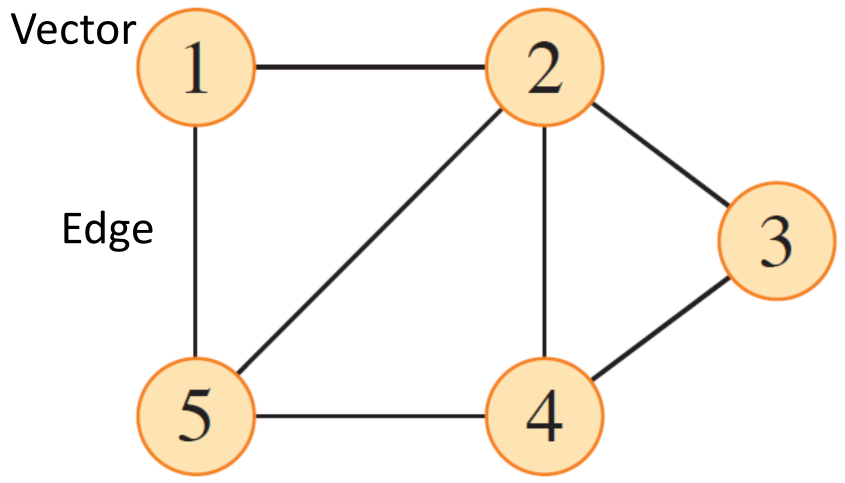
CS211FZ: Data Structures and Algorithms II
5- Graphs
Elementary Graph Algorithms

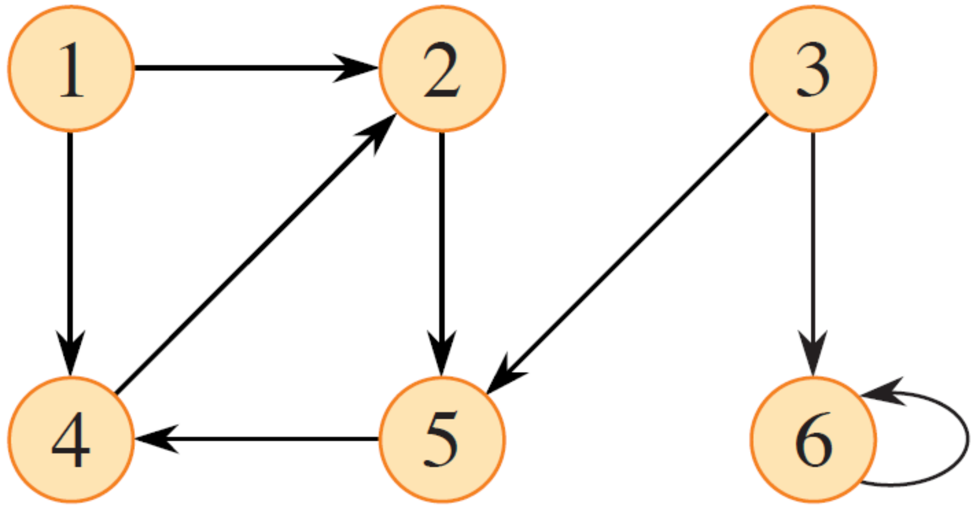
LECTURER: CHRIS ROADKNIGHT

CHRIS.ROADKNIGHT@MU.IE

Examples of graphs



An undirected graph (bidirectional)



A directed graph

Representations of graphs

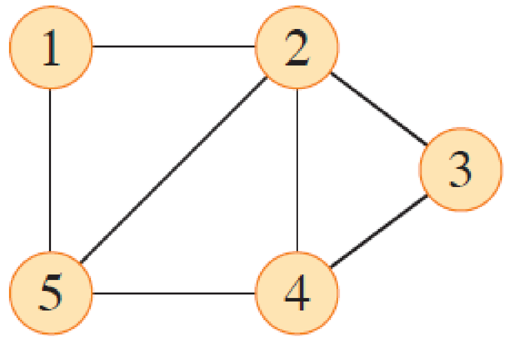
There are two standard ways to represent a graph $G = (V, E)$: as a collection of **adjacency lists** or as an **adjacency matrix**. Either way applies to both directed and undirected graphs.

$$V = vertex, E = edge$$

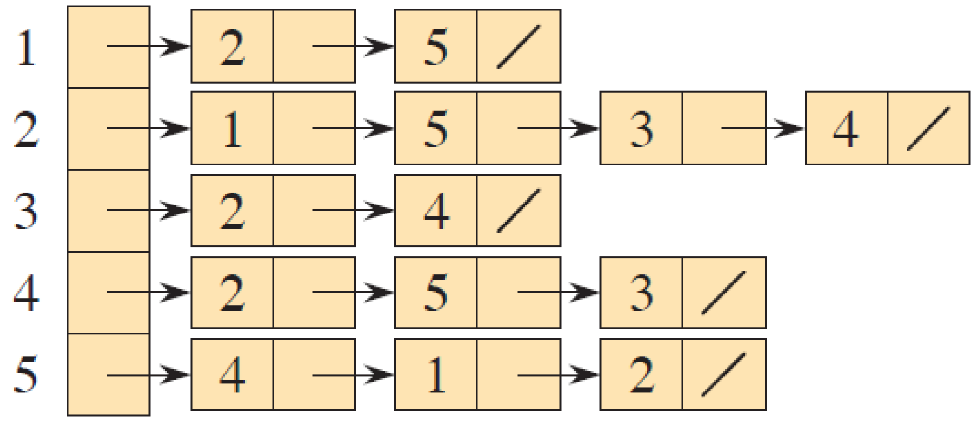
Because the adjacency-list representation provides a compact way to represent sparse graphs — those for which $|E|$ is much less than $|V|^2$ — it is usually the method of choice.

We may prefer an adjacency-matrix representation, however, when the graph is dense —$|E|$ is close to $|V|^2$ — or when we need to be able to tell quickly if there is an edge connecting two given vertices. (arrays vs linked lists at scale)

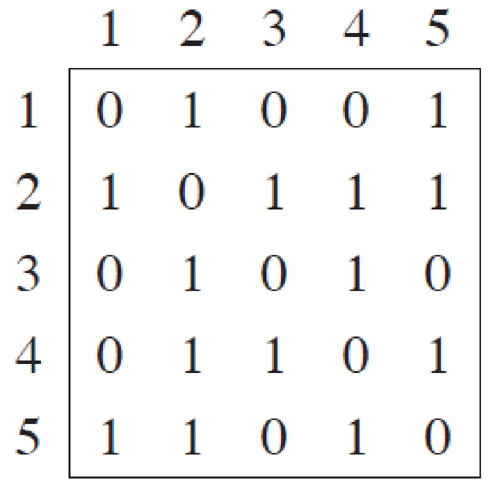
Example - undirected graph



(a)



(b)



$$\begin{array}{c|ccccc|}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 0 & 0 & 1 \\
2 & 1 & 0 & 1 & 1 & 1 \\
3 & 0 & 1 & 0 & 1 & 0 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 1 & 1 & 0 & 1 & 0 \\
\hline
\end{array}$$

(c)

Two representations of an undirected graph. **(a)** An undirected graph G having five vertices and seven edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

Adjacency-list representation

The adjacency-list representation of a graph $G = (V\ , E)$ consists of an array of Adj of $|V|$ lists, one for each vertex in V. For each $u \in V$, the adjacency list $Adj[u]$ contains all the vertices v such that there is an edge $(u, v) \in E$. That is $Adj[u]$ consists of all the vertices adjacent to u in G.

If G is a directed graph, the sum of the lengths of all the adjacency list is $|E|$, since an edge of the form (u, v) is represented by having v appear in $Adj[u]$.

If G is an undirected graph, the sum of the lengths of all the adjacency list is $2|E|$, since if (u, v) is an undirected edge, then u appears in v's adjacency list and vice versa.

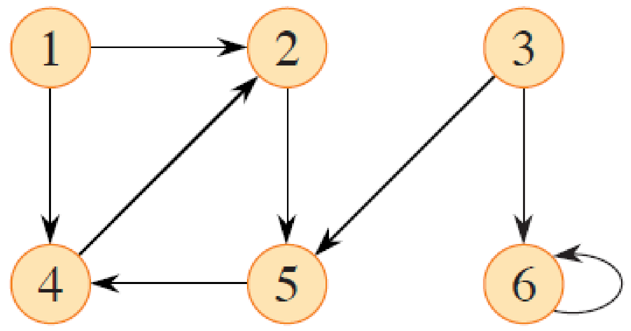
Adjacency-matrix representation

For the adjacency-matrix representation of a graph $G = (v\,, E)$, we assume that the vertices are numbered $1, 2, \ldots, |V|$ in some arbitrary manner. Then the adjacency-matrix representation of a graph G consists of a $|V| \times |V|$ matrix $A = (a_{ij})$ such that:

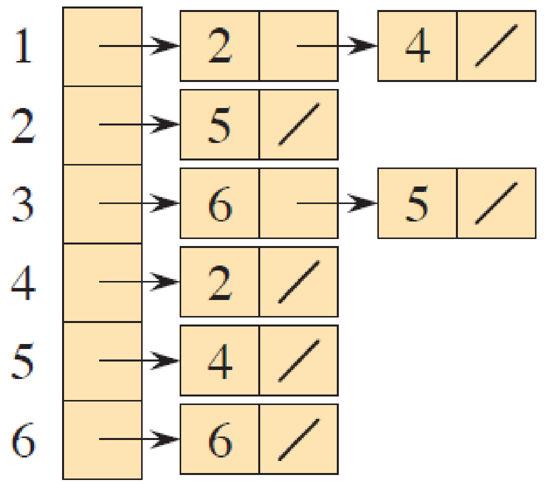
$$a_{i,j} = \begin{cases} 1 & if\,(i,j) \epsilon E \\ 0 & otherwise \end{cases}$$

The adjacency matrix of a graph requires $\Theta(V^2)$ memory, independent of the number of edges in the graph.

Example - directed graph



(a)



(b)

$$\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 1 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 1 & 0 \\
3 & 0 & 0 & 0 & 0 & 1 & 1 \\
4 & 0 & 1 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
\end{array}$$

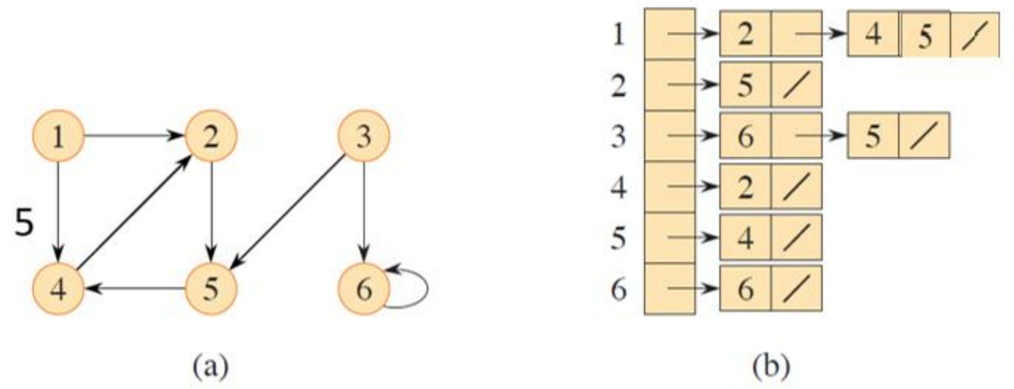
(c)

Two representations of a directed graph. **(a)** A directed graph G with 6 vertices and 8 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

Weighted graphs

We can readily adapt adjacency lists to represent **weighted graphs**, that is, graphs for which each edge has an associated **weight**, typically given by a **weighted function** w: $E \rightarrow R$.

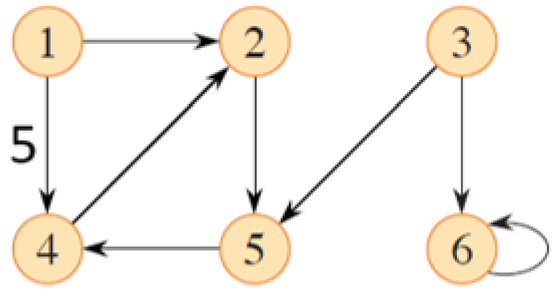
We can simply store the weight $w(u, v)$ of the edge $(u, v) \in E$ with vertex v in u's adjacency list. The adjacency-list representation is quite robust in that we can modify it to support many other graph variants.



Weighted graphs

We can readily adapt adjacency matrix to represent **weighted graphs**

We can simply store the weight $w(u,v)$ of the edge $(u,v) \in E$ with vertex v in a second matrix.



