all v, s.t. u $\neq$ v, and (u,v) $\in$ E



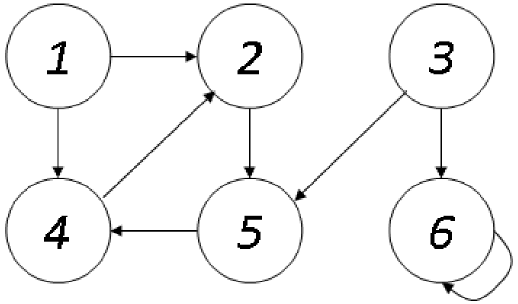


Space complexity: O(|E|+|V|)

- **Adjacency matrix:**
  - It is a |V| $\times$ |V| matrix A = $\{a_{ij}\}$, where

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

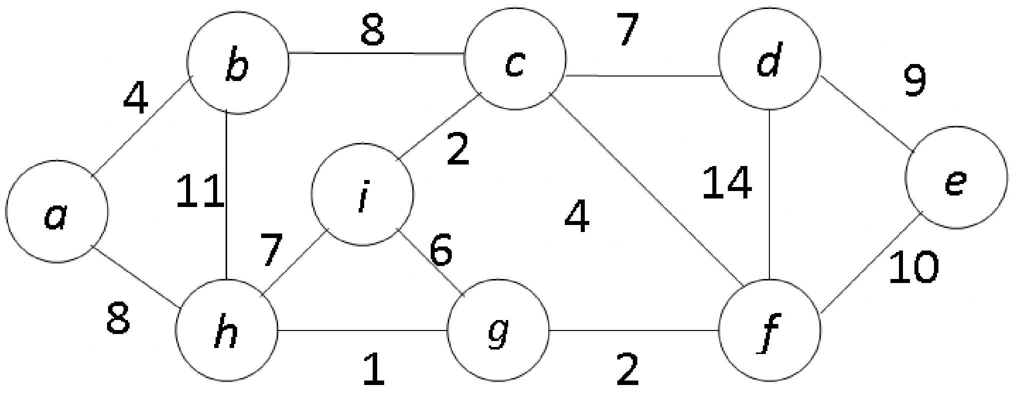
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

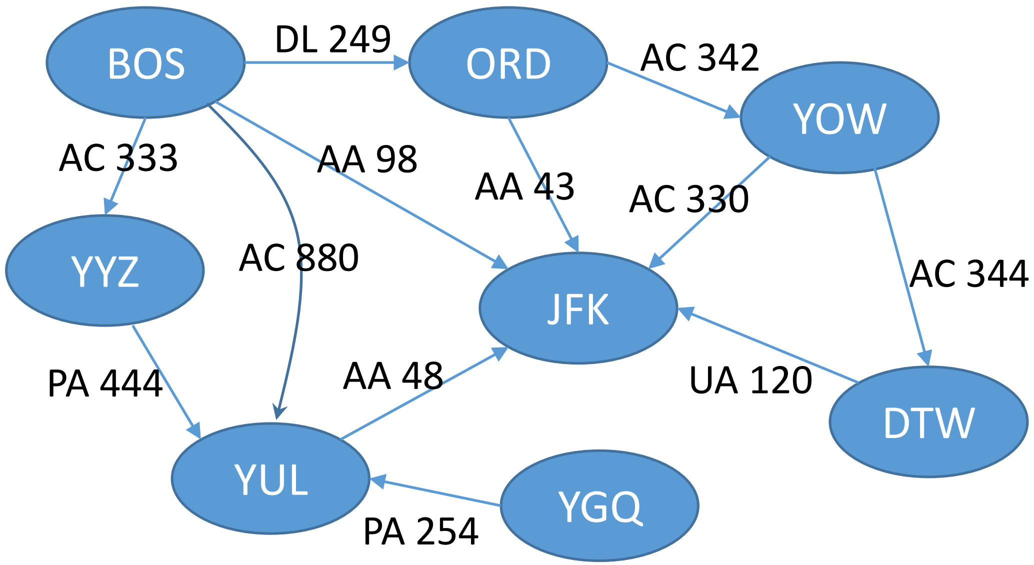


Space complexity: O(|V|^2)

Weighted graphs

- **Definition** (weighted graph):
  - A connected, undirected graph G = (V, E) is *weighted*, if
  - ∀ (u,v) ∈ E, ∃ a *weight*, given by function w(u,v)
- More general:
  - w(u,v) can be an arbitrary object
  - Example: flights from one airport to another.
  - Object may contain more info about flight (e.g., date, departure time, arrival time, etc.)





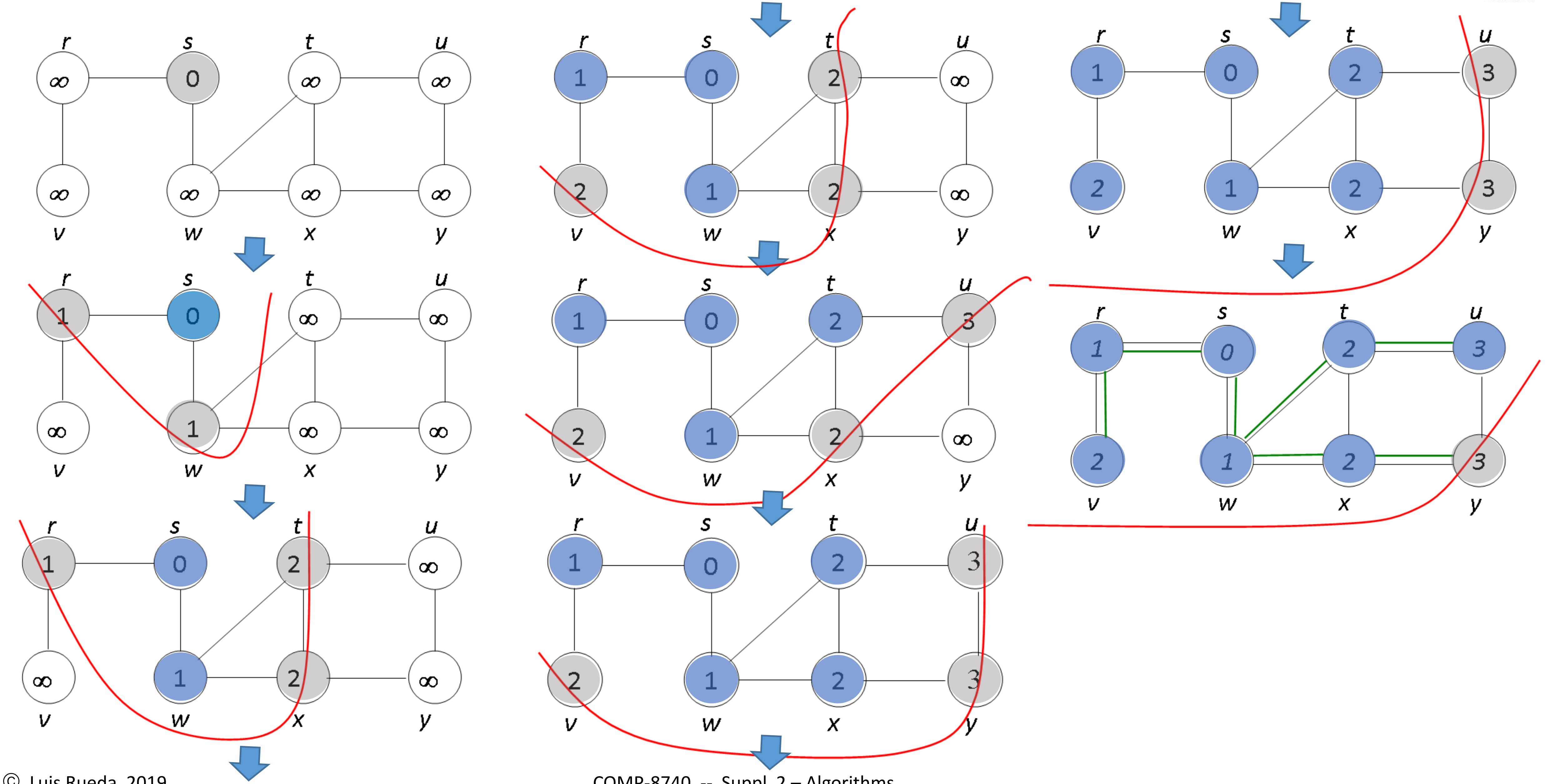
Graphs – traversal

Breath-first search

- Given G = (V, E), and a *source* vertex, s,
- Breadth-first explores *every* vertex, v, reachable from s:
  - Computes distance from s to v
  - Produces a breadth-first tree whose root is s
- Path from s to v is the shortest one
- Works on both *directed* and *undirected* graphs
- Worst-case time of BFS is $O(|V| + |E|)$

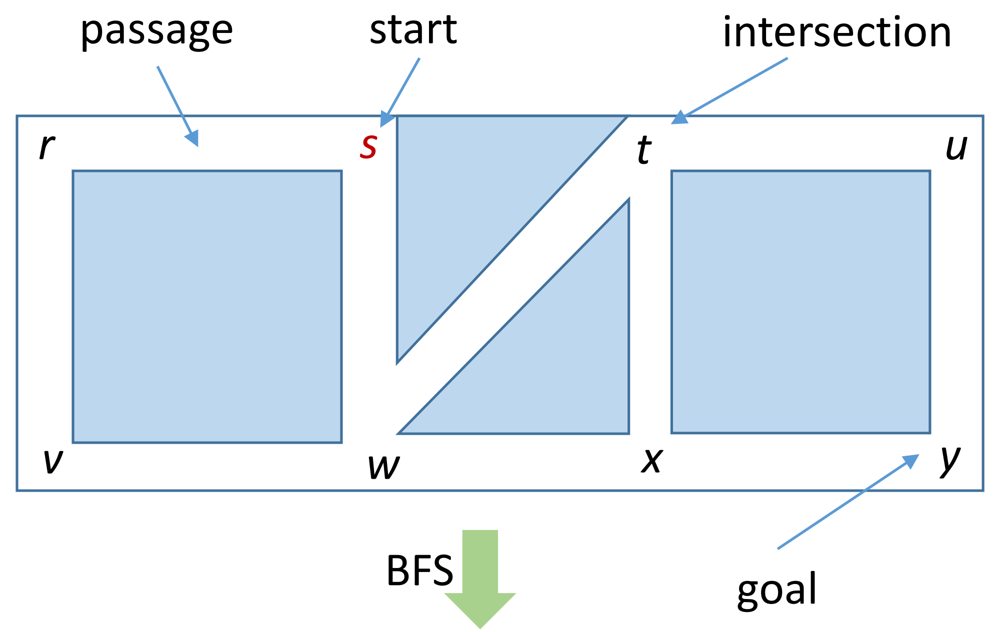
Algorithm BFS(G, s)
for each $u \in V - \{s\}$
    color[u] $\leftarrow$ "white"
    $d[u] \leftarrow \infty$; $\pi[u] \leftarrow$ nil
color[s] $\leftarrow$ "gray"
$d[s] \leftarrow 0$; $\pi[s] \leftarrow$ nil   // p[u] = predecessor of u
$Q \leftarrow \varnothing$ // Create an empty queue
enqueue s to Q
while $Q \neq \varnothing$
    $u \leftarrow$ dequeue from Q
    for each $v \in Adj[u]$
        if color[v] = "white"
            color[v] $\leftarrow$ "gray"; $d[v] \leftarrow d[u] + 1$; $\pi[v] \leftarrow u$
            enqueue v to Q
    color[u] $\leftarrow$ "blue"