(a)

$$\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 0 & 0 & 0 & 1 & 0 \\
2 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 0 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
\end{array}$$

(b) Weights

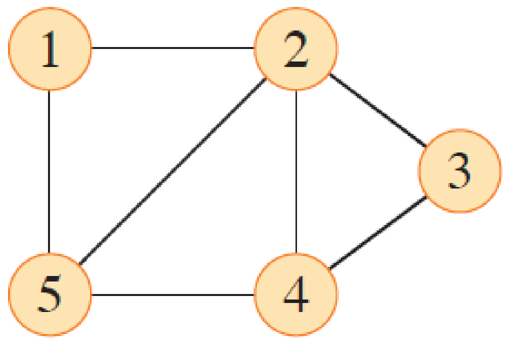
$$\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 1 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 1 & 0 \\
3 & 0 & 0 & 0 & 0 & 1 & 1 \\
4 & 0 & 1 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
\end{array}$$

(c) Edges

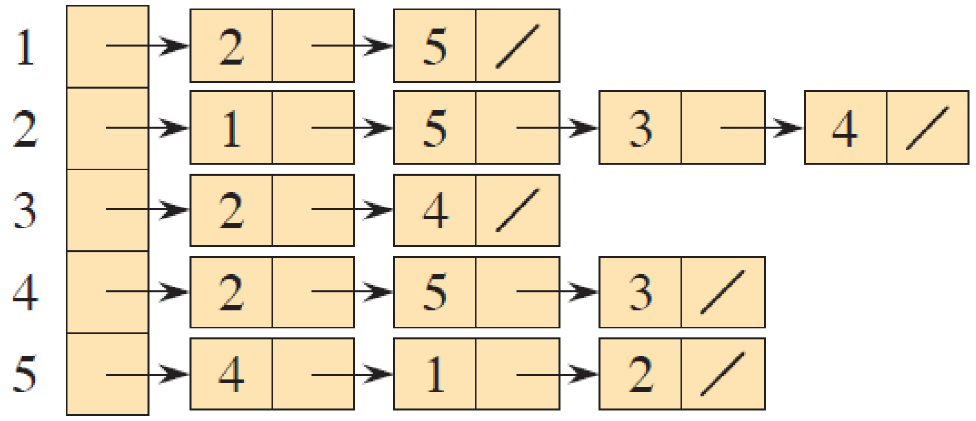
Finding if edge exists

A potential disadvantage of the adjacency-list representation is that it provides no quicker way to determine whether a given edge (u, v) is present in the graph than to search for v in the adjacency list $Adj[u]$. (O(n))

Finding if edge exists **O(n)** vs **O(1)**



(a)



(b)

$$\begin{array}{c|ccccc|}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 0 & 0 & 1 \\
2 & 1 & 0 & 1 & 1 & 1 \\
3 & 0 & 1 & 0 & 1 & 0 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 1 & 1 & 0 & 1 & 0 \\
\hline
\end{array}$$

(c)

Representing attributes

Most algorithms that operate on graphs need to maintain attributes for vertices and/or edges. We indicate these attributes using notation such as $v.d$ for an attribute d of a vertex v. We indicate edges as pairs of vertices, we use the same style of notation. For example, if edges have an attribute f,we use $(u, v).f$.

There is no one best way to store and access vertex and edge attributes. For a given situation, your decision will likely depend on the programming language you are using, the algorithm you are implementing, and how the rest of your program uses the graph.

For example, in an object-oriented programming language, vertex attributes might be represented as instance variables* within a subclass of a Vertex class.

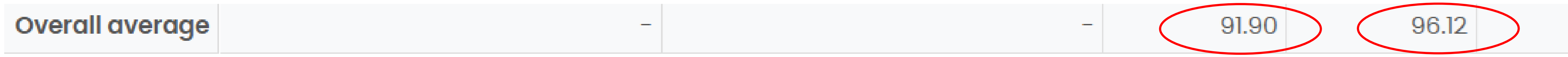
**variables that belong to an instance of a class. These variables are initialized when an object of the class is created. The initialization is typically done within a constructor*

Start here 21/4/2025

Labs – Continual assessment

“the labs are very hard, could you give us more feedback on them”

A. Are they hard? CA1 CA2



Overall average	-	-	91.90	96.12

So…..either, they are hard but you are all geniuses (possible) or…
..they arnt too hard.

I think the latter.
Mainly because: You receive TA support, templates, time to finish offline (open book time)

Labs - Continual Assessment

B. More feedback and support

Lecture on Sunday 27th will be entirely a review, including review of Coursework answers so far, sample exam question and any other questions (email me them so I can prepare answers and appear clever). For example, how would I answer this question?

Rank the following functions by order of growth.

		$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$(3/2)^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	
$\ln \ln n$		$n \cdot 2^n$	$n^{\lg \lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
	$2^{\sqrt{2 \lg n}}$	n	2^n	$n \lg n$	$2^{2^{n+1}}$

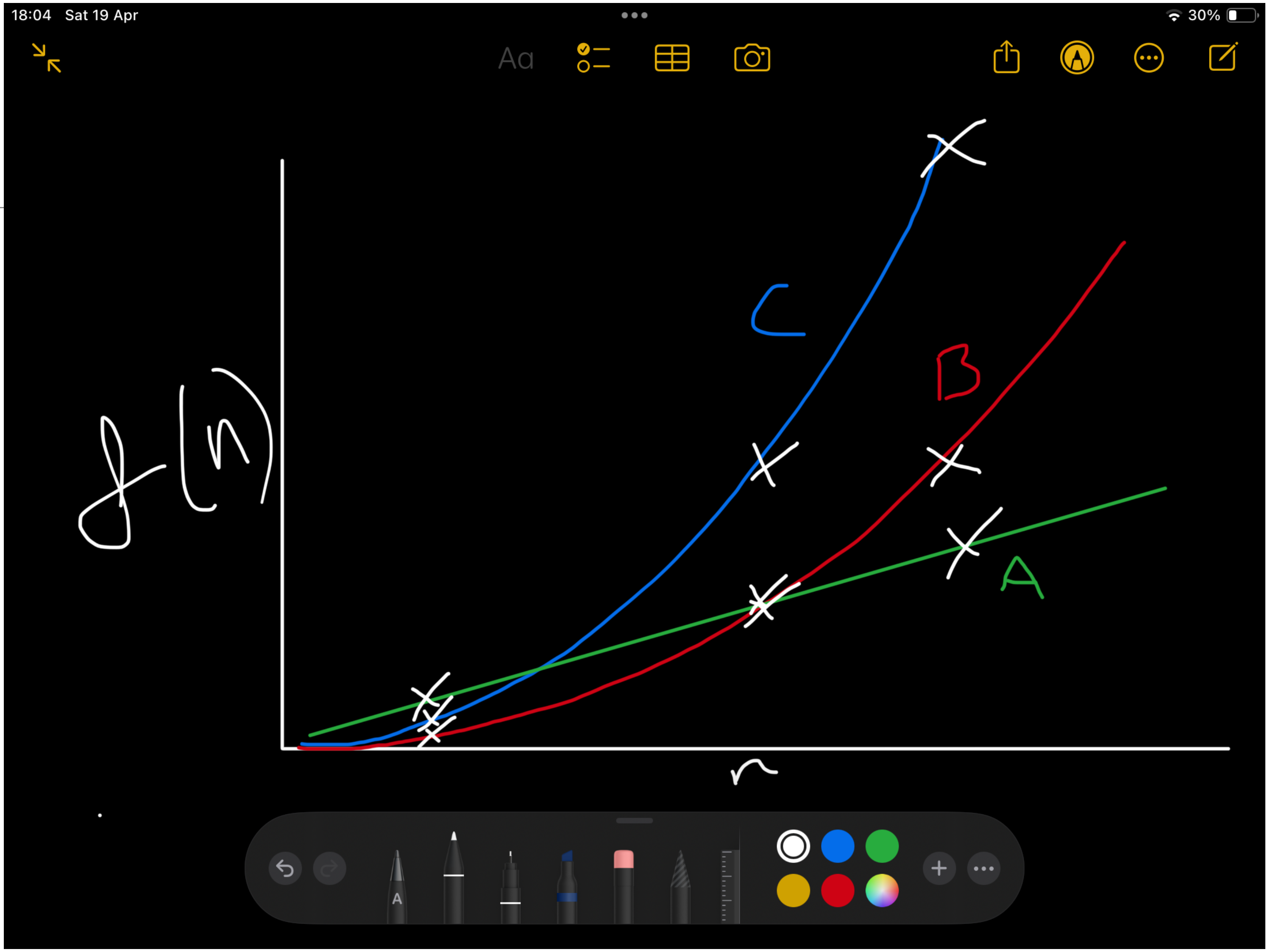
Top Tip

Approximate!!!

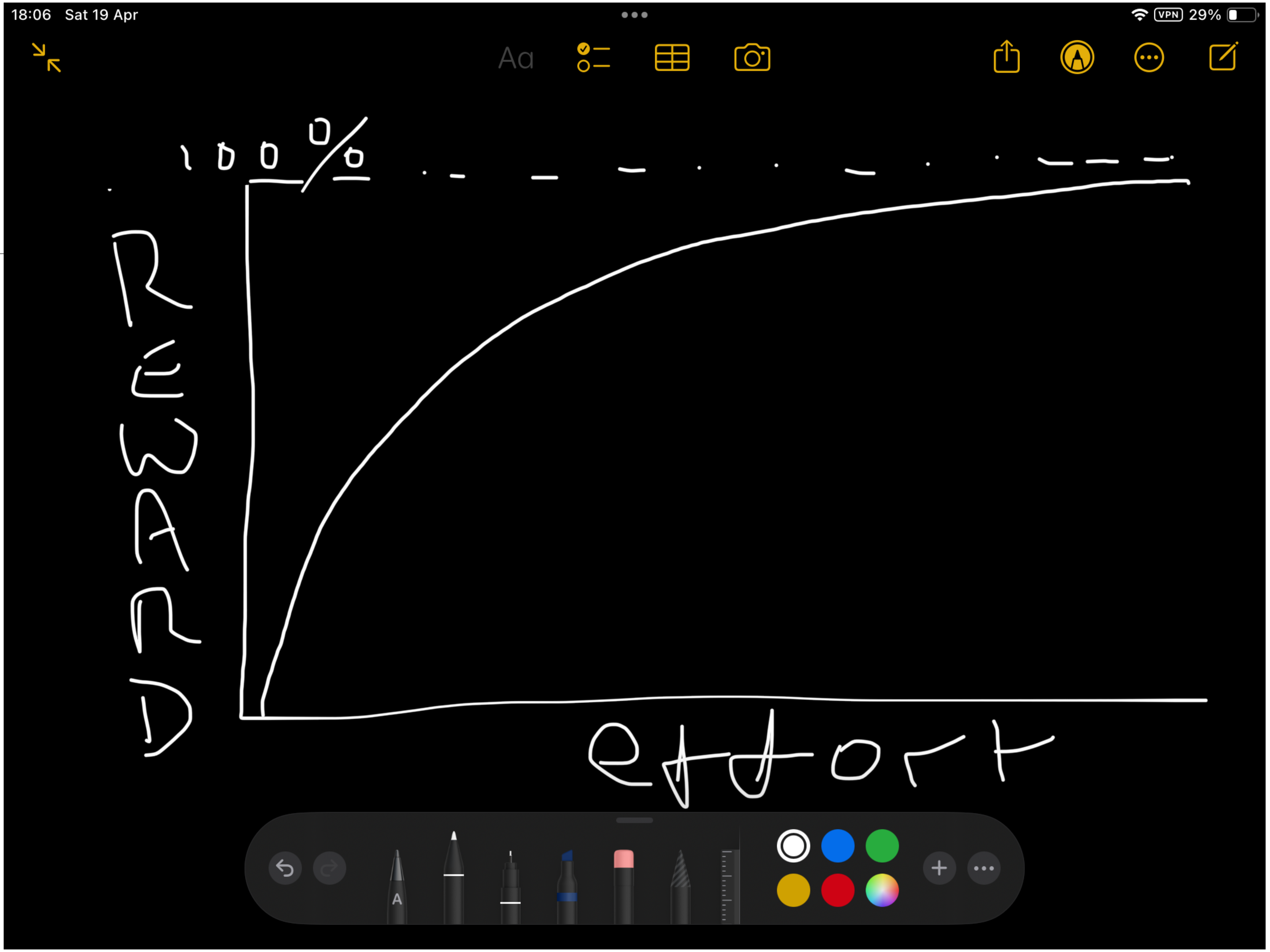
Unless a question specifically asks for exact quantities (and give no marks for being close)

……. approximate

This graph is an approximation



Question design



RECAP

BFS, DFS

Breadth-first search (1)

Breadth-first search is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms. **Prim's minimum-spanning tree algorithm and Dijkstra's single-source shortest paths algorithm use ideas similar to those in breadth-first search.**

Given a graph $G = (V, E)$ and a distinguished **source** vertex s, breadth-first search systematically explores the edges of G to "discover" every vertex that is reachable from s. It computes the distance from s to each reachable vertex. For any vertex v 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 (a path containing smallest number of edges). The algorithm works on both directed and undirected graphs.

Breadth-first search (2)

Breadth-first search is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. That is, the algorithm discovers all vertices at **distance $\boldsymbol{k}$** from s before discovering any vertices at **distance $\boldsymbol{k+1}$.**

To keep track of progress, the algorithm colours each vertex **white, gray, or black.** All vertices start out **white** and (if connected) later become **gray** and then **black**. A vertex is **discovered** the first time it is encountered during the search, at which time it becomes non-white. Gray and black vertices, therefore, have been discovered, but the algorithm distinguishes between them to ensure that the search proceeds in a breadth-first manner.