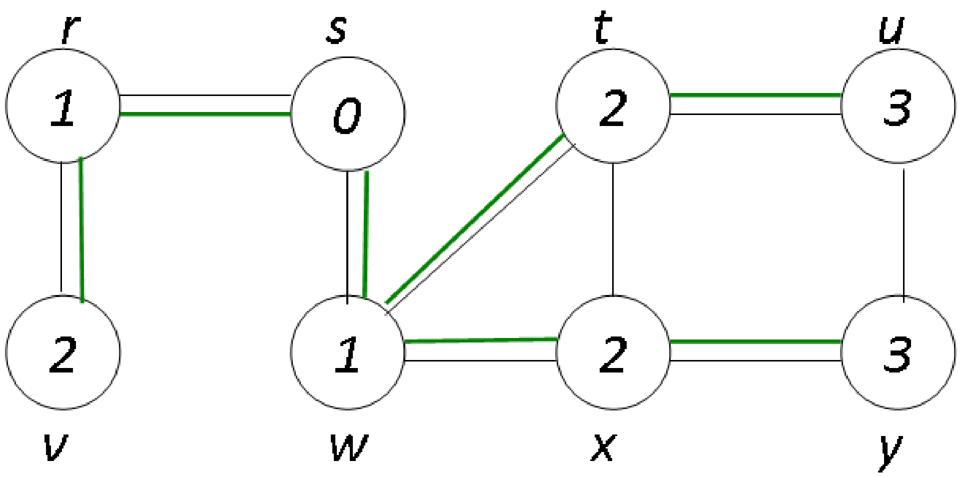
Breath-first search - example



Application of BFS: Search in a maze

- Maze
  - Given a maze represented by a graph
  - Each passage is an edge in the graph
  - Each vertex is an intersection in the graph
- Problem
  - Given a starting point in the maze, *s*
  - Explore the maze using BFS
- BFS can be used to:
  - Explore the maze
  - Find a lost object or a goal in the maze
- BFS will avoid visiting any passage or intersection twice





Graphs – traversal

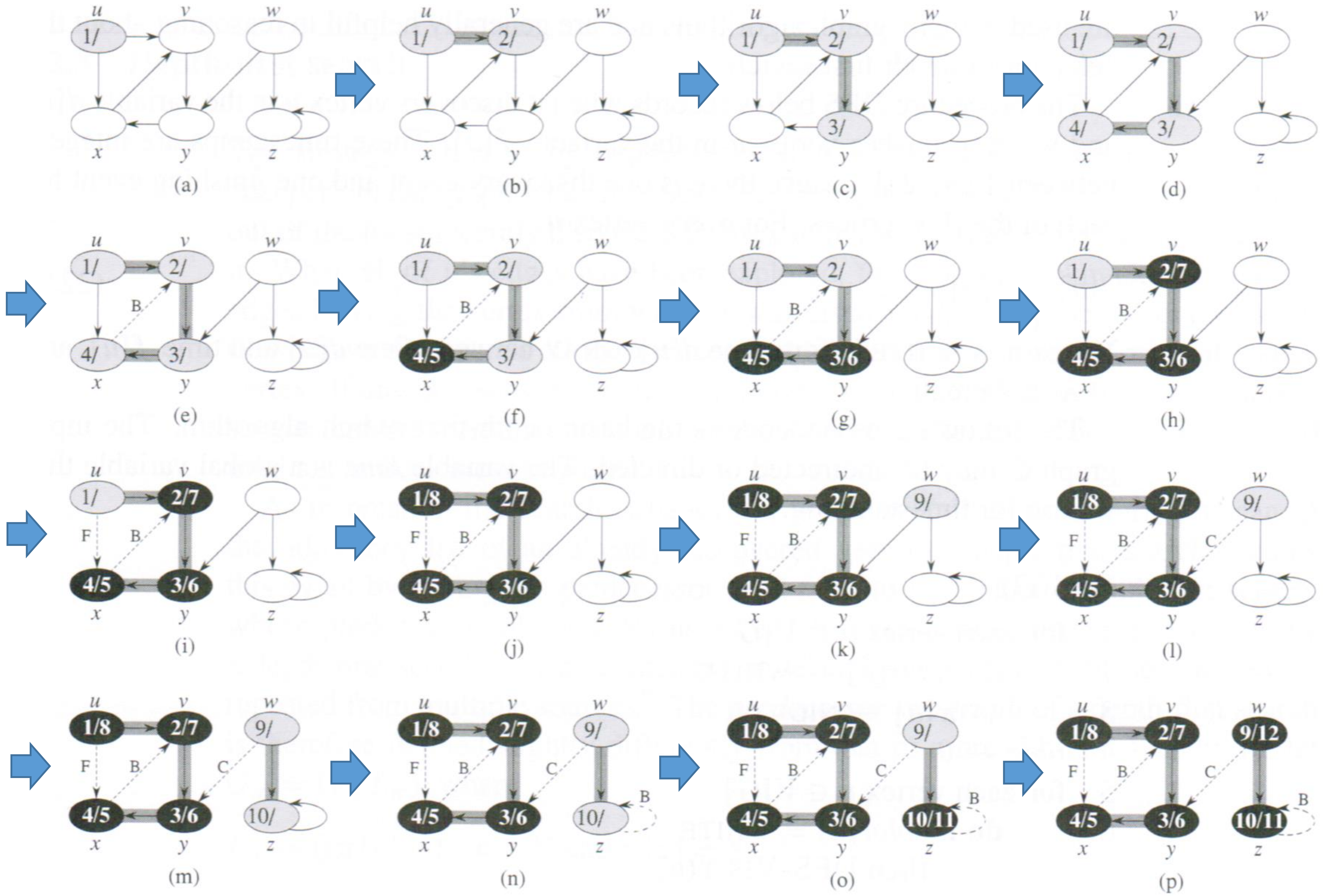
Depth-first search (DFS)

- Given G = (V, E), and a *source* vertex, s,
- Depth-first explores *every* vertex, v, reachable from s
- Simplest way to implement DFS is through recursion
- Unlike BFS, DFS goes "deep" first and then continues the search
- The recursion stack allows to go deep first
- Nonrecursive DFS uses a stack instead of a queue
- Works on both *directed* and *undirected* graphs
- Running time of DFS is O(|V|+|E|)
- DFS provides both preorder and postorder traversals of the graph

```
Algorithm DFS(G)
for each u ∈ V
   color[u] ← "white"
   p[u] ← nil
time ← 0
for each u ∈ V
   if color[u] = "white"
      DFS-Visit(u)
```

```
DFS-Visit(u)
color[u] ← "gray"
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]
   if color[v] = "white"
      p[v] ← u
      DFS-Visit(v)
color[u] ← "black"
f[u] ← time
time ← time + 1
```

Depth-first search – example: directed graph



Example not discussed in full detail –
found on page 542 of [5]

Shortest paths

- **Definition:**
  - A path of length $\delta(s, v)$ is a *shortest* path from s to v if it has the *minimum* number of edges.
  - No path from s to v $\Rightarrow \delta(s, v) = \infty$
- **Single-source shortest-path problem:**
  - Given:
    - A weighted, directed graph G = (V, E)
    - A weight function $w : E \rightarrow \mathbf{R}^+$
    - A source vertex $s \in V$
    - Weights are nonnegative
  - Aim: Find shortest path from s to *every* $v \in V, v \neq s$
- Dijkstra's algorithm is the most popular
- Other algorithms: Ch. 14 of [1]

Related problems:

- Single-destination shortest-path:
  - Given t (a destination), find a shortest path form every $v \in V$.
- Single-pair shortest-path:
  - Given $u, v \in V$, find shortest path from u to v.
- All-pairs shortest-path:
  - Given a weighted, directed graph G = (V, E), $\forall\, u, v \in V$, find a shortest-path from u to v.

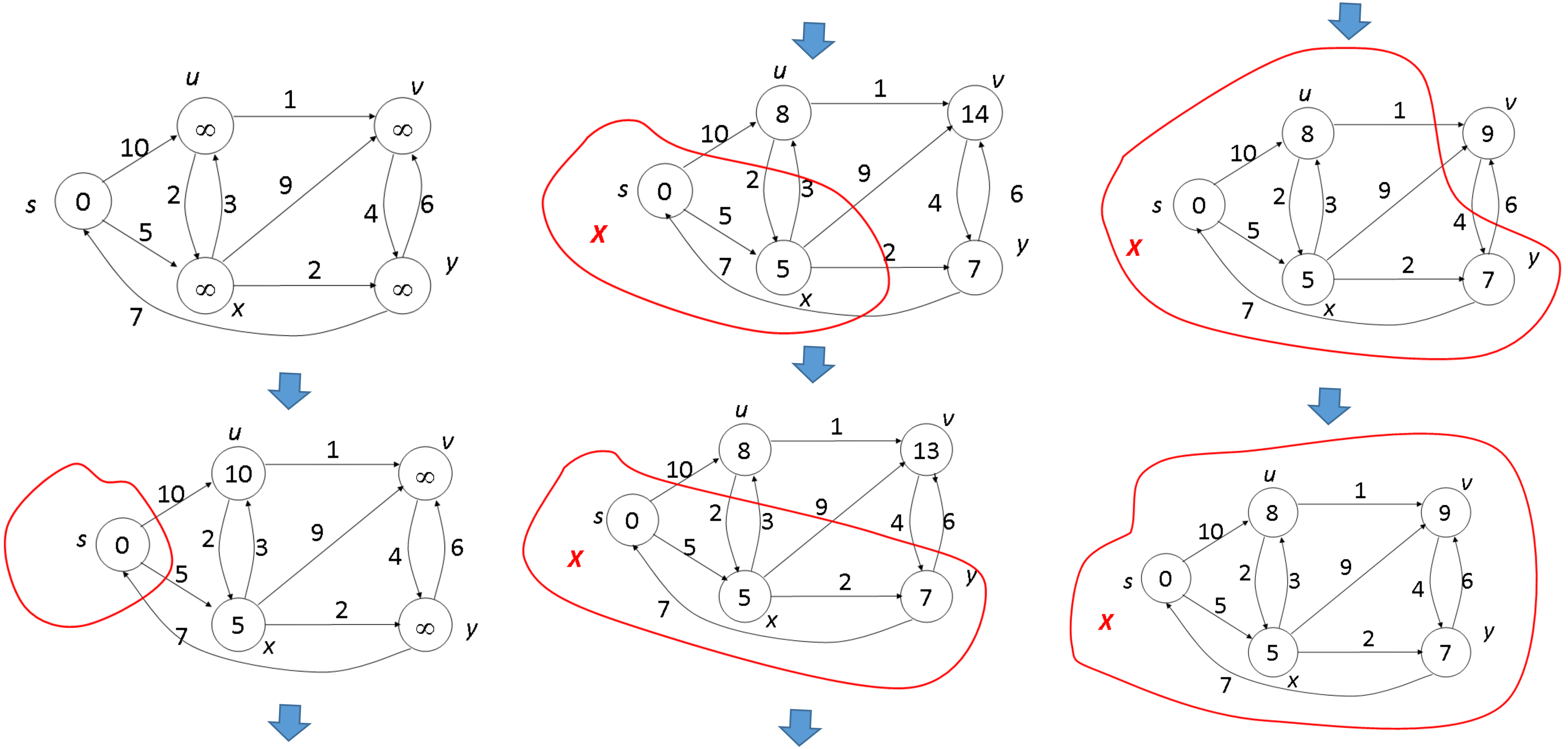
```
Algorithm SP-Dijkstra(G, w, s)
Initialize-Single-Source(G,s)
S ← ∅
Q ← V    // the queue is a heap
while Q ≠ ∅
   u ← Extract-Min(Q)
   S ← S ∪ {u}
   for each v ∈ Adj[u]
      Relax(u,v,w)

Initialize-Single-Source(G, s)
for each v ∈ V
   d[v] ← ∞
   p[v] ← nil
d[s] ← 0

Relax(u, v, w)
if d[v] > d[u] + w(u,v)
   d[v] ← d[u] + w(u,v)
   p[v] ← u
```

- Dijkstra's algorithm is greedy
- Running time: $O(|E|+|V| \log |V|)$ by using a Fibonacci heap

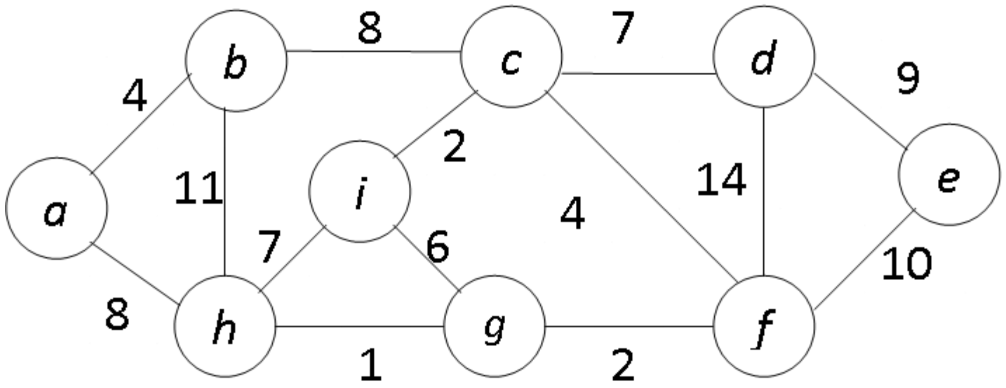
Dijkstra’s algorithm – example



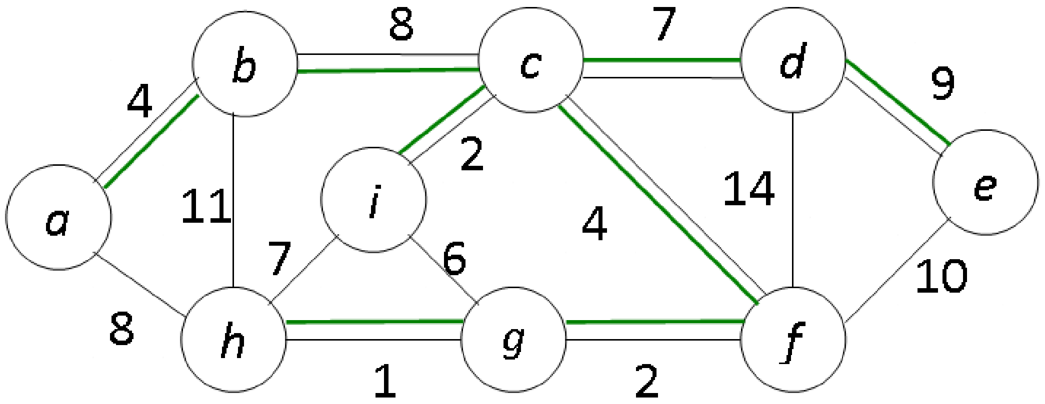
Minimum spanning trees - MST

- **Definition** (MST):
  - Given a connected, undirected, weighted graph G = (V, E),
  - a MST is an *acyclic* graph T = (V_T, E_T), where:
  - V_T = V and $E_T \subseteq$ E, and
  - $w(T) = \sum_{(u,v)\in E_T} w(u,v)$ is *minimum*
- Called MST since it "spans" graph G
- Algorithms for finding the MST:
  - Kruskal's algorithm
  - Prim's algorithm
  - both algorithms are *greedy*
- Assumptions:
  - *G* is connected, undirected, and weighted

Weighted graph:



MST:



In clustering, the weights are the distances between points: e.g., Euclidean, Manhattan, etc.

Kruskal's algorithm

- It is a *greedy* algorithm
- The set A is a forest
- Use a specific rule to find a safe edge
- Safe edge (u,v):
  - an edge whose weight is the smallest, and
  - that connects two trees, C_1 and C_2, in the forest,
  - yielding a new tree
- (u,v) is a light edge connecting C_1 to another tree,

  $\Rightarrow$ (u,v) is a safe edge for C_1
- Worst case running time: O(|E| log |E|)

Algorithm MST-Kruskal(G, w)

$A \leftarrow \varnothing$

for each $v \in V$

  Make-Set(v)

sort E in increasing order of weight w

for each $(u,v) \in E$

  if Find-Set(u) $\neq$ Find-Set(v)

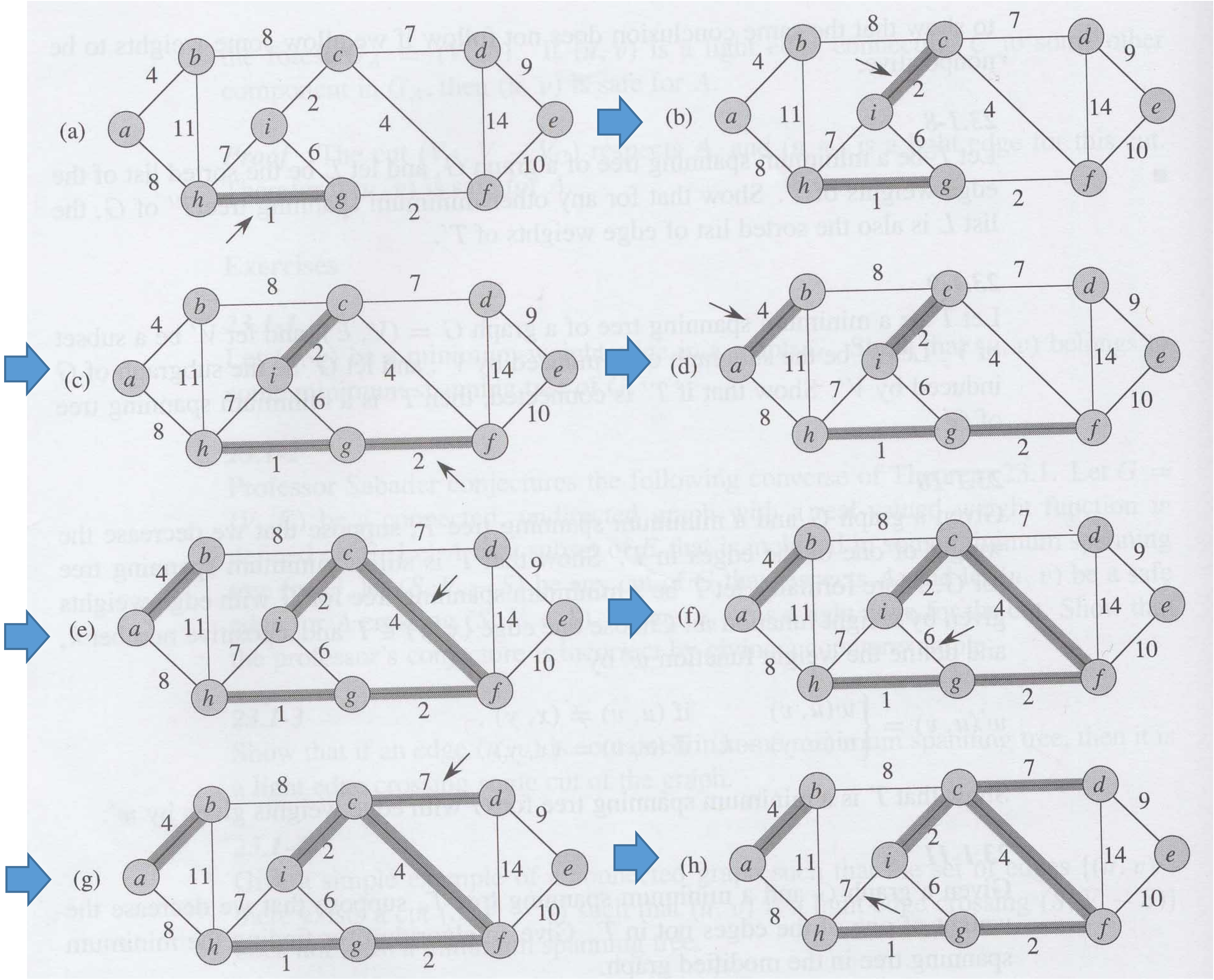
    $A \leftarrow A \cup \{(u,v)\}$

    Union(u,v)