Breadth-first search – Pseudocode

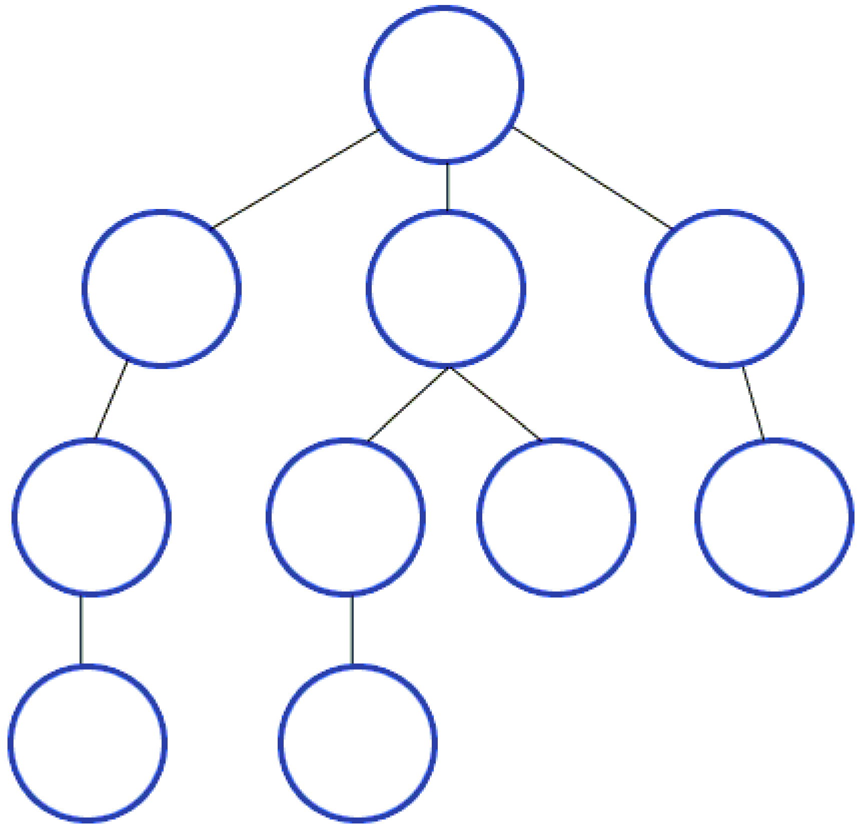
```
BFS($G, s$)
                                                  // Initialize all vertices in the graph G
for each vertex $u \in G.V - \{s\}$              //except the source vertex s
    $u.color$ = WHITE
    $u.d = \infty$          //distance to source, initially unreachable
    $u.\pi$ = NIL           //parent
$s.color$ = GRAY            //Initialise the source variable
$s.d = 0$ //Distance from source to itself is 0
$s.\pi$ = NIL //Source has no parent
$Q = \emptyset$  //Initialize an empty queue Q to manage vertices to explore
ENQUEUE($Q, s$)  //Add the source vertex s to the queue to start BFS
while $Q \neq \emptyset$  //Process vertices in the queue until it's empty
    $u$ = DEQUEUE($Q$)  // Remove the next vertex u from the front of queue
    for each vertex $v$ in $G.Adj[u]$   // search the neighbors of $u$
        if $v.color$ == WHITE           // is $v$ being discovered now?
            $v.color$ = GRAY   // Mark v as discovered (GRAY)
            $v.d = u.d + 1$  // Distance of v as distance of u plus 1
            $v.\pi = u$  // Set u as the parent of v in the BFS tree
            ENQUEUE($Q, v$)             // $v$ is now on the frontier
    $u.color$ = BLACK                   // $u$ is now behind the frontier
```

Visualisations

https://www.youtube.com/watch?v=gLHZdQeaqJA

https://www.cs.usfca.edu/~galles/visualization/BFS.html

https://visualgo.net/en/dfsbfs



https://commons.wikimedia.org/wiki/File:Breadth-First-Search-Algorithm.gif

An Example

The following slides illustrate the operation of BFS on an undirected graph. Each part shows the graph and the queue Q at the beginning of each iteration of the **while** loop of lines 10-18.

Vertex distances appear within each vertex and below vertices in the queue. The tan region surrounds the frontier of the search, consisting of the vertices in the queue.

The light blue region surrounds the vertices behind the frontier, which have been dequeued.

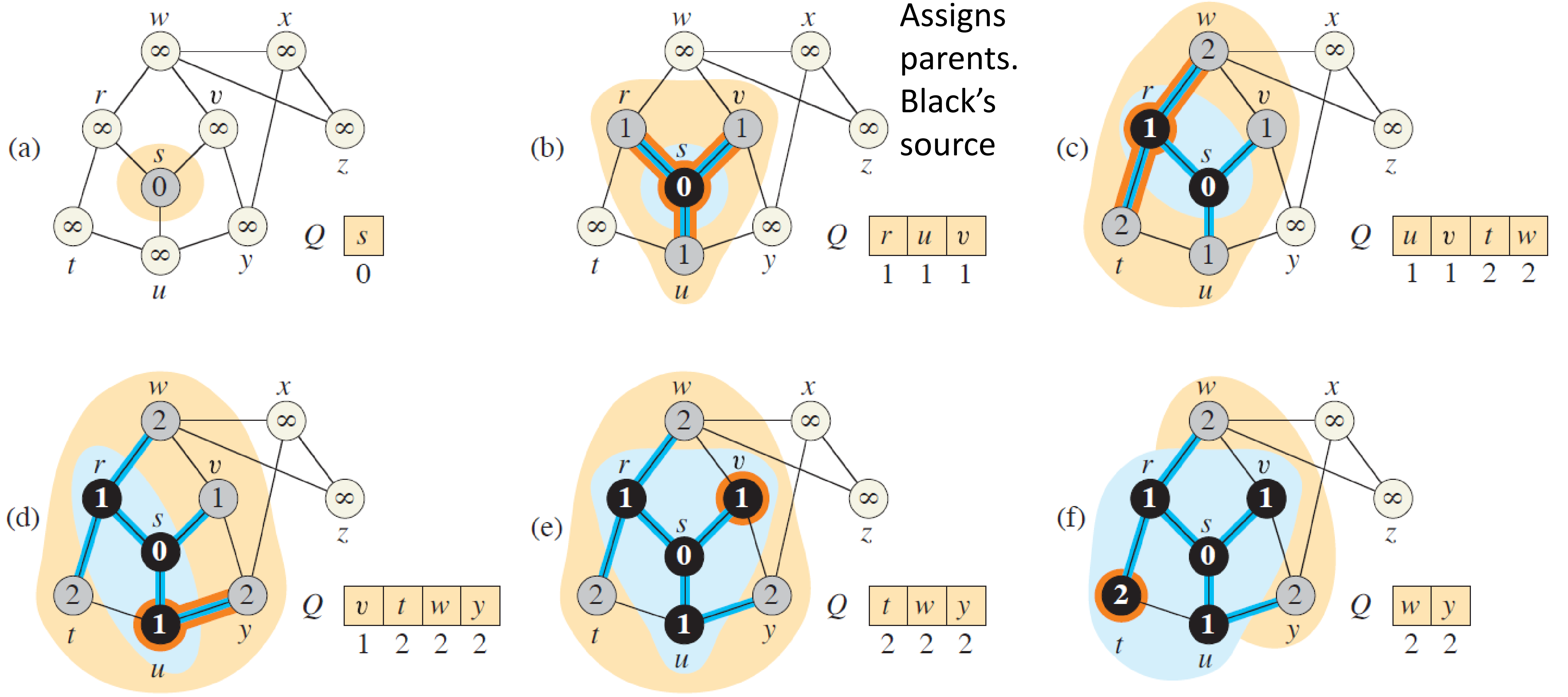
Each part highlighted in orange the vertex dequeued and the breadth-first tree edges added, if any, in the previous iteration. Blue edges belong to the breadth-first tree constructed so far.

BFS - An Example

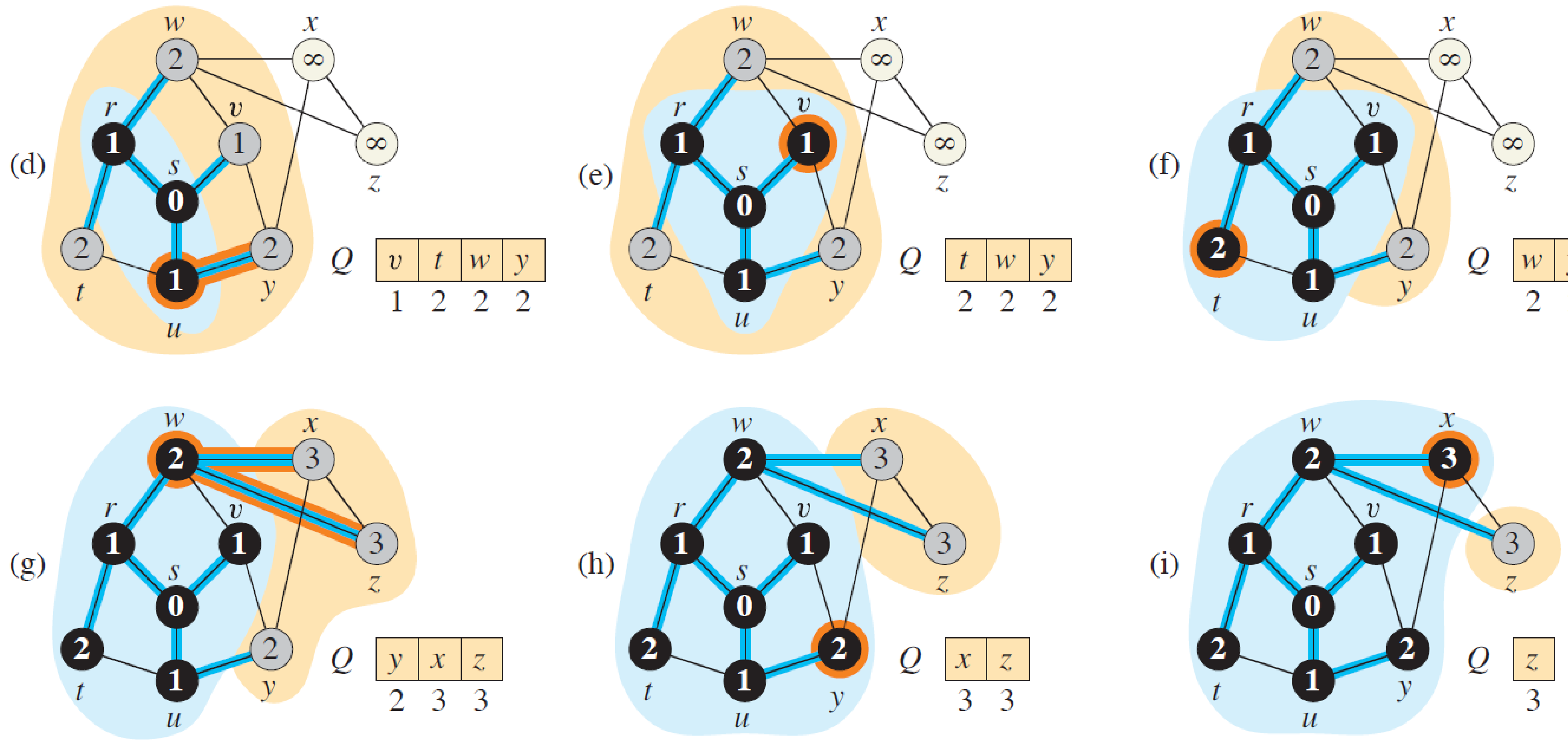
Begins all white, with no parents

Gray''s nodes. Assigns parents. Black's source

Extend frontier.

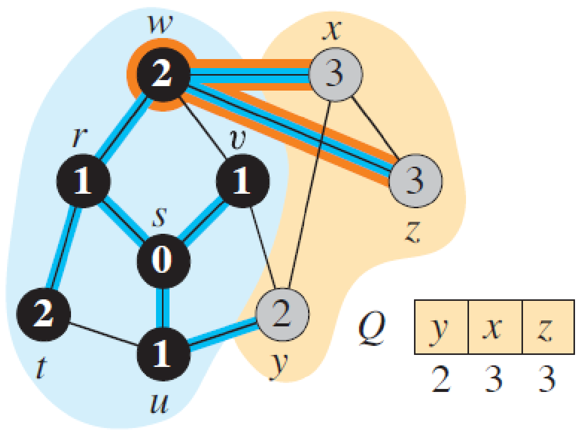


An Example (continue…)

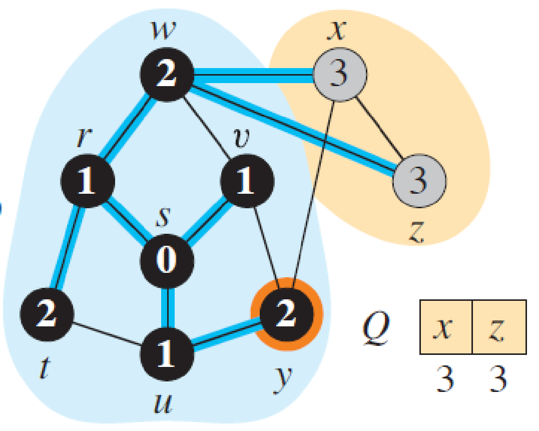


An Example (continue)

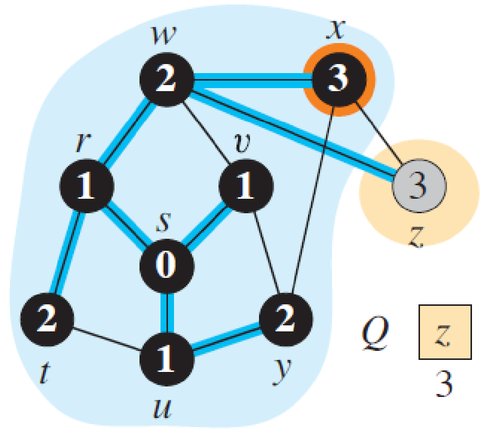
(g)



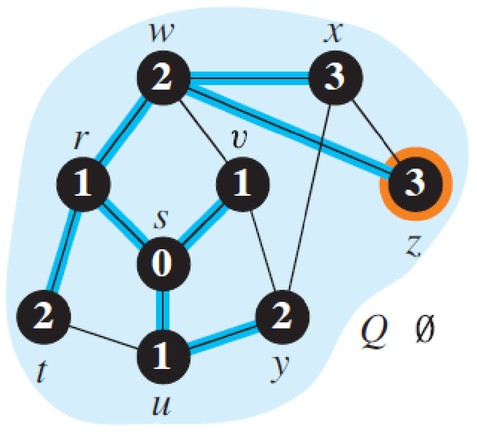
(h)



(i)



(j)



Analysis

The operations of enqueuing and dequeuing take $O(1)$ time, and so the total time devoted to queue operation 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.

The sum of the lengths of all the adjacency list is $\Theta(E)$, the total time spent in scanning adjacency list is $O(E)$. The overhead for initialization is $O(V)$, and thus the total running time of BFS procedure is $O(V + E)$.