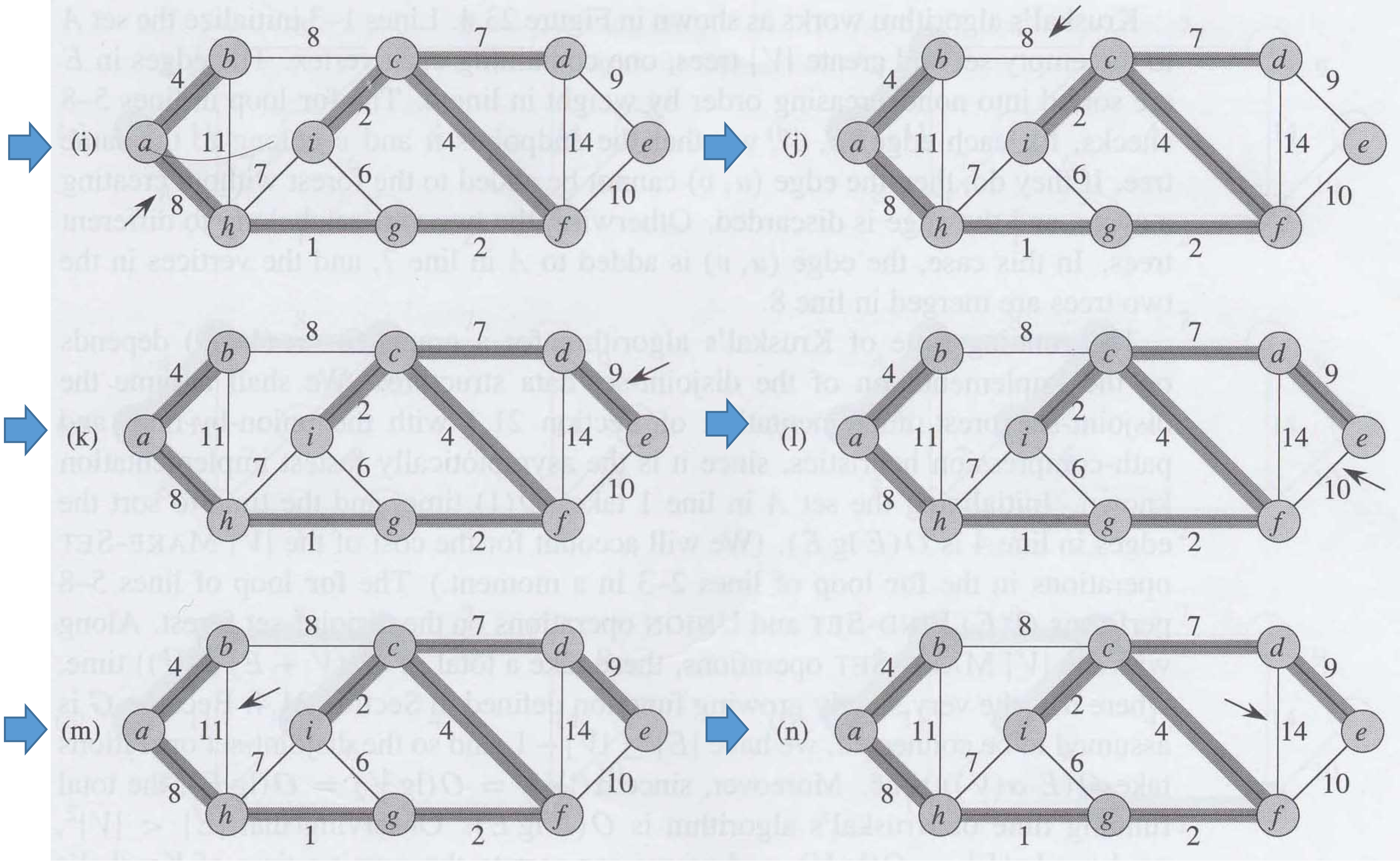
return A

Note: Implementation of Make-Set(v), Find-Set(u), and Union(u,v) can be found in [4]

Kruskal's algorithm - example



Kruskal’s algorithm - example



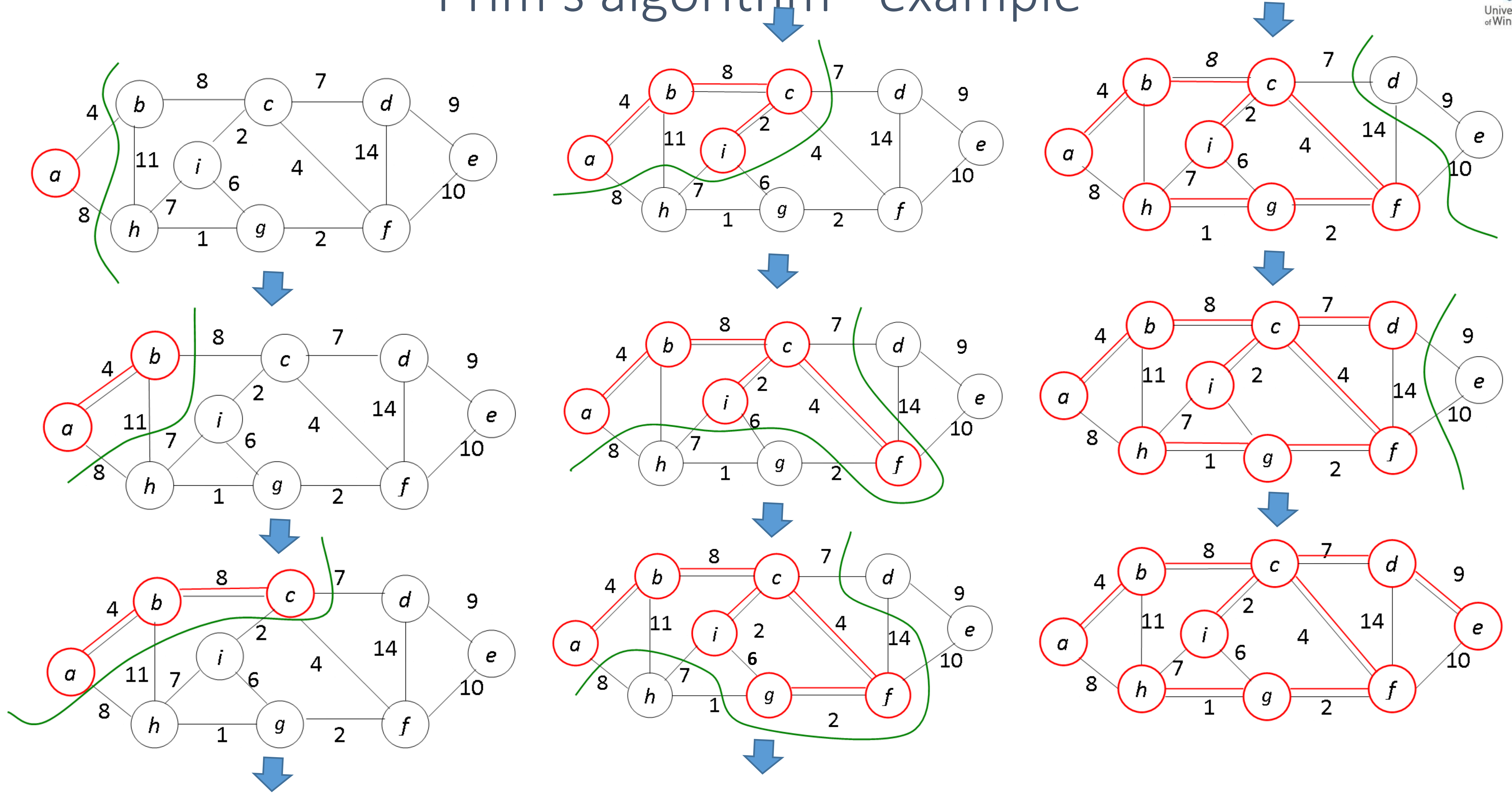
Prim's algorithm

- Unlike Kruskal's algorithm, it maintains a *single* tree, A.
- Starts from an arbitrary vertex, say r,

  and spans all vertices in V
- At each step,
  - adds a *safe* vertex to A
  - $\Rightarrow$ at the end, A forms a *minimum spanning tree*.
- Worst-case running time:
  - Using a binary heap: O(|E| log |V|)

```
MST-Prim(G, w, r)
for each u ∈ V
   key[u] ← ∞
   p[u] ← nil
key[r] ← 0
Q ← V                  // Q is a heap
while Q ≠ ∅
   u ← Extract-Min(Q)
   for each v ∈ Adj[u]
      if v ∈ Q and w(u,v) < key[v]
         p[v] ← u
         key[v] ← w(u,v)
```

Prim's algorithm - example



Connected components – undirected graphs

- Connected components can be found in an undirected graph

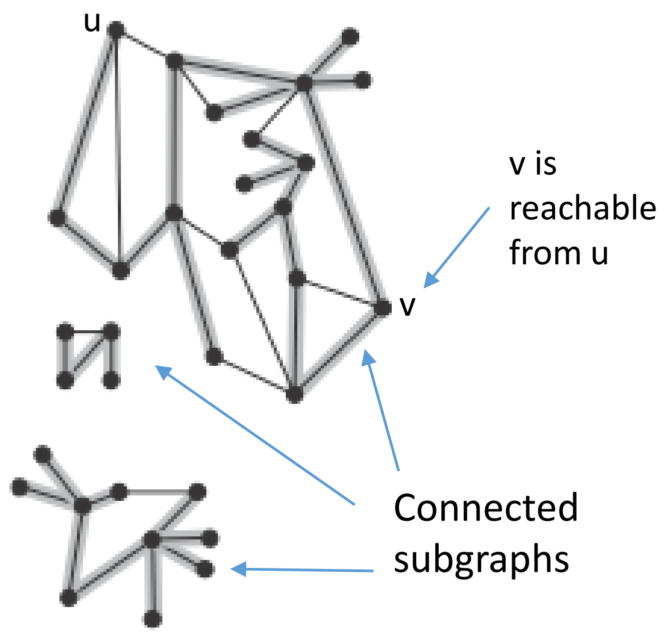
Important definitions:

- <u>Reachable:</u> Vertex u is reachable from v if there is a path from v to u
- <u>Connected:</u> A vertex v is connected to another vertex u if there is a path from v to u
- A graph is connected if there is a path from every vertex to every other vertex

High-level pseudocode:

- Given graph G
- Run DFS on G
- If DFS fails to find all vertices in G:
  - Restart DFS on unvisited vertices
- Return spanning forest

Spanning forest:



- Connected components can be found in O(|V|+|E|)

NP-Completeness

- Consider a *polynomial-time* algorithm, A that solves problem Π
- If the size of the input is *n*,

  $\Rightarrow$ worst-case running time is $O(n^k)$,

  where *k* is a *constant*, or $k = O(1)$
- In this case,
  - $\Pi \in$ P,
  - where P is a "class" of problems defined later
- Not all problems can be solved in polynomial time
- And not all problems can be solved by a computer even in infinite time!
- **Example 1:**
  - The *Halting Problem* **cannot** be solved by **any** computer
- **Example 2:**
  - Traveling salesman problem (TSP) -- more later
  - So far, it cannot be solved in polynomial time
  - However, we do not know if $\exists$ an algorithm that can solve TSP in polynomial time!
- A problem Π, like TSP, belongs to a certain class of problems:

  NPC (NP-complete),

  and we say that Π is NP-complete,

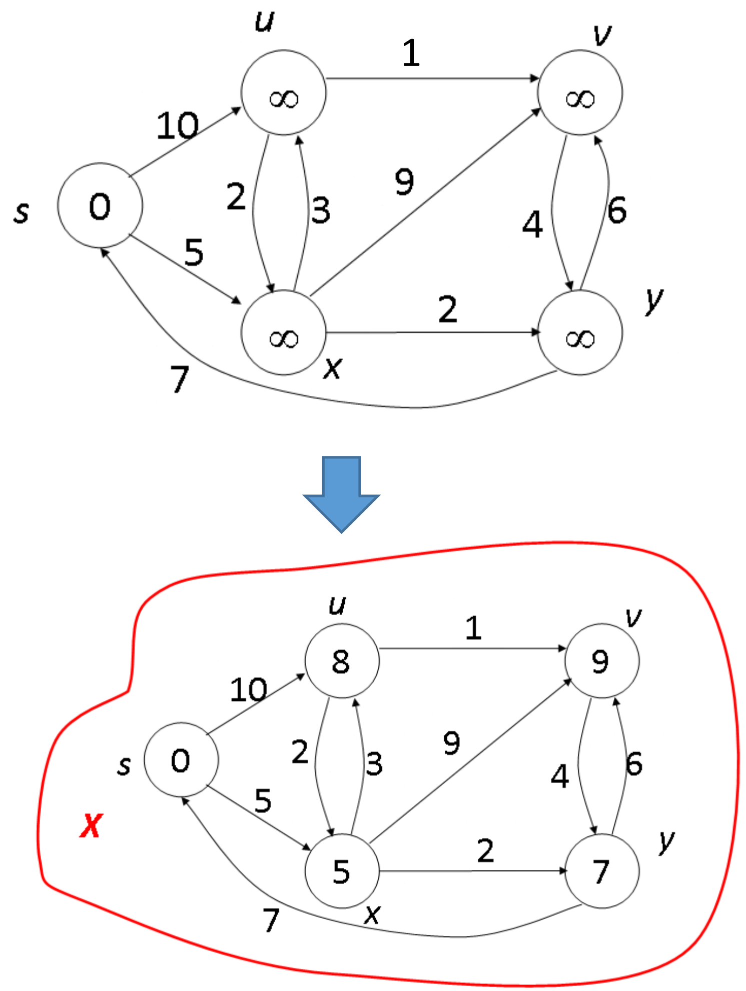
  where NP is a "third" class of problems
- **Key issues:**
  - Some problems can be solved by an algorithm in polynomial time
  - No algorithm that solves an NP-complete problem has been proposed, so far
  - If so, *every* NP-complete problem could be solved in polynomial time
  - There are problems that cannot be solved by a computer