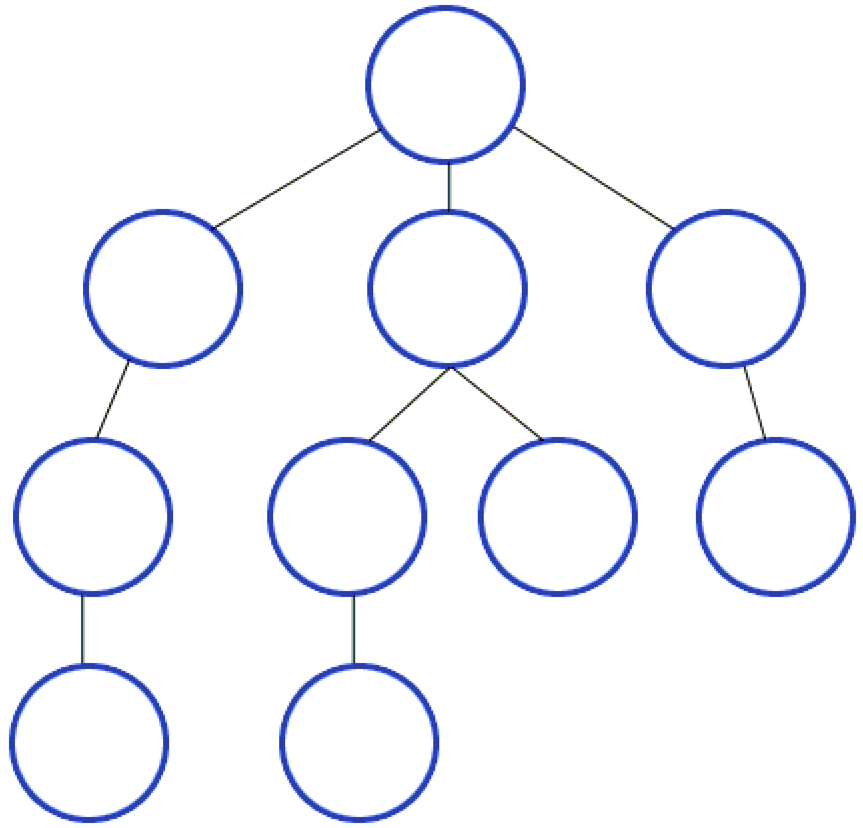
Queuing operations

Scanning adjacency for each vertex

Depth-first search (1)

The strategy followed by depth-first search is, as its name implies, to search "deeper" in the graph whenever possible. Depth-first search explores 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, the search "backtracks" 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.



Depth-first search (2)

As in breath-first search, depth-first search colours vertices during the search to indicate their state. Each vertex is white, is grayed when it is **discovered** in the search, and is blackened when it is **finished**, that is, when its adjacency list has been examined completely. This technique guarantees that each vertex ends up in exactly one depth-first tree, so that these trees are disjoint.

Besides creating a depth-first forest, depth-first search also **timestamps** each vertex. Each vertex v has two timestamps: the first timestamp $v.d$ records when v is first discovered, and the second timestamp $v.f$ records when the search finishes examining v's adjacency list (and blackens v). These timestamps provide important information about the structure of the graph and are generally helpful in reasoning about the behaviour of the depth-first search.

Depth-first search – Pseudocode

```
DFS(G)
1  for each vertex $u \in G.V$              // Iterate over all vertices in the graph G
2      $u.color$ = WHITE
3      $u.\pi$ = NIL                          //initialise
4  $time = 0$
5  for each vertex $u \in G.V$              // Iterate over all vertices again
6      if $u.color$ == WHITE                 // If the vertex u is unvisited
7          DFS-VISIT$(G, u)$                  // Call DFS-VISiT

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

DFS procedure explanation

The DFS procedure works as follows. Lines 1-3 paint all vertices white and initialize their π (parent) attributes to NIL. Line 4 resets the global time counter.

Lines 5-7 check each vertex in V in turn and, when a white vertex is found, visit it by calling DFS-VISIT.

Upon every call of DFS-VISIT(G, u) in line 7, vertex u becomes the root of a new tree in the depth-first forest. When DFS returns, every vertex u has been assigned a **discovery time** $u.d$ and a **finish time** $u.f$.

DFS-VISIT procedure explanation

In each call DFS-VISIT(G, u), vertex u is initially white.

Lines 1-3 increment the global variable time, record the new value of time as the discovery time $u.d$, and paint u gray.

Lines 4-7 examine each vertex v adjacent to u and recursively visit v if it is white.

As line 4 considers each vertex $v \in Adj[u]$, the depth-first search **explores** edge (u, v).

Finally, after every edge leaving u has been explored, lines 8-10 increment time, record the finish time in $u.f$, and paint u black.

An Example

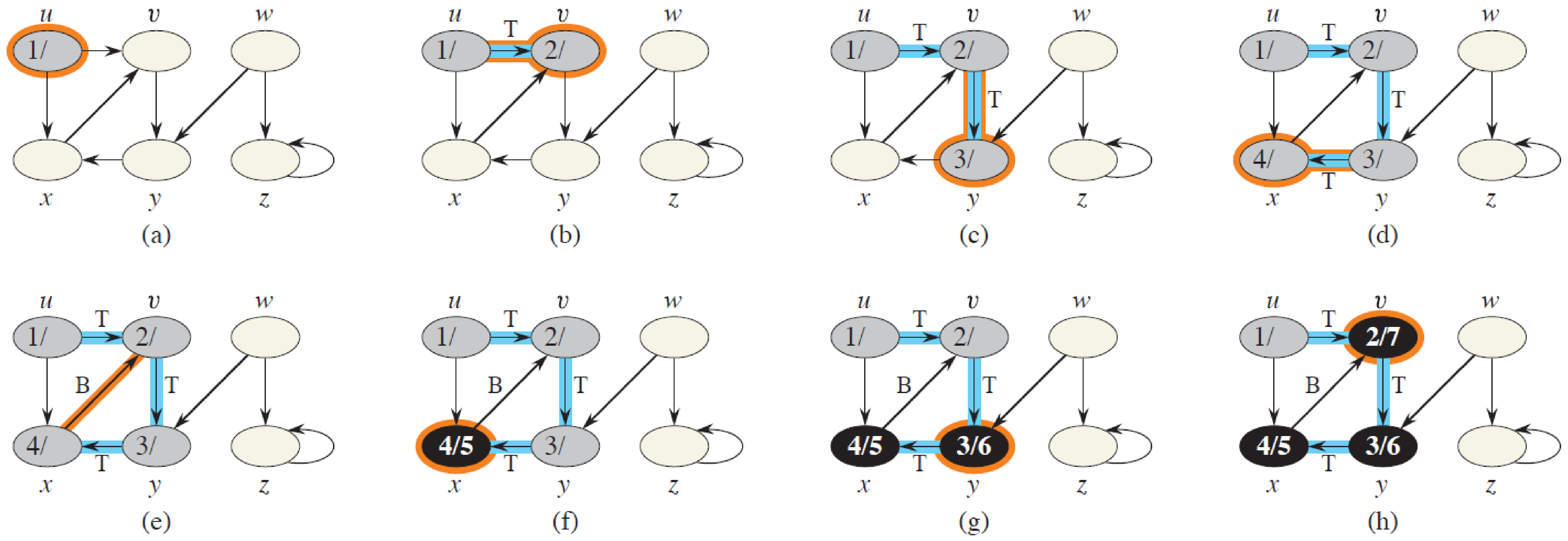
The following slides illustrate the progress of the depth-first-search algorithm DFS on a directed graph.

Edges are classified as they are explored: tree edges are labelled T, back edges B, forward edges F, and cross edges C. (B,F,C are all unused edges for different reasons)

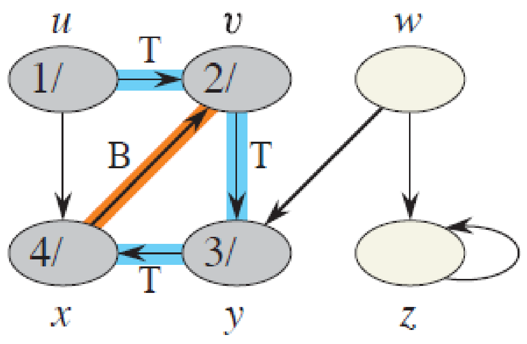
Timestamps within vertices indicate discovery time/finish times. Tree edges are highlighted in blue.

Orange highlights indicate vertices whose discovery or finish times change and edges that are explored in each step.

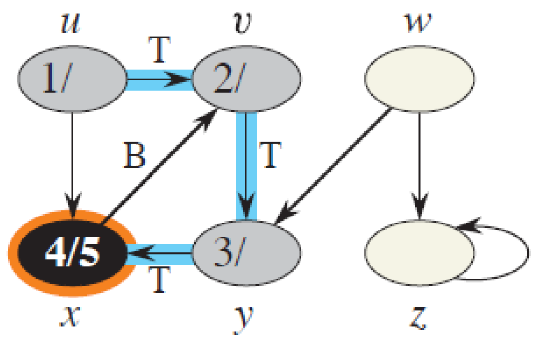
An Example (continue…)



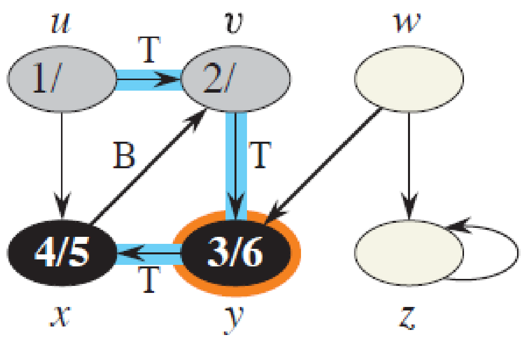
An Example (continue…)



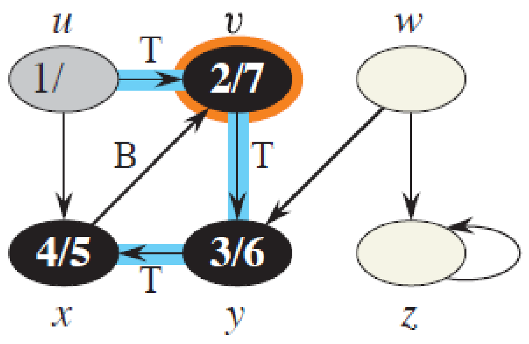
(e)



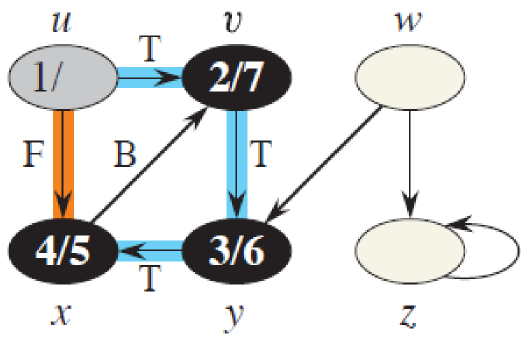
(f)



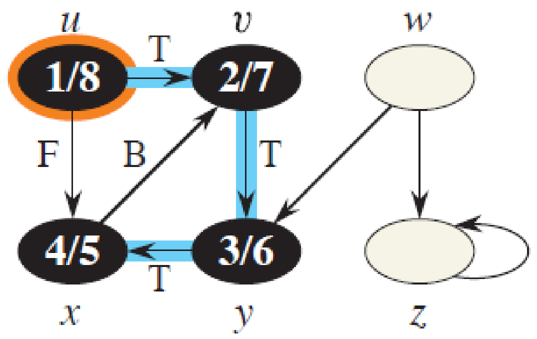
(g)



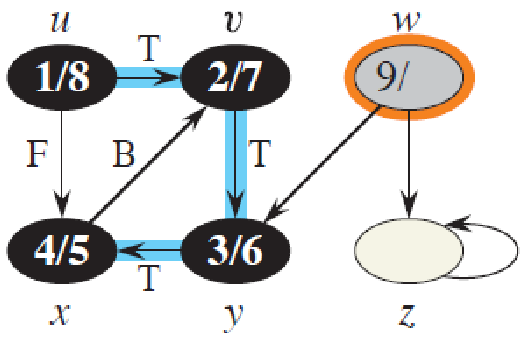
(h)



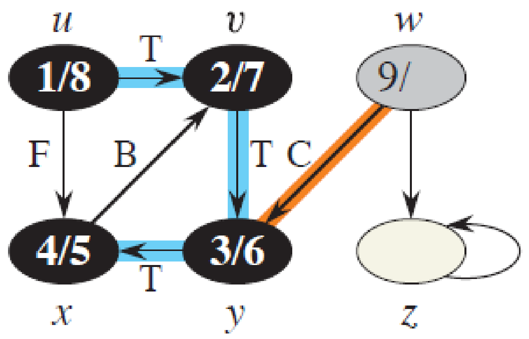
(i)



(j)



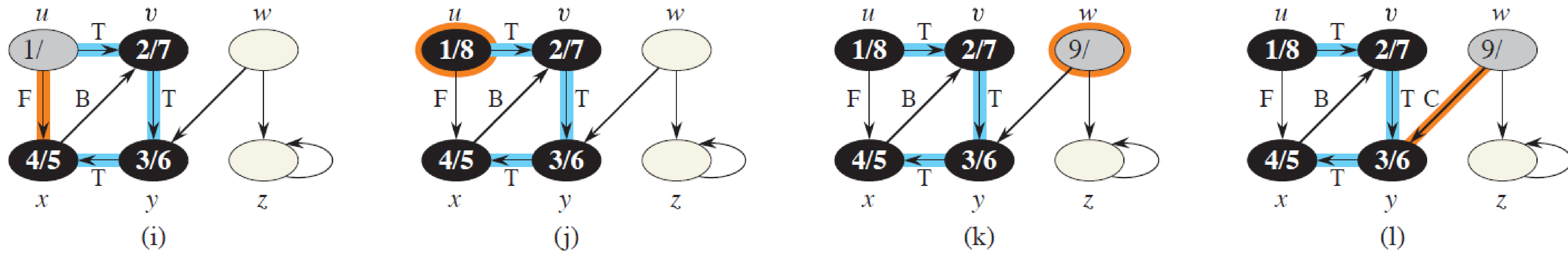
(k)

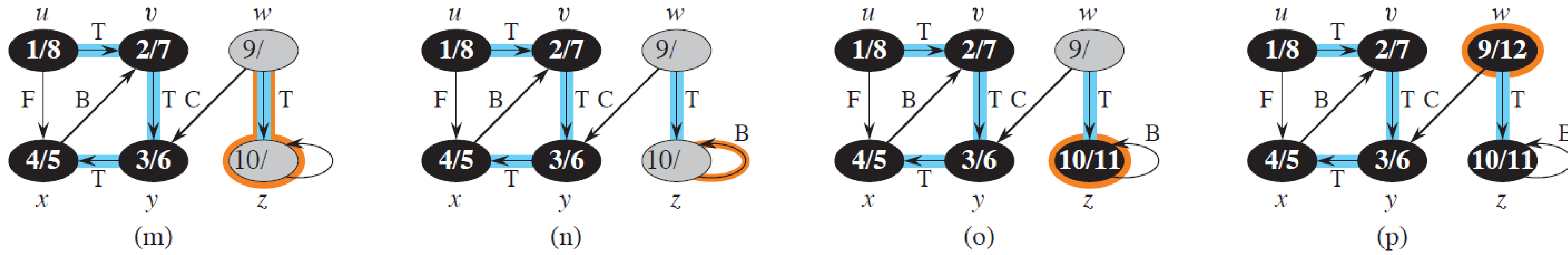


(l)

An Example (continue…)

(w) doesn't connect to (y) because they are on different branches of the tree, and the structure of the tree, as explored by DFS, keeps them separate.



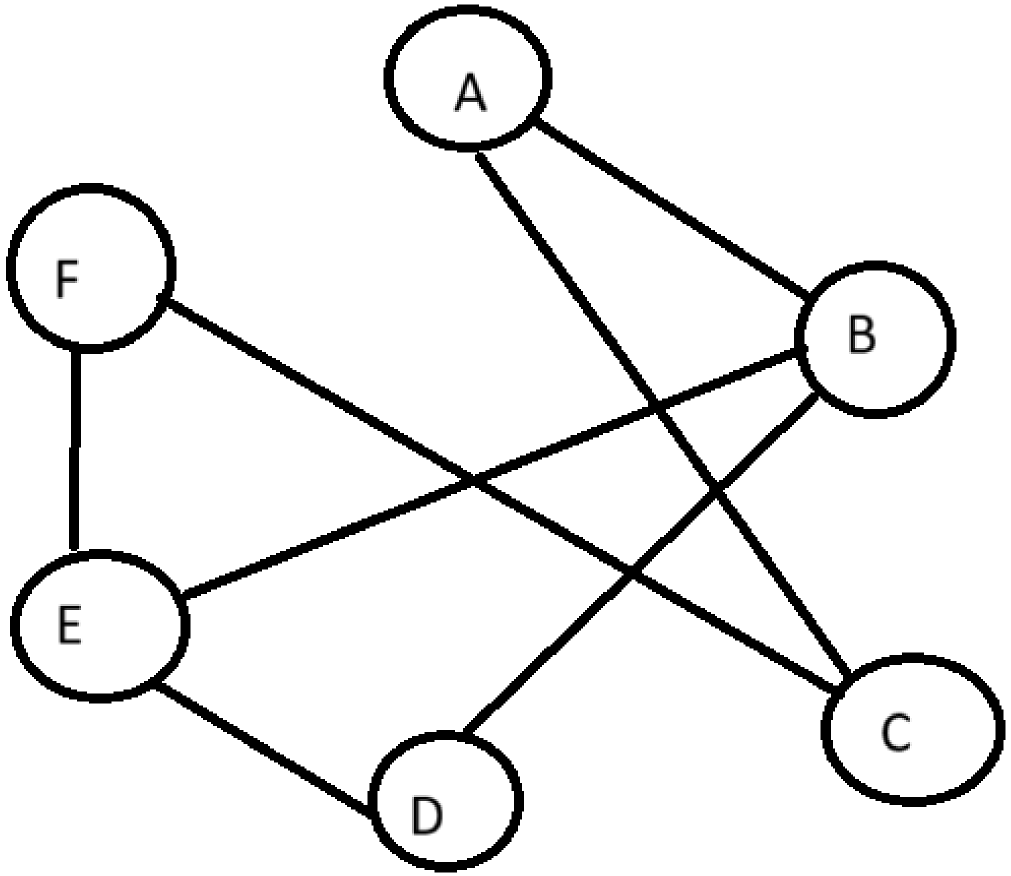


Running times

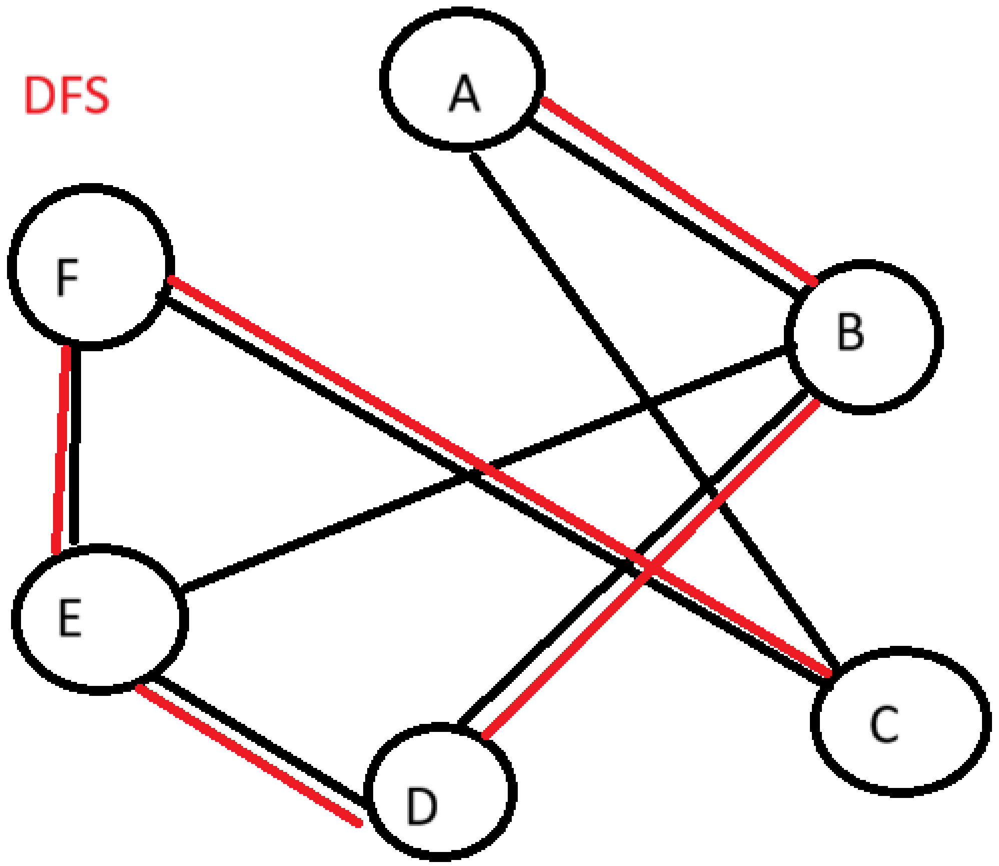
Both of BFS and DFS?

$\Theta(V + E)$

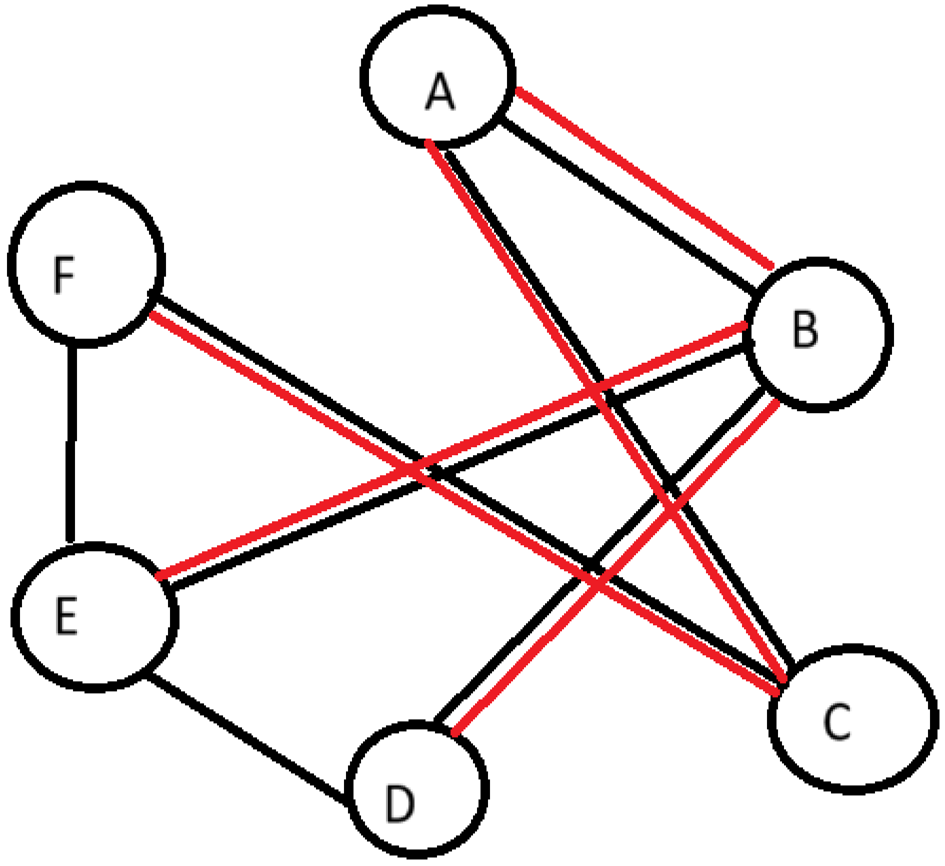
Do BFS and DFS produce the same trees?



DFS



BFS

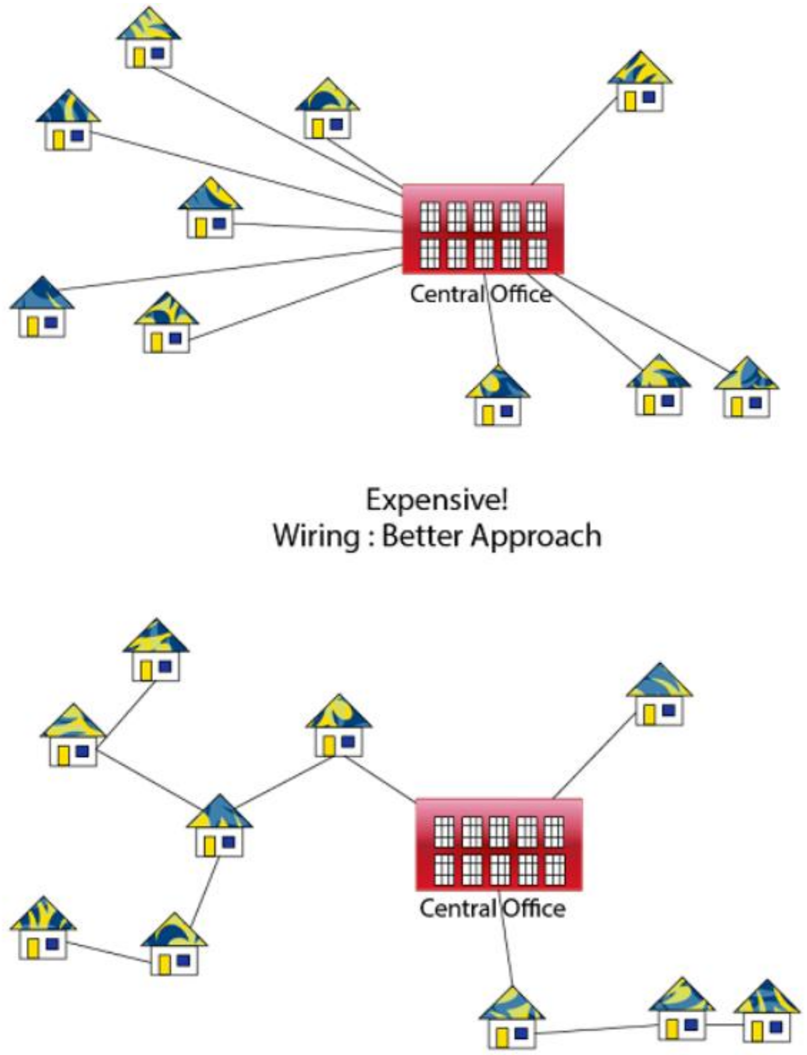


Minimum Spanning Trees Introduction (2)

To model a wiring problem, use a connected, undirected graph $G = (V, E)$, where V is the set of locations, E is the set of possible interconnections between pairs of locations, and for each edge $(u, v) \in E$, a weight $w(u, v)$ specifies the cost (amount of wire needed) to connect u and v.

The goal is to find an acyclic subset $T \subseteq E$ that connects all of the vertices and whose total weight

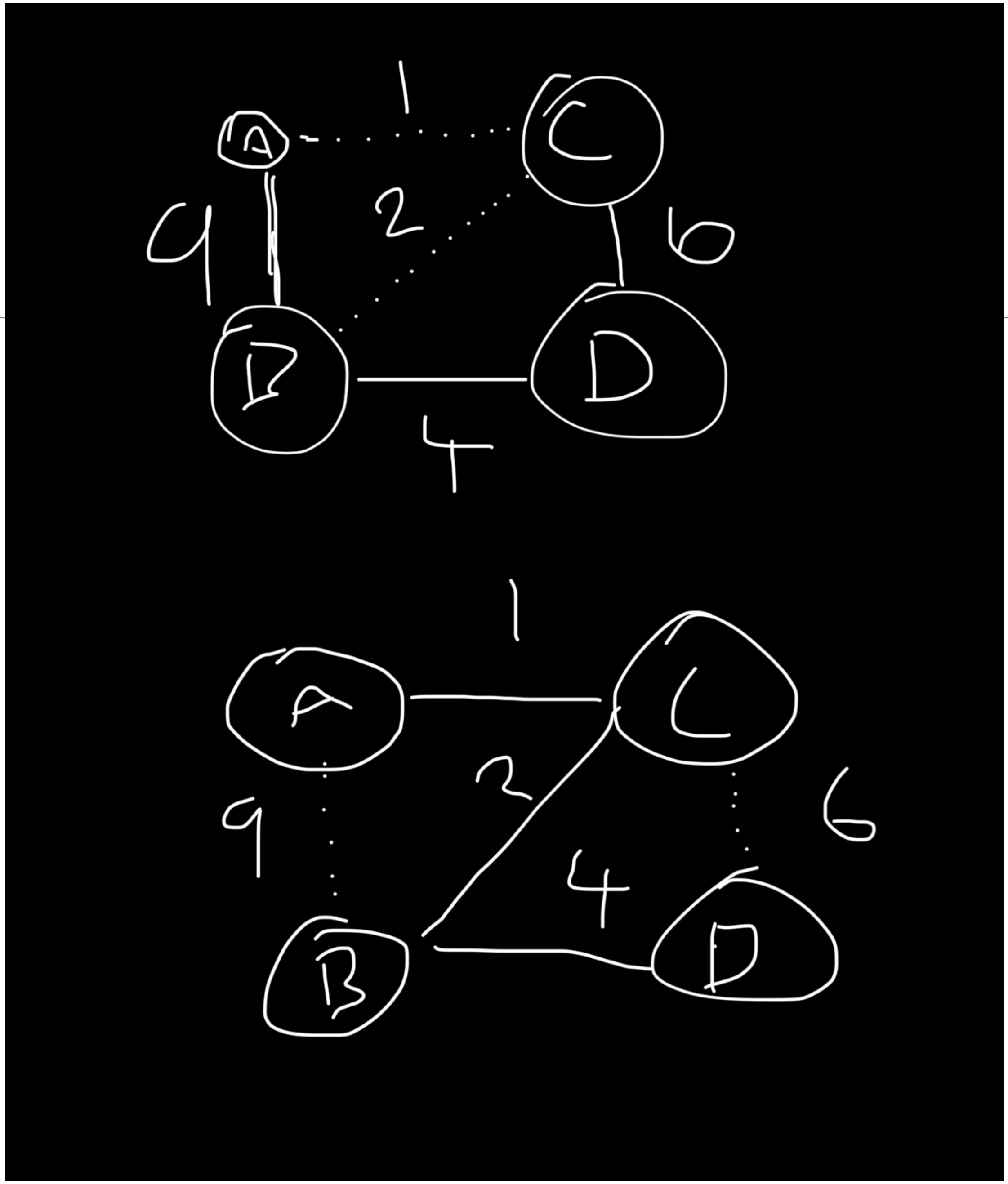
$\boldsymbol{w(T) = \sum_{(u,v)\in T} w(u,v)}$ is minimized.

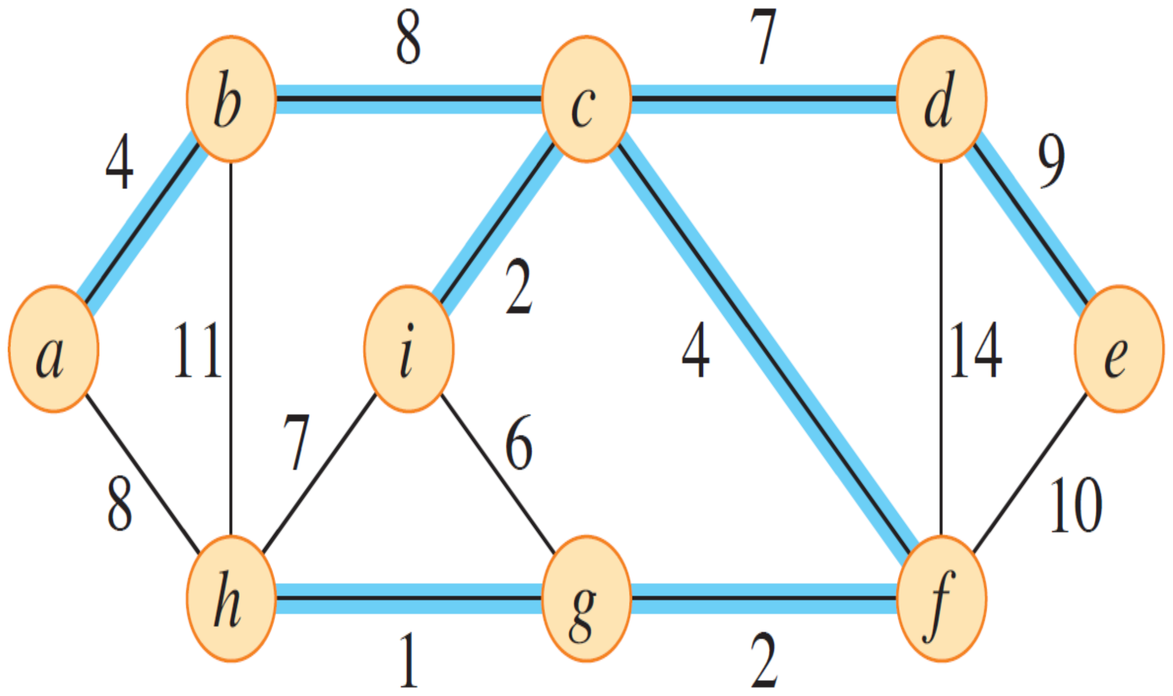


Introduction (3)

Since T is acyclic and connects all of the vertices, it must form a tree, which we call a ***spanning tree*** since it “spans” the graph G.

We call the problem of determining the tree T the ***minimum-spanning-tree problem***. Next slide shows an example of a connected graph and a minimum spanning tree.





37.

Selected algorithms (1)

As the graph gets bigger it gets more difficult to ‘see’ the minimum spanning tree

This section presents two ways to solve the minimum-spanning-tree problem.

Kruskal’s algorithm and **Prim’s** algorithm both run in approximately $O(E \lg V)$ time.

Kruskal’s Algorithm

Approach: Builds the minimum spanning tree (MST) by sorting all edges by weight and adding the smallest edge that doesn't create a cycle (using a union-find to detect cycles.)

Time Complexity: O(ElogV)

(dominated by sorting edges)

Best For: Sparse graphs (fewer edges), as it doesn't need to explore the graph's structure deeply.

Data Structure: Relies on sorting and union-find, not a priority queue.

Prim’s algorithm

Approach: Builds the MST by starting from a vertex, adding the smallest edge connecting the MST to a new vertex, and repeating until all vertices are included, using a priority queue to find the smallest edge.

Time Complexity: O((V+E)logV) with a binary heap.

Best For: Dense graphs (many edges) when using a simple array O(V^2), or sparse graphs with a good priority queue implementation.

Data Structure: Uses a priority queue (e.g., binary heap or Fibonacci heap) to manage edge weights.

Key Differences

Kruskal's works globally (sorts all edges), while Prim's works locally (grows the MST from a starting vertex).

Kruskal's is better for sparse graphs; Prim's can be tuned for both sparse and dense graphs depending on the data structure.

Kruskal's doesn't use heaps; Prim's often uses a priority queue, like a binary or Fibonacci heap.

Selected algorithms

The two algorithms are greedy algorithms. Each step of a **greedy** algorithm must make one of several possible choices.

The greedy strategy advocates making the choice that is the best at the moment. Such a strategy does not generally guarantee that it always finds globally optimal solutions to problems.

For the minimum-spanning-tree problem, however, we can prove that certain greedy strategies do yield a spanning tree with minimum weight.

Growing a minimum spanning tree

Growing a minimum spanning tree

The two algorithms considered in this section use a greedy approach to the problem, although they differ in how they apply this approach.

This greedy strategy is captured by the procedure GENERIC-MST on the next slide, which grows the minimum spanning tree one edge at a time. (only finding that edge is different)

The generic method manages a set A of edges, maintaining the following loop invariant:

Prior to each iteration, A is a subset of some minimum spanning tree.

GENERIC-MST Procedure