Graph problems

- **Shortest path:** PATH
- Many variants… consider the…
- Single-source shortest-path problem:
  - Given:
    - ➢ A weighted (un)directed graph G = (V, E)
    - ➢ A weight function $w : E \rightarrow \mathbf{R}^+$
    - ➢ A source vertex $s \in V$
  - Aim: Find shortest path from vertex s to *every* $v \in V, v \neq s$
- Example: Directed weighted graph
  - Apply Dijkstra's algorithm
  - Find shortest path between s and any other vertex can be found
  - Shortest path from s to v: 5 + 3 + 1 = 9
  - Runs in polynomial time:

    $O(|E|+|V| \log |V|)$
  - PATH is in P

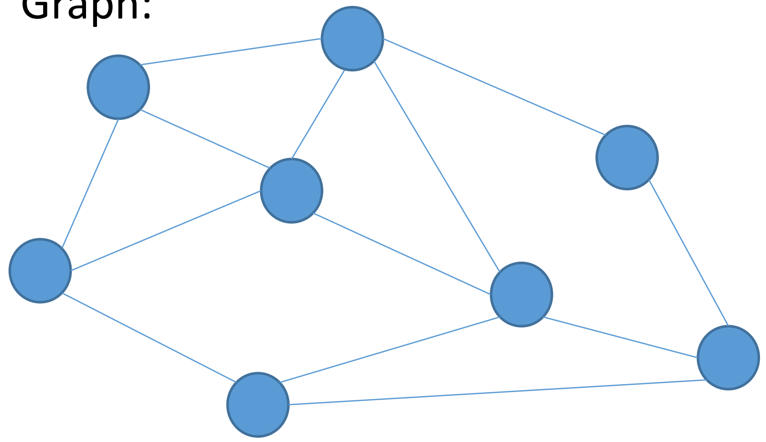
Example:



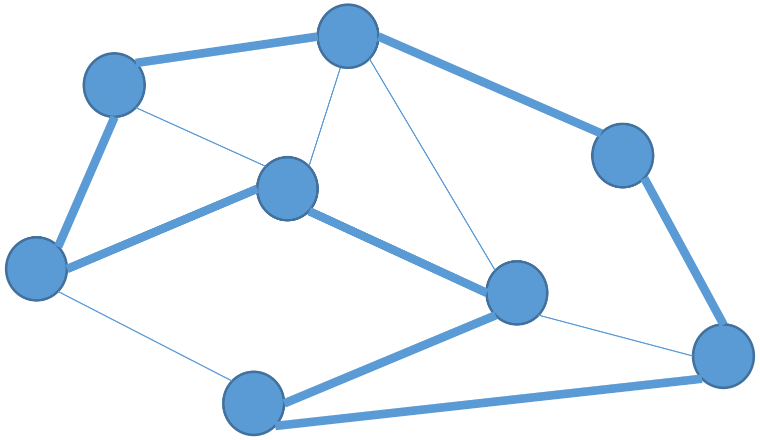
Graph problems

- **Hamiltonian cycle:** HAM-CYCLE
- Consider an undirected graph G = (V, E)
- **Cycle:** A path $\langle v_0, v_1, \ldots, v_k \rangle$, if $v_0 = v_k$, and the path contains *at least* one edge
- **Simple cycle:** A cycle is simple if $v_1, v_2, \ldots, v_k$ are *all* distinct
- **Hamiltonian cycle:** A simple cycle that contains each $v \in V$
- **Problem:** Given a graph G=(V,E) does G contain a Hamiltonian cycle?
- HAM-CYCLE cannot be solved in polynomial time!

Graph:



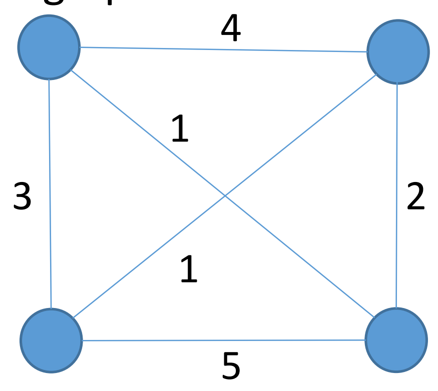
Hamiltonian cycle:



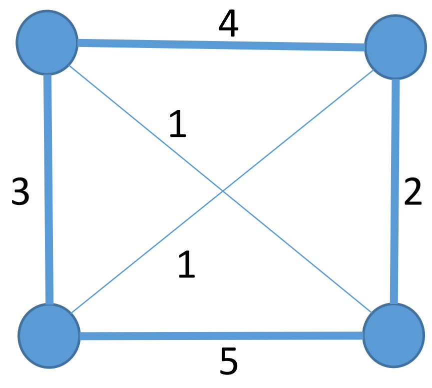
Graph problems

- **Traveling Salesman Problem:** TSP
- Given:
  - an undirected weighted graph G = (V, E)
- Find: a Hamiltonian cycle whose sum of weights is minimum
- TSP cannot be solved in polynomial time!
- Many Hamiltonian cycles may exist
- Hamiltonian cycle of (a) doesn't have the minimum weight
- Hamiltonian cycle of (b) is optimal: its weight is minimum
- There may be more than one Hamiltonian cycle whose weights are the minimum

Weighted graph:

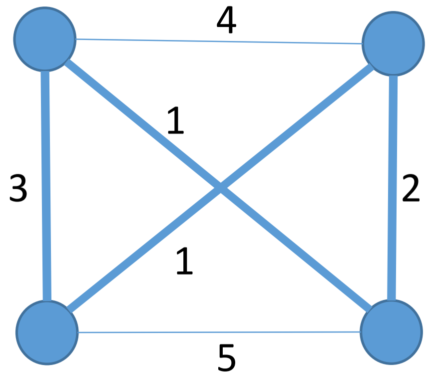


(a)



Sub-optimal
weight = 14

(b)



Optimal
Minimum weight = 7

Decision problems and Optimization problems

- Many problems we want to solve are *optimization problems*
- For example, PATH: we want to *minimize* the length of the path
- Indeed, we are looking for a path with the *smallest* weight
- However, NP-completeness focuses on *decision problems*

  $\Rightarrow$ set of solutions, S = {0,1},

  where 1 means "yes", and 0 means "no"

- Then, how do we deal with an optimization problem?
- If it is a *minimization* problem, we can "associate" that problem with a *decision* problem
- Include:
  - a numerical bound *k*, and
  - a parameter, which says that a structure has cost *less than k*
- Similarly, this can be done for *maximization* problems
- Key point:
  - There is a *relationship* between the optimization problem, and the decision problem

Decision problems and Optimization problems

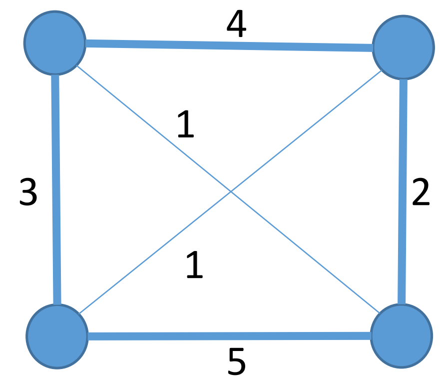
- The optimization problem is solved in polynomial time

if and only if

the decision problem is solved in polynomial time

- **Examples:**
- TSP:
  - Given a weighted, undirected graph G,
  - and an integer *k*,
  - Does there exist a Hamiltonian cycle with cost/weight at most *k*?
- PATH:
  - Given a
  - weighted, undirected graph G,
  - two vertices u and v,
  - and an integer $k \geq 0$,
  - Does there exist a path from u to v with weight at most *k*?

An instance of TSP with a specific cycle:



Cycle weight = 14

- <u>Decision problem</u>: Is there a Hamiltonian cycle with weight less than 14?
- <u>Optimization problem</u>: Find the Hamiltonian cycle with minimum weight

Problems and Languages

- In intractability, problems and languages are exactly the same!
- A decision problem Π can be expressed as a language L in {0,1}
- An instance I of problem Π is a binary string in L, e.g., x = 00110
- Then strings in {0,1}* can be divided into 3 classes:
  - (a) strings which are not encodings of instances of Π
  - (b) strings that encode instances of Π, and
    whose answer is "no" $\equiv \Pi(x) = 0$, and
  - (c) strings that encode instances of Π, and
    whose answer is "yes" $\equiv \Pi(x) = 1$
- Algorithm A decides language L if for every string x in {0,1}*, A can place x in one of these categories

Example 1:

- Whether or not a number is divisible by 4 can be represented in two ways:
- Problem:
  - DIV-FOUR: Is a number divisible by 4?
- Language:
  - L = {100, 1000, 1100, 10000, 10100, …} $\subset \{0,1\}^*$
- An algorithm A *accepts* a string x $\in \{0,1\}^*$ if A(x) = 1,
- If A runs in polynomial time for any string, then we say that Π is in P

Example 2: TSP
Problem: Given a weighted graph, does there exist a Hamiltonian cycle with cost/weight at most *k*?
Language: Given a weighted graph, encode it in binary; i.e., encode vertices, edges, weights.
A hypothetical language: L = {00011010100101, 100101010001, …}
where each binary string is the binary encoding of a Hamiltonian cycle whose weight is < *k*
Given a binary string x = 01110100101010, which represents Hamiltonian cycle, is x in L?

Class P

Class P: It's a set of languages

- P = {L $\subseteq$ $\{0,1\}^*$: $\exists$ an algorithm A that decides L in polynomial time}
- Examples:
  - DIV-FOUR, PATH, SORT

Class NP: It's also a set of languages

- A class of languages that can be *verified* by a polynomial-time algorithm, and *decided* by a nondeterministic algorithm (Turing machine)
- Problems in NP can be decided by

  a *nondeterministic Turing machine* (NDTM)

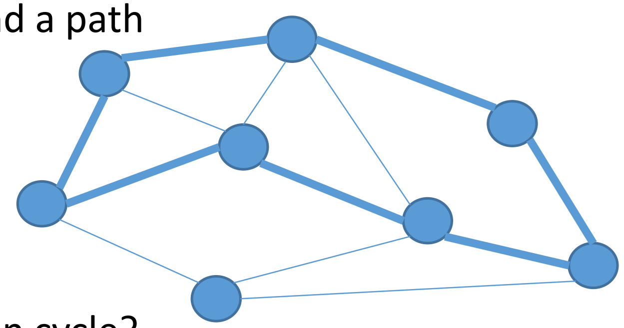
  while problems in P can be decided by a DTM or a NDTM
- Meaning of NP:
  - **N** : stands for "nondeterministic", because of the **N**DTM
  - **P** : every problem in NP can be verified in **P**olynomial time
- Examples
  - HAM-CYCLE, TSP $\in$ NP

Class P:

- Problems in class P can be easily solved by polynomial-time algorithms
  - SORT can be solved by Mergesort, Heapsort, etc. in $O(n^2)$ – i.e., in polynomial time

Class NP: Verification algorithm

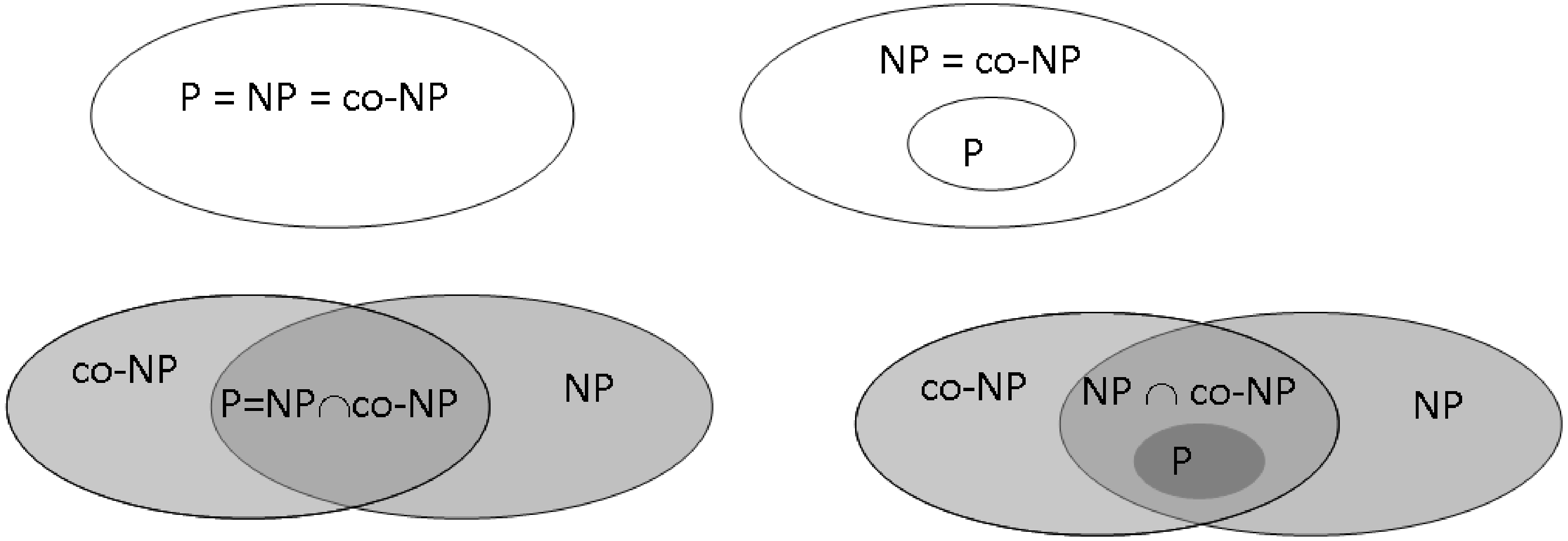
  - Given a graph and a path



  - is it a Hamiltonian cycle?
  - This can be "easily" verified in polynomial time
- How?
  - Check that every vertex is in the path
  - Check that every vertex is visited once
- Verification is simple… but finding the Hamiltonian cycle is difficult (NP-complete)!

Class co-NP

- The complexity class co-NP: all languages L such that the complement $\bar{L} \in NP$
- 4 possibilities:



- **Important axiom:** P $\subseteq$ NP.
- But we don't know if NP $\not\subset$ (or $\subseteq$?) P
- That is, if P = NP, then any problem in NP can be decided in polynomial time
- Or if P $\neq$ NP, then a problem in NP would have to be solved in exponential time!

Classes NP-hard and NP-complete

Two new classes of problems:

- NP-complete (NPC)
- NP-hard
- Using reduction, we "compare" two languages (problems)
- i.e., we use $\leq_P$ symbol

- Given L_1 and L_2,
- we can state how "harder" L_1 is w.r.t. L_2

- If L_1 can be reduced to L_2 ($L_1 \leq_P L_2$)
- $\Rightarrow$ L_1 is no more than a polynomial factor harder than L_2

- A language $L \subseteq \{0,1\}^*$ is **NP-complete** if:
  - $L \in NP$
  - $\forall L' \in NP, L' \leq_P L$
- A language $L \subseteq \{0,1\}^*$ is **NP-hard** if:
  - $\forall L' \in NP, L' \leq_P L$
- NP-completeness is crucial in proving whether or not $P \neq NP$

Theorem: Either

- (a) If *any* $L \in NPC$ is *polynomial-time* solvable

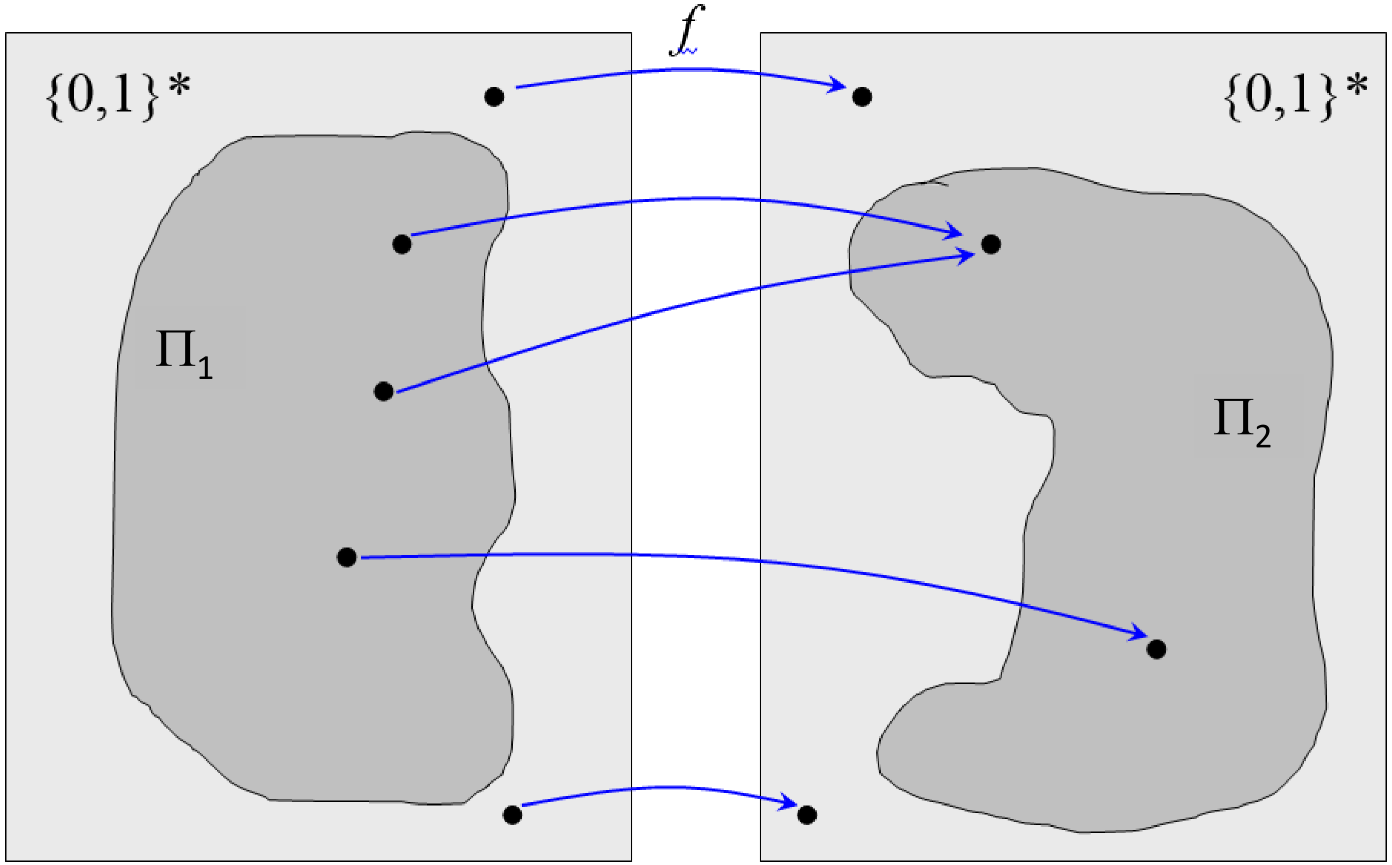
  $\Rightarrow$ P = NP, or
- (b) If *any* $L \in NP$ is **not** polynomial-time solvable