```
GENERIC-MST$(G, w)$
1  $A = \emptyset$                 //$A$ is a subset of some minimum spanning tree
2  while $A$ does not form a spanning tree
3      find an edge $(u, v)$ that is safe for $A$
4      $A = A \cup \{(u, v)\}$
5  return $A$                      //keep adding edges that maintains rules until a MST is built
```

GENERIC-MST Procedure – explanation

Each step determines an edge (u, v) that the procedure can add to A without violating this invariant, in the sense that $A \cup \{(u, v)\}$ is also a subset of a minimum spanning tree.

We call such an edge a ***safe edge*** for A, since it can be added safely to A while maintaining the invariant.

GENERIC-MST Procedure – Loop Invariant (1)

This generic algorithm uses the loop invariant as follows:

Initialization: After line 1, the set A trivially satisfies the loop invariant.

Maintenance: The loop in lines 2-4 maintains the invariant by adding only safe edges.

Termination: All edges added to A belong to a minimum spanning tree, and the loop must terminate by the time it has considered all edges. Therefore, the set A returned in line 5 must be a minimum spanning tree.

GENERIC-MST Procedure – Loop Invariant (2)

The tricky part is finding a safe edge in line 3.

- The algorithm maintains a set of edges called A, which is a part of some spanning tree T
- At the start of the algorithm (line 3), A is included in a valid spanning tree T, meaning A is a piece of a tree that could eventually become the MST.
- Inside the while loop, A is not yet the full spanning, so there are still edges in T that aren't in A.
- Because A is not complete, there must be at least one edge (called (u, v)) in T that: Is not yet in A.
- Can be added to A without creating a cycle or violating the rules of a spanning

The algorithms of Kruskal and Prim

The two minimum-spanning-tree algorithms described in this section elaborate on the generic method. They each use a specific rule to determine a safe edge in line 3 of GENERIC-MST.

In Kruskal's algorithm, the set A is a forest whose vertices are all those of the given graph. The safe edge added to A is always a lowest-weight edge in the graph that connects two distinct components.

In Prim's algorithm, the set A forms a single tree. The safe edge added to A is always a lowest-weight edge connecting the tree to a vertex not in the tree.

Both algorithms assume that the input graph is connected and represented by adjacency lists.

Kruskal’s algorithm

Kruskal’s algorithm explanation

Kruskal’s algorithm finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge (u, v) with the lowest weight.

Kruskal’s algorithm qualifies as a greedy algorithm because at each step it adds to the forest an edge with the lowest possible weight.

The procedure MST-KRUSKAL on the following slide uses a disjoint-set* data structure to maintain several disjoint sets of elements.

Each set contains the vertices in one tree of the current forest. The operation FIND-SET(u) returns a representative element from the set that contains u.

To combine trees, Kruskal’s algorithm calls the UNION procedure.

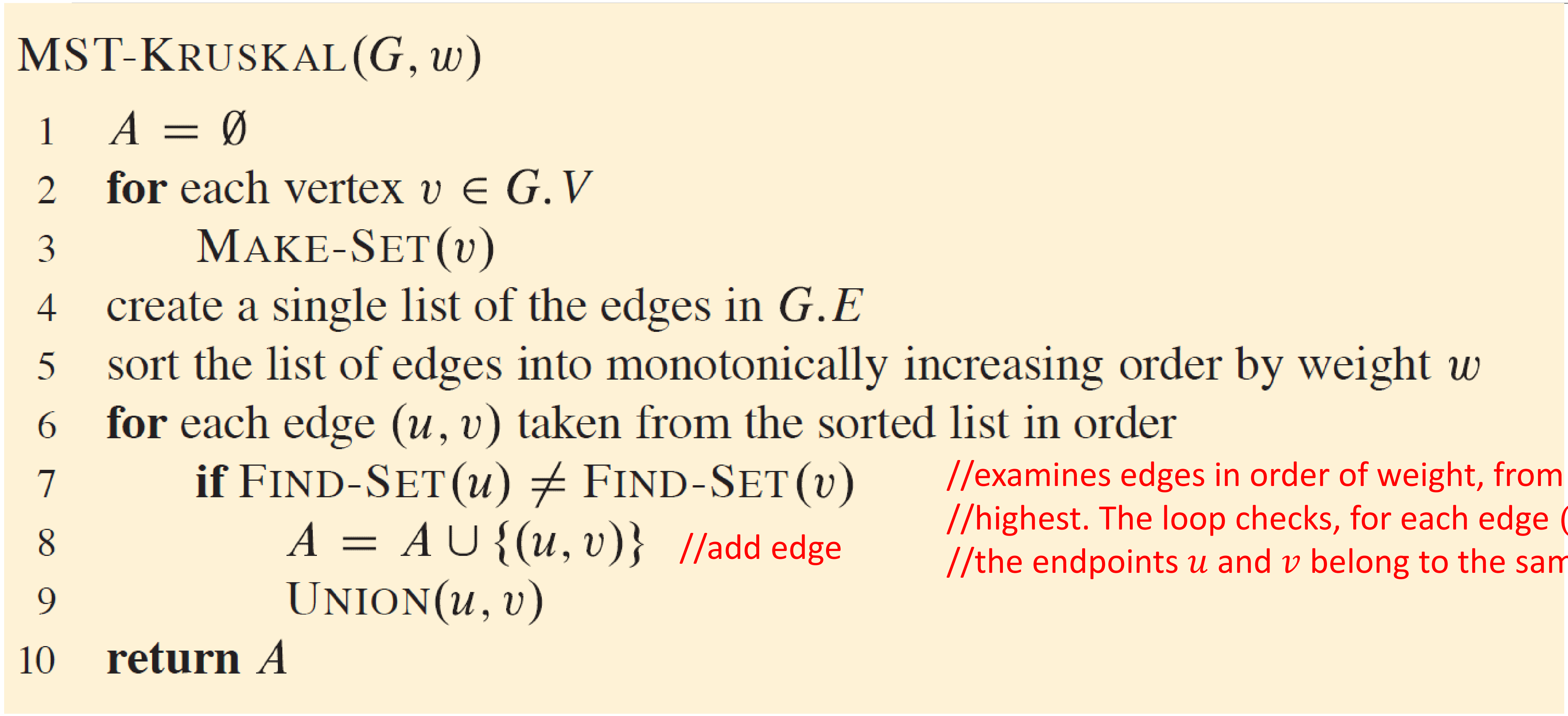
*Key operations of a Disjoint-Set data structure

MakeSet(x): Creates a new set containing only the element (x), with (x) as its representative.

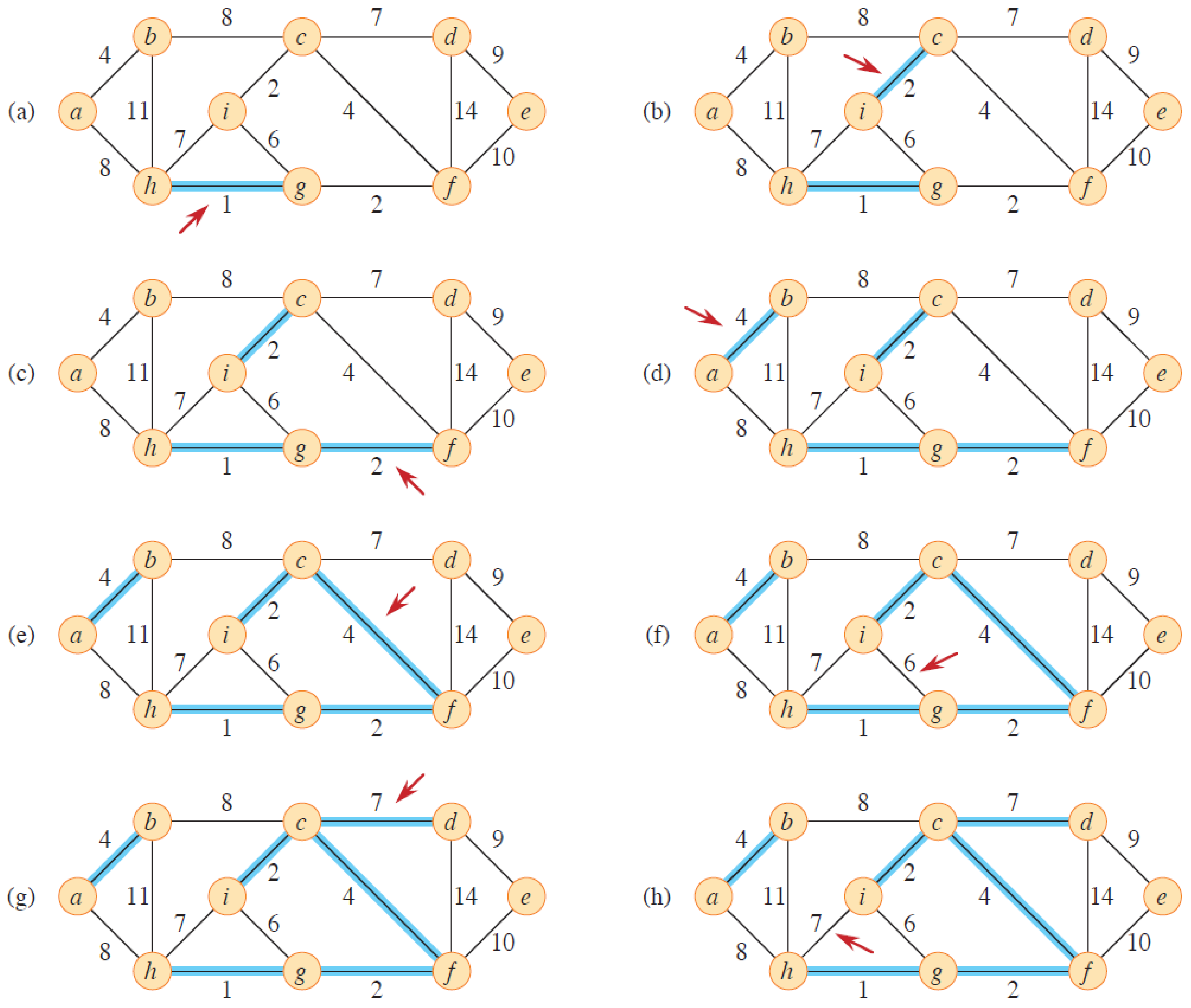
Find(x): Returns the representative (root) of the set containing (x). If two elements have the same representative, they are in the same set.

Union(x, y): Merges the sets containing (x) and (y) into a single set, choosing one representative for the combined set.

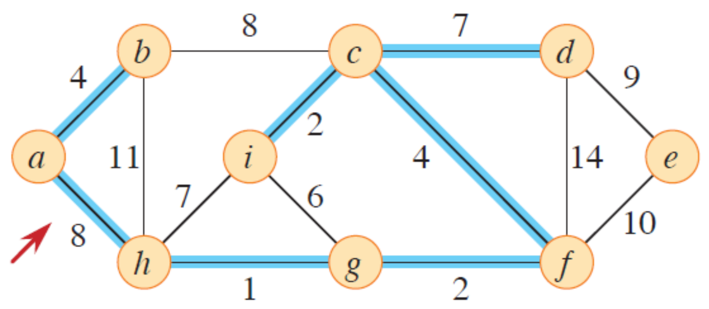
MST-KRUSKAL procedure



```
MST-KRUSKAL$(G, w)$
$A = \emptyset$
for each vertex $v \in G.V$
    MAKE-SET$(v)$
create a single list of the edges in $G.E$
sort the list of edges into monotonically increasing order by weight $w$
for each edge $(u, v)$ taken from the sorted list in order
    if FIND-SET$(u) \neq$ FIND-SET$(v)$        //examines edges in order of weight, from lowest to
        $A = A \cup \{(u, v)\}$  //add edge     //highest. The loop checks, for each edge $(u, v)$, whether
        UNION$(u, v)$                            //the endpoints $u$ and $v$ belong to the same tree.
return $A$
```

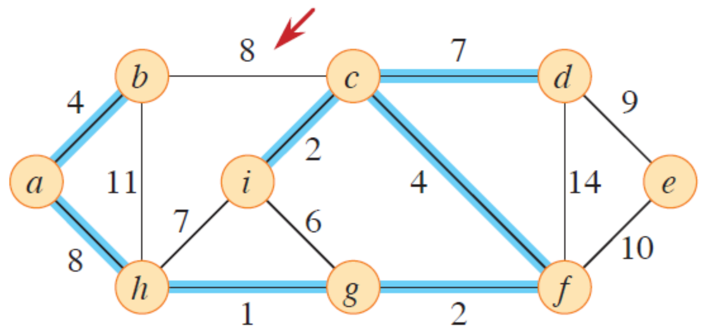

(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
14
7
6
4
10
8
1
2

(i)



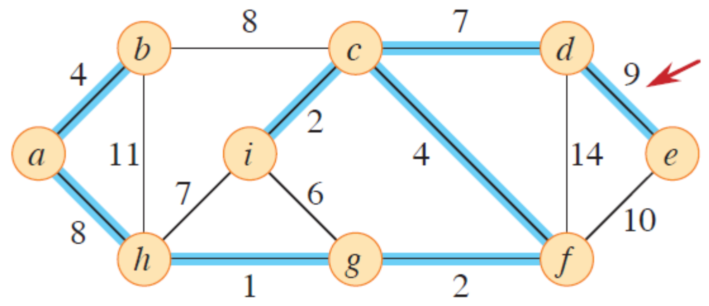
a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
10
8
1
2

(j)



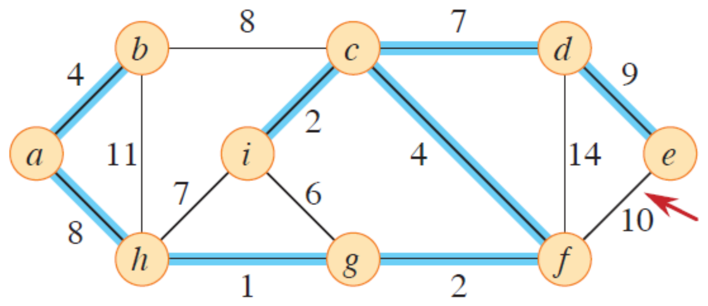
a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
10
8
1
2

(k)



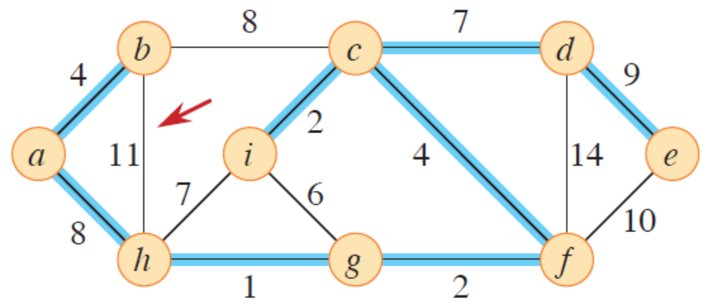
a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
10
8
1
2

(l)



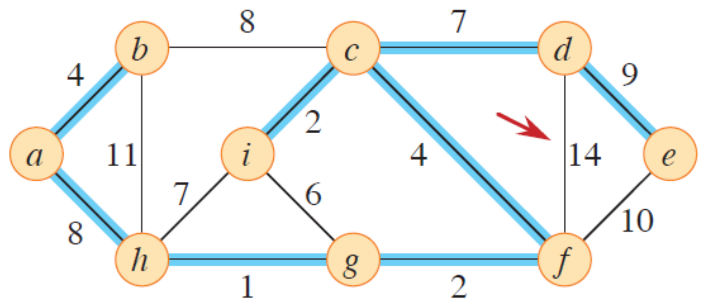
a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
10
8
1
2

(m)



a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
10
8
1
2

(n)



a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
10
8
1
2

The running time of Kruskal’s algorithm

The running time of Kruskal’s algorithm for a graph $G = (V, E)$ depends on the specific implementation. But in simple terms, the running time is O(E log E), where E is the number of edges in the graph.

- The algorithm sorts all edges by weight, which takes **O(E log E)** time.
- It processes each edge to check for cycles and merge components. With optimizations (like path compression and union by rank), each Union-Find operation is nearly constant time, so processing all edges takes **O(E)** time in practice.
- As long as the edge checking is O(E log E) or less, the sorting step dominates, so the overall running time is O(E log E).

Prim’s algorithm

Prim’s algorithm explanation

Like Kruskal’s algorithm, Prim’s algorithm is a special case of the generic minimum-spanning-tree method.

Prim’s algorithm has the property that the edges in the set A always form a single tree.

The following slide shows, the tree starts from an arbitrary root vertex r and grows until it spans all the vertices in V. Each step adds to the tree A a light edge that connects A to an isolated vertex - one on which no edge of A is incident. By the Theorem in slide 16, this rule adds only edges that are safe for A.

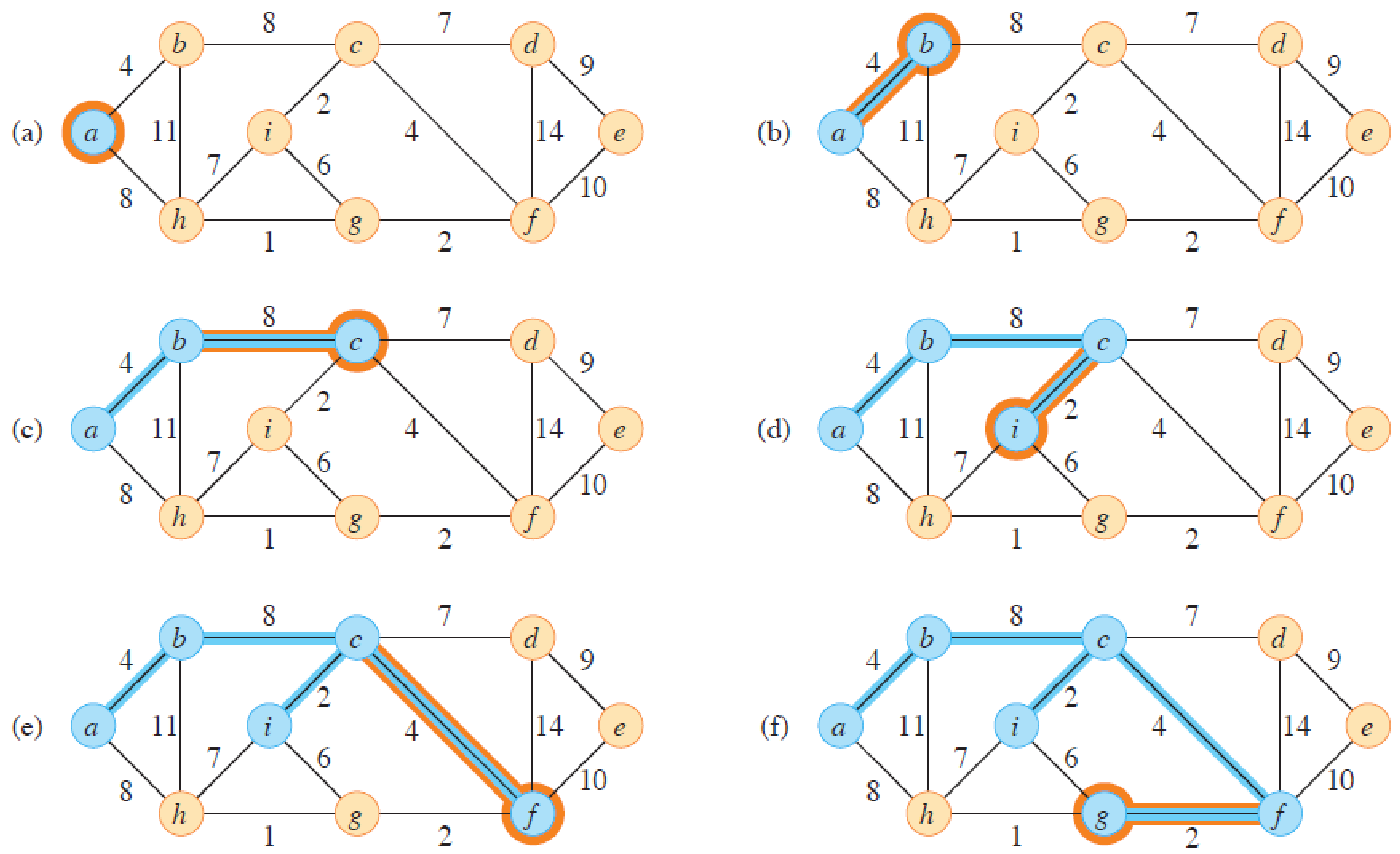
Therefore, when the algorithm terminates, the edges in A form a minimum spanning tree. This strategy qualifies as greedy since at each step it adds to the tree an edge that contributes the minimum amount possible to the tree’s weight.

Prim’s basic (hard) constraints

We want all vertices used

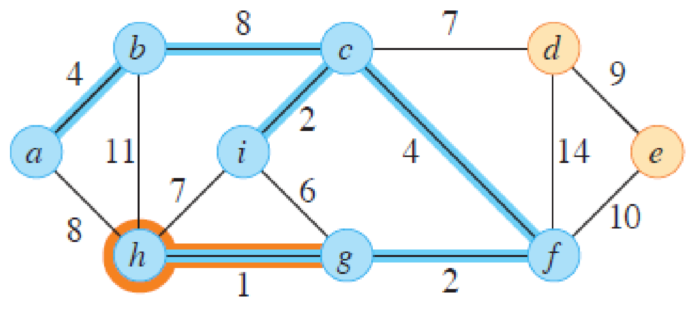
We want all vertices connected

We want no cycles



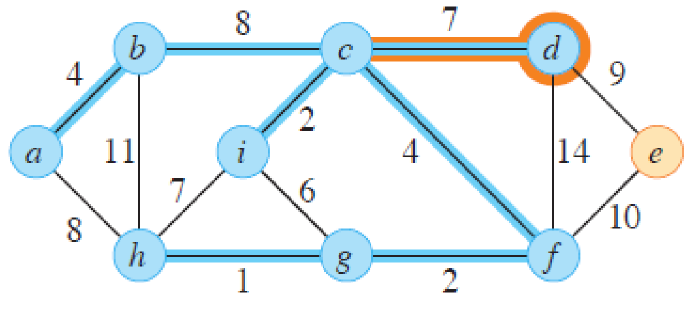
are light edges crossing the cut.

(g)



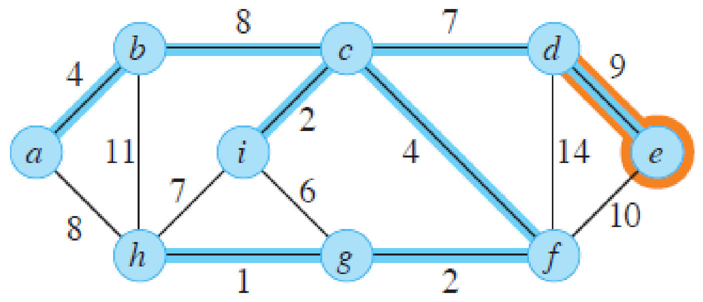
a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
10
8
1
2

(h)



a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
10
8
1
2

(i)



a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
10
8
1
2