  $\Rightarrow$ *no* $L \in NPC$ is polynomial-time solvable

Reduction

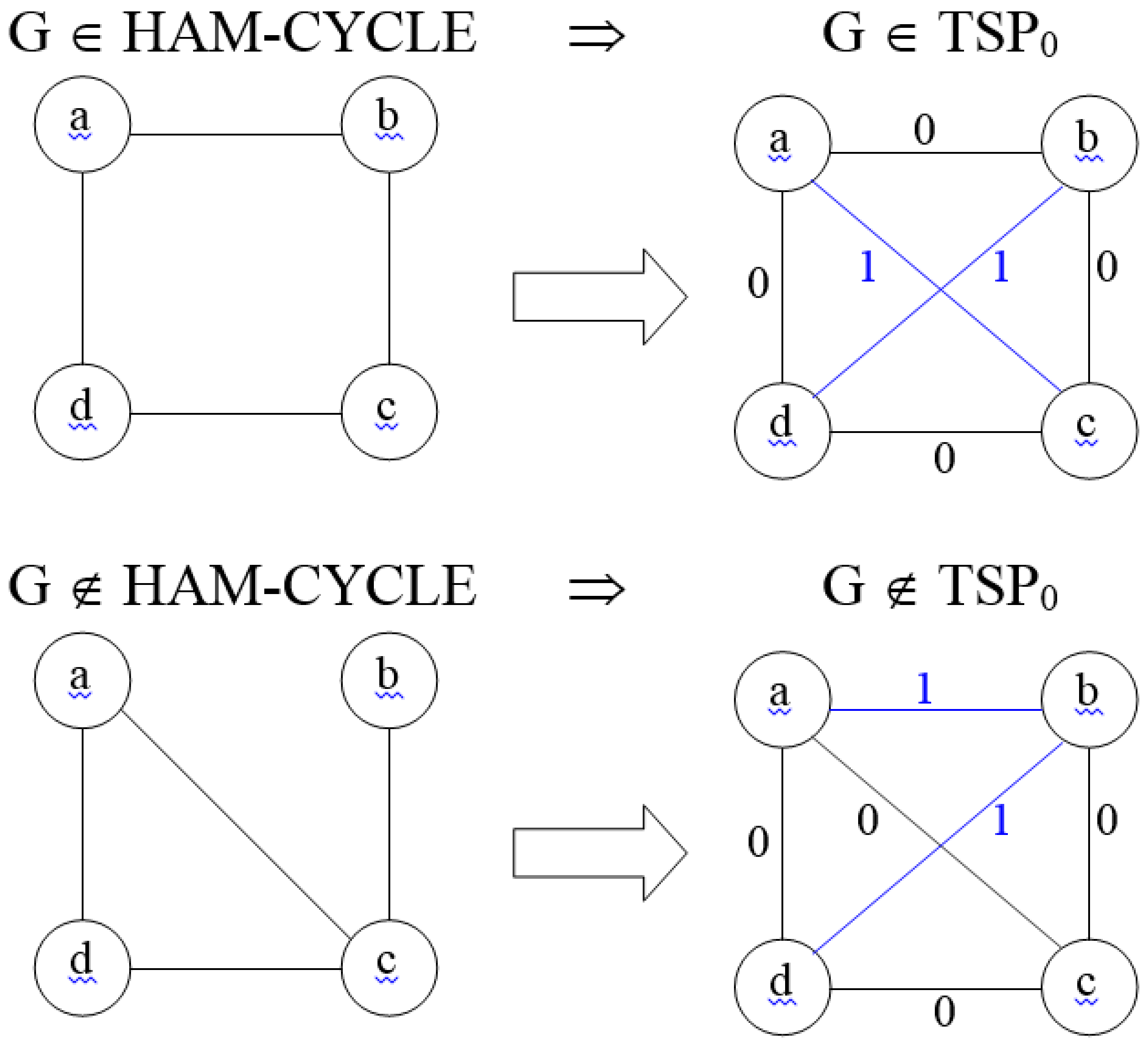
- It is a technique used to show a problem Π_1 belongs to a "certain" complexity class:
  - NP-complete, or
  - to state that Π_1 is NP-hard
- A problem Π_1 *reduced* to another problem Π_2
- Means that solution to an instance of Π_2 provides solution to an instance of Π_1
- Reduction refers to *polynomial-time* reduction

  i.e., Π_1 is *reduced* to Π_2 in O(n^k)



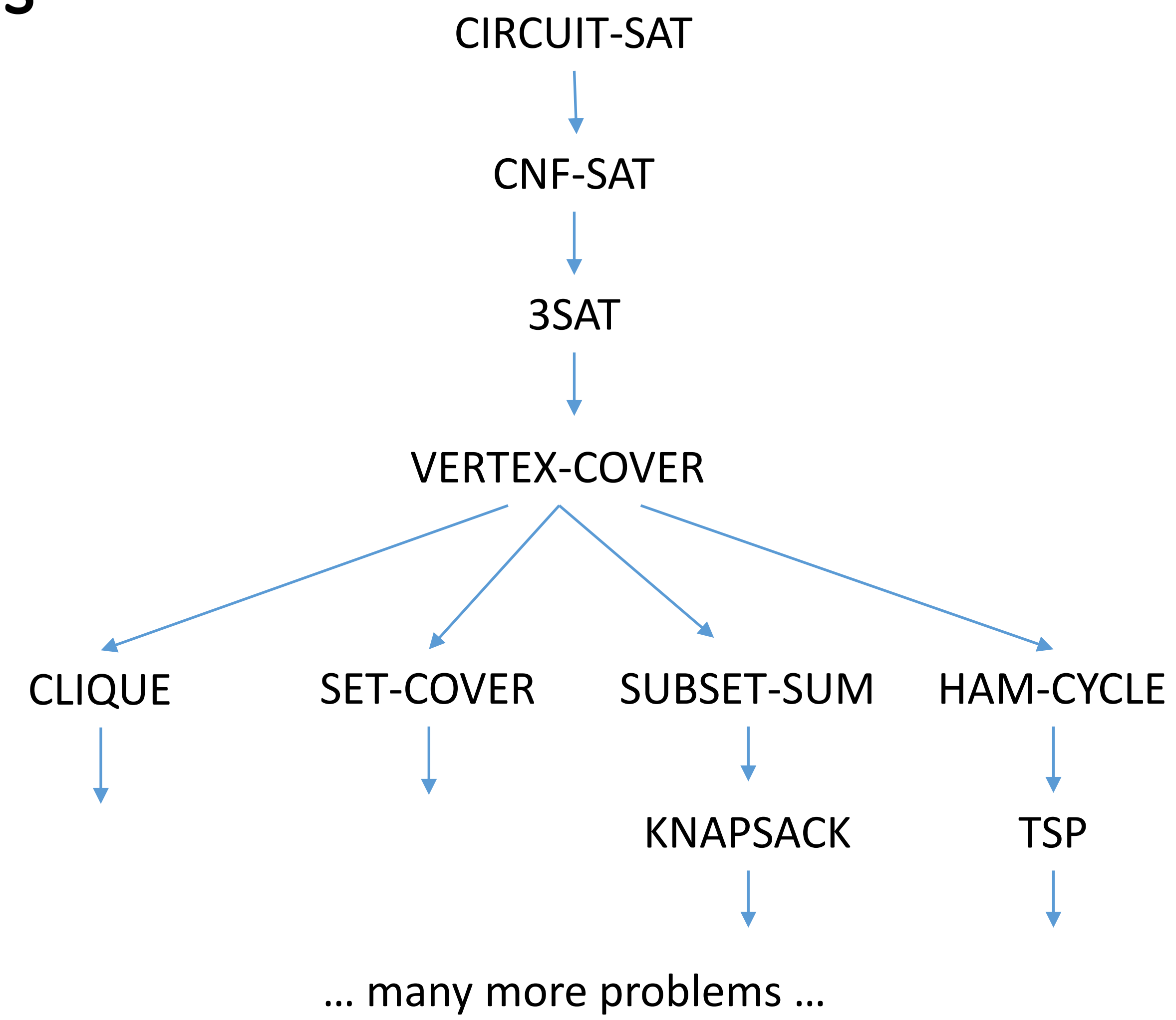
Reduction - example

- An instance of HAM-CYCLE can be reduced to a "binary" instance of TSP
- Binary means that the weights are 0 or 1
- A graph has a Hamiltonian cycle if and only if the corresponding instance of TSP has a cycle of weight 0, which is the minimum weight
- Reduction takes place in polynomial time:
  - Given the unweighted graph, we can easily create a complete weighted graph
- We can do this for every NP-complete problem



NP-complete problems

- It's a family of problems
- More than 300 problems can be found in [6]
- Arrows indicate reduction
- CIRCUIT-SAT was the first problem found to be NP-complete



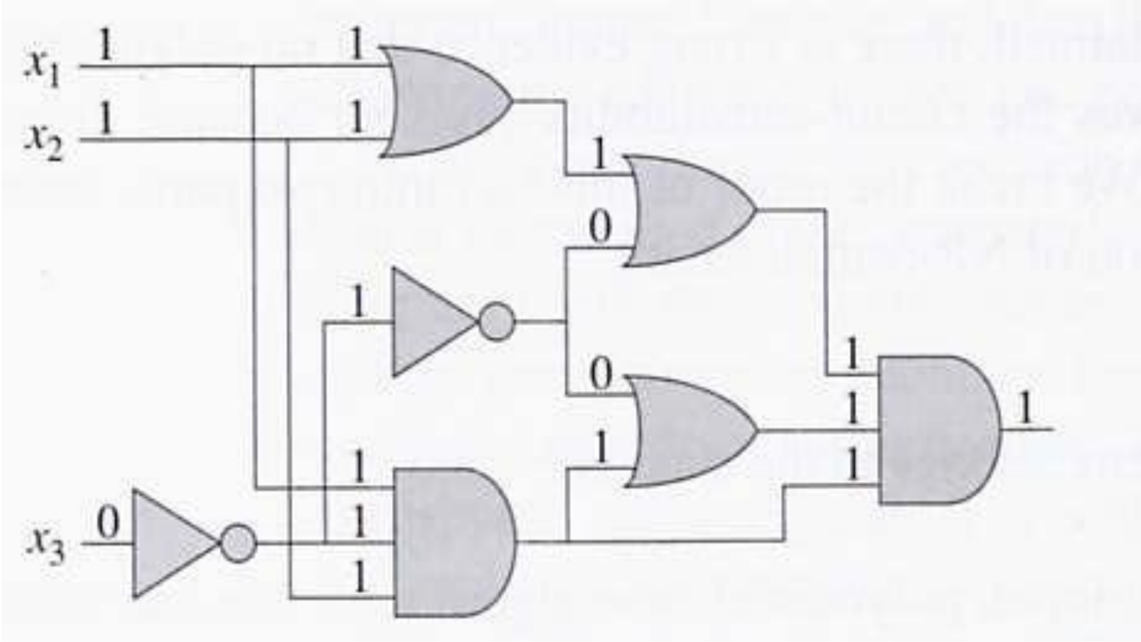
NP-complete problems

- CIRCUIT-SAT
  - Given a circuit
  - Input variables can be assigned 0 or 1
  - Is there any satisfying assignment of input variables (whose output is 1)?
- CNF-SAT
  - Given a Boolean formula
  - Variables can be assigned 0 or 1
  - Is it satisfiable?
- Example:

$f = (x_1 \to x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4) \wedge \neg x_2$

True for $x_1 = 0$, $x_2 = 0$, $x_3 = 1$, $x_4 = 1$

Satisfiable circuit:



NP-complete problems

- 3SAT
  - Conjunctive normal form (CNF):

    AND (product) of clauses
  - Clause: OR of one or more literals
  - Literal: A variable or its negation
  - 3-CNF formula: A CNF, where each clause has exactly 3 literals
  - Given a 3-CNF formula, is it satisfiable?
- **Example:**

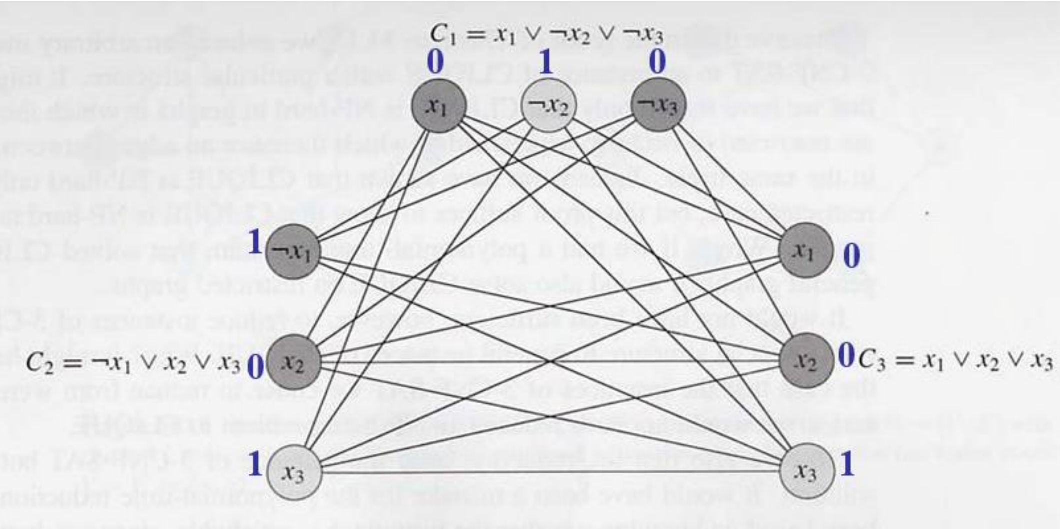
$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$

is satisfiable for $x_2 = 0$ and $x_3 = 1$

A clique of size k = 3 is formed with lightly shaded vertices

CLIQUE

- **Complete graph:** G = (V, E) is *complete* if $\forall$ u,v $\in$ V, (u,v) $\in$ E.
- **Clique:** A set of vertices $V' \subseteq V$, s.t. $G' = (V', E')$, $E' \subseteq E$, is complete.
- **Clique size:** number of vertices of V′
- **Optimization problem:** Given G = (V, E), find the clique of *maximum* size.
- **Decision problem (language):** CLIQUE = $\{\langle G,k \rangle$: G is a graph with a clique of size $k\}$



Dealing with NP-complete problems

State of the art:

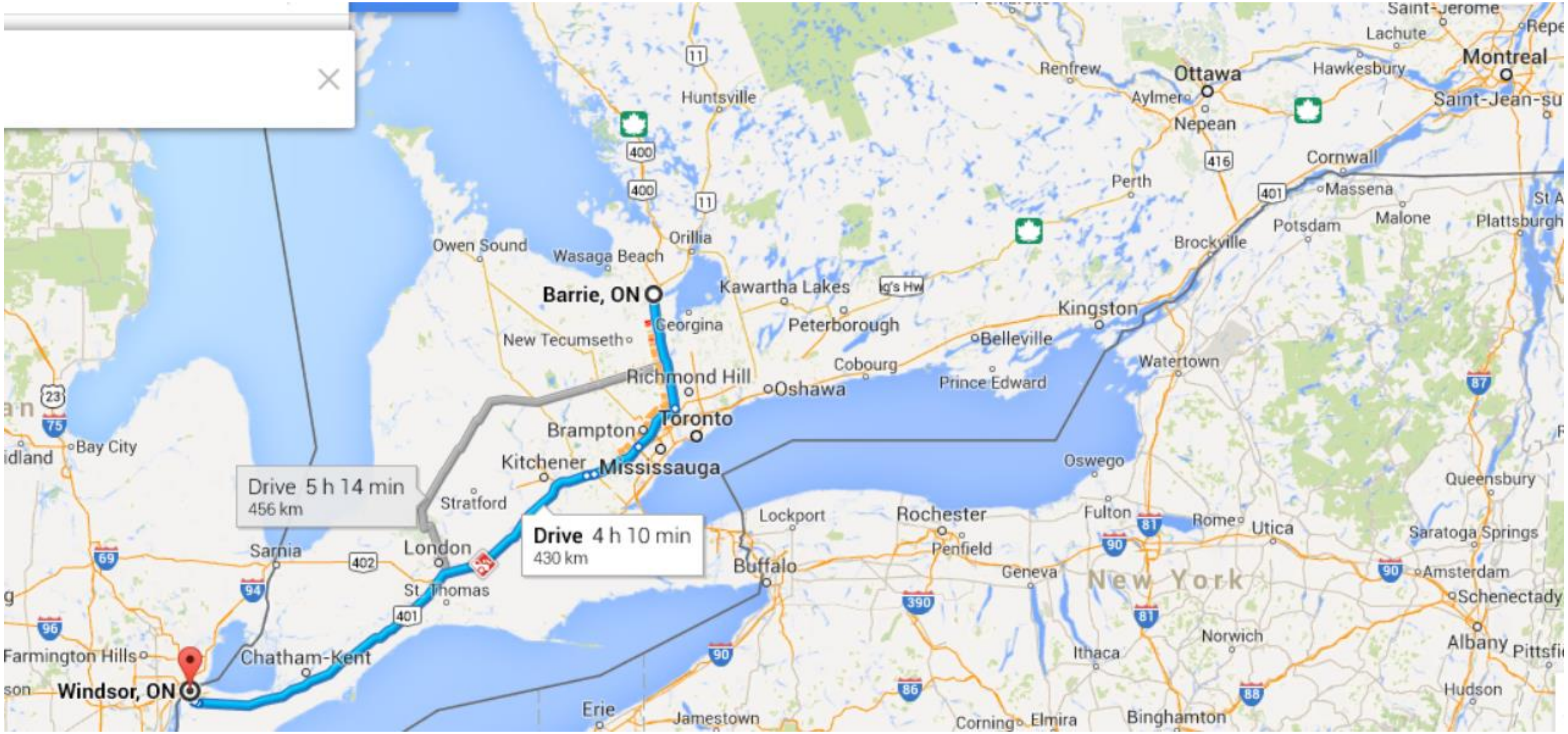
- So far, no algorithm can solve any NP-complete problem in polynomial time
- If there was one such a problem solvable in polynomial time
- All NP-complete problems could be solved in polynomial time
- However, we have no such an algorithm!... and so we use…

Main approaches

- Exhaustive search (uninformed)
  - Breadth-first search
  - Depth-first search
  - Backtracking: a variant of depth-first search
  - Other variants of these algorithms
- Informed search
  - Best-first search
    - Greedy best-first search
    - A^* search
  - Heuristics
- Approximation algorithms
- Advanced heuristics/approaches
  - Hill-climbing
  - Simulated annealing
  - Local beam search
  - Genetic algorithms
  - Particle swarm optimization
  - Ant colony optimization
  - Tabu search
  - Many others…
  - More details in [7]

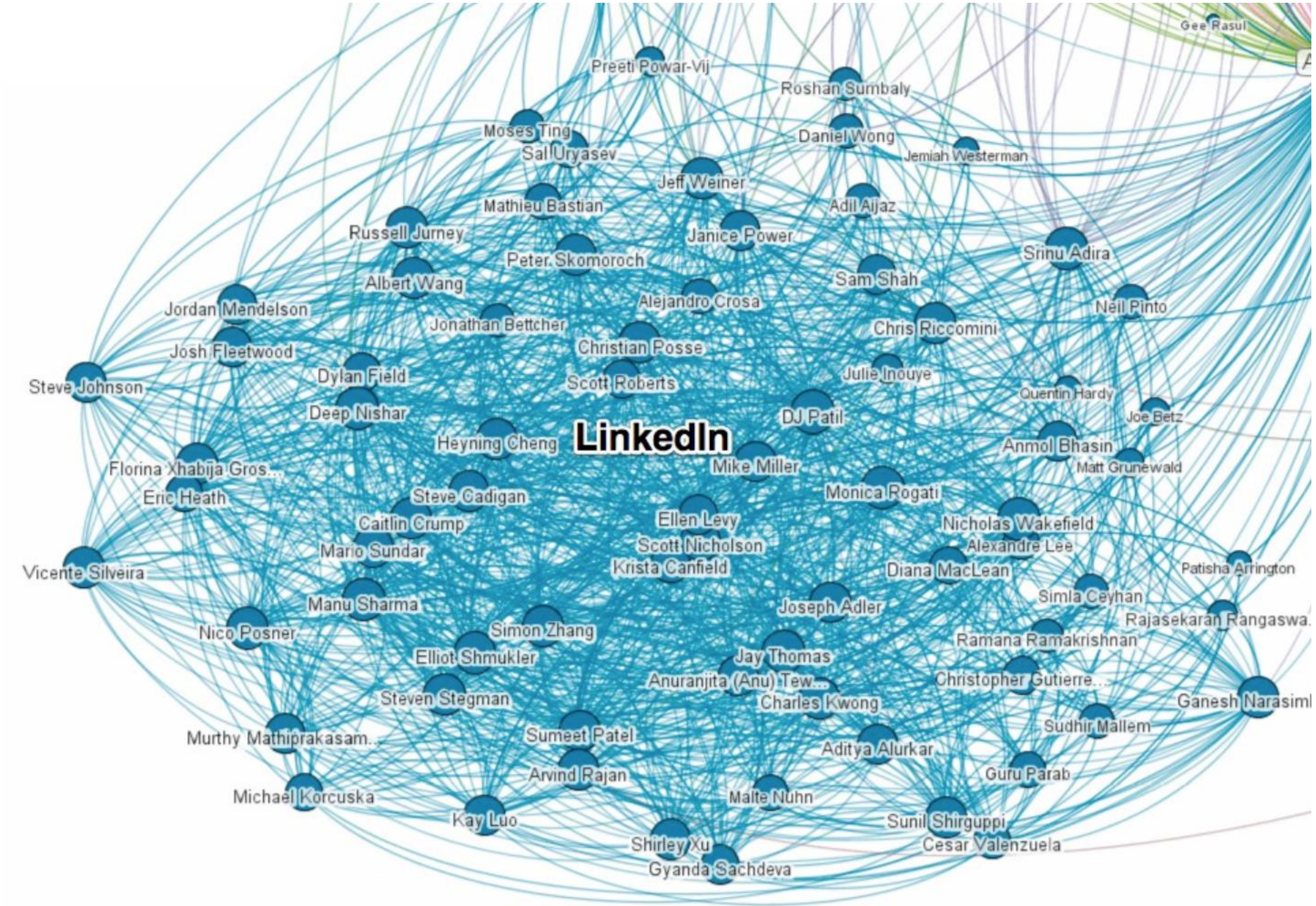
Shortest path – application and comparison

- Find a minimum cost path from Windsor to Barrie. Can be found in polynomial time.
- But finding the minimum cost path from Windsor, visiting every other city in, say Ontario, and coming back to Windsor is NP-complete!



Applications of Clique – Social Networks

- Consider Linkedin
- Suppose we wanted to find a complete subnetwork
- Example:
  - is there a group of students in this class who are all connected?
  - What is the maximum number of students?
- How to solve this problem?
  - Find all users of Linkedin and their connections
  - Find max clique of size k



References

1. Algorithm Design and Applications by M. Goodrich and R. Tamassia, Wiley, 2015.
2. Data Structures and Algorithms in Java, 6th Edition, by M. Goodrich and R. Tamassia, Wiley, 2014. *(On reserve in the Leddy Library)*
3. Data Structures and Algorithm Analysis in Java, 3rd Edition, by M. Weiss, Addison-Wesley, 2012.
4. Algorithm Design by J. Kleinberg and E. Tardos, Addison-Wesley, 2006.
5. Introduction to Algorithms, 2nd Edition, by T. Cormen et al., McGraw-Hill, 2001.
6. Computers and Intractability, by Michael Garey et al., Freeman, NY, 1979.
7. Artificial Intelligence: A Modern Approach, 3rd Edition, S. Russell and P. Norvig, Prentice Hall, ISBN: 0136067387, 2010.