```
MST-PRIM($G, w, r$)
1   for each vertex $u \in G.V$
2       $u.key = \infty$
3       $u.\pi = \text{NIL}$
4   $r.key = 0$
5   $Q = \emptyset$
6   for each vertex $u \in G.V$
7       INSERT($Q, u$)
8   while $Q \neq \emptyset$
9       $u$ = EXTRACT-MIN($Q$)            // add $u$ to the tree
10      for each vertex $v$ in $G.Adj[u]$   // update keys of $u$'s non-tree neighbors
11          if $v \in Q$ and $w(u, v) < v.key$
12              $v.\pi = u$
13              $v.key = w(u, v)$
14              DECREASE-KEY($Q, v, w(u, v)$)
```

/*Lines 1-7 set the key of each vertex to ∞ (except for the root r , whose key is set to 0 to make it the first vertex processed), set the parent of each vertex to NIL, and insert each vertex into the min-priority queue Q*/

MST-PRIM procedure – explanation (1)

In order to efficiently select a new edge to add into tree A, the algorithm maintains a min-priority queue Q of all vertices that are not in the tree, based on a key attribute.

For each vertex v, the attribute $v.key$ is the minimum weight of any edge connecting v to a vertex in the tree, where by convention, $v.key = \infty$ if there is no such edge.

The attribute $v.\pi$ names the parent of v in the tree.

The running time of Prim’s algorithm (1)

The running time of Prim’s algorithm depends on the specific implementation of the min-priority queue Q.

You can implement Q with a binary min-heap, including a way to map between vertices and their corresponding heap elements. The BUILD-MIN-HEAP procedure can perform lines 5-7 in $O(V)$ time.

The body of the **while** loop executes $|V|$ times, and since each EXTRACT-MIN operation takes $O(\lg V)$ time, the total time for all calls to EXTRACT-MIN is $O(V \lg V)$.

The running time of Prim's algorithm (2)

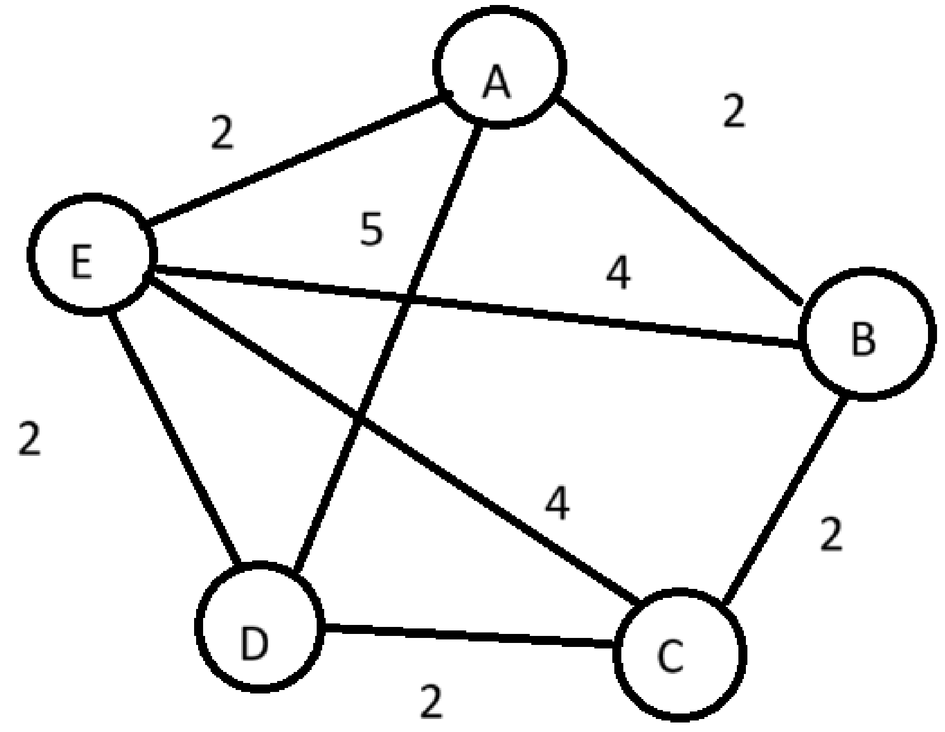
The **for** loop in lines 10-14 executes $O(E)$ times altogether, since the sum of the lengths of all adjacency lists is $2|E|$.

Within the **for** loop, the test for membership in Q in line 11 can take constant time if you keep a bit for each vertex that indicates whether it belongs to Q and update the bit when the vertex is removed from Q.

Each call to DECREASE-KEY in line 14 takes $O(\lg V)$ time.

Thus, the total time for Prim's algorithm is $O(V \lg V + E \lg V) = O(E \lg V)$, which is asymptotically the same as for our implementation of Kruskal's algorithm.

Do Prim’s and Kruskel’s Algorithm Produce different MSTs?



Shortest Path

Shortest Path

Prim’s Algorithm

- Starting from an arbitrary vertex and greedily adding the cheapest edge that connects a new vertex to the growing tree until all vertices are included. **Does not produce shortest paths**

Kruskal’s Algorithm

- Sorts all edges by weight and adding them one by one to the tree, ensuring no cycles are formed, until all vertices are connected. **Does not produce shortest paths**

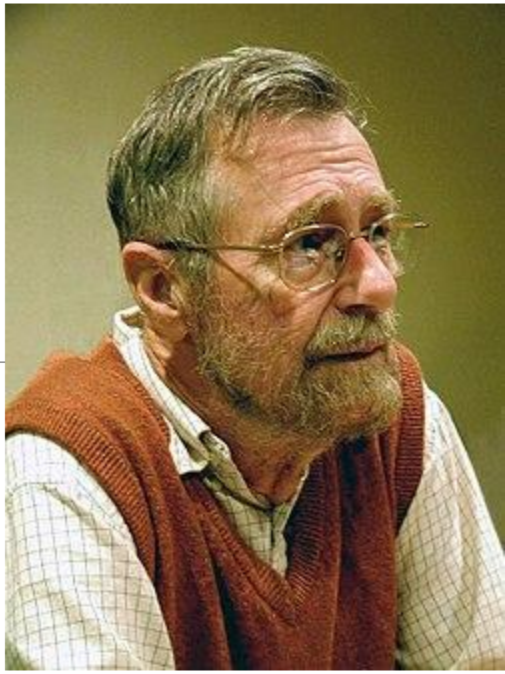
Dijkstra's algorithm

- Computes the shortest path from a single source vertex to all other vertices in a weighted graph with non-negative edge weights by iteratively selecting the vertex with the smallest tentative distance and updating distances to its neighbors.

Dijkstra's Algorithm (pronounced dike struh)

Finding the Shortest Path in Weighted Graphs

Dijkstra’s Algorithm



Named after Edsger Dijkstra, 1956

“Dijkstra’s algorithm finds the shortest path from a single source node to all other nodes in a weighted graph.”

Works on directed or undirected graphs.

Requires non-negative edge weights.

He wanted to demonstrate the capabilities a new computer. He choose a problem and a computer solution that non-computing people could understand. He designed the shortest path algorithm and later implemented it for a simplified transportation map of 64 cities in the Netherlands

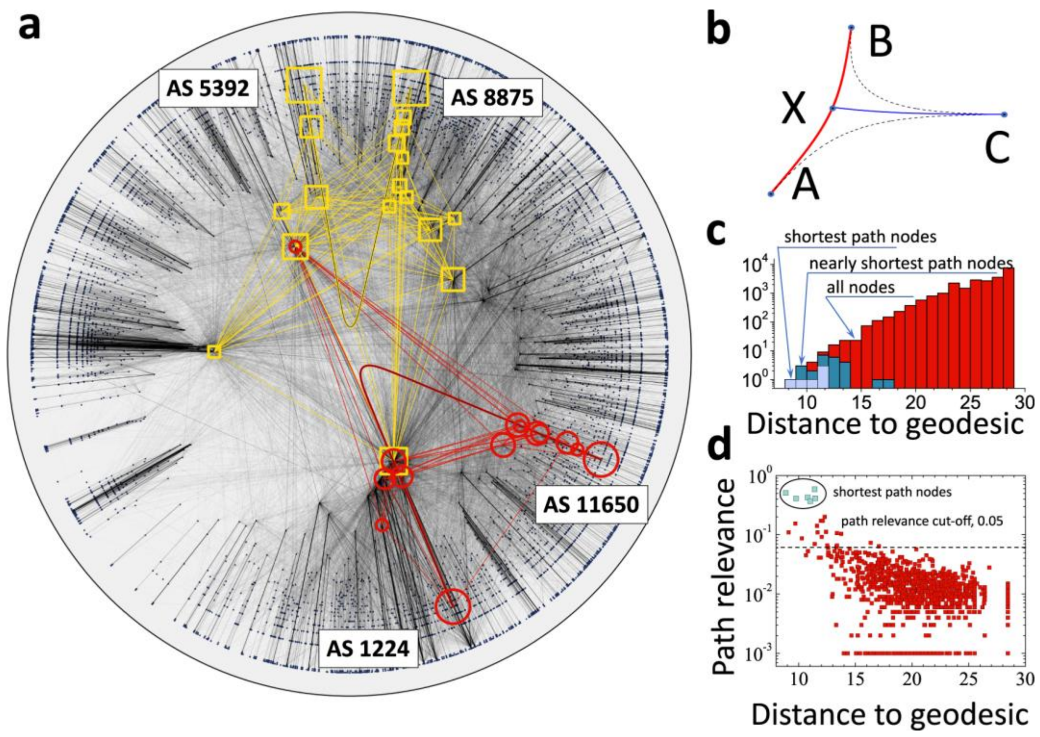
Introduction (1)

Many systems need to find the shortest path between 2 points (e.g. networks) or need precise measurements between points (e.g. electronic circuits)

For several reasons, the shortest path is usually desirable.

For networks, lower latency, less corruption

For physical wiring less cost, lower resistance….



https://www.nature.com/articles/s41467-022-35181-w

Modern applications

GPS navigation (shortest route).

Network routing (e.g., internet data packets).

Logistics (delivery route optimization).

Dijkstra’s Algorithm – Steps

Initialize:

- Set the source vertex’s distance to 0 and all others to infinity.
- Add all vertices to the priority queue with their distances.
- Mark all vertices as unvisited.

Main Loop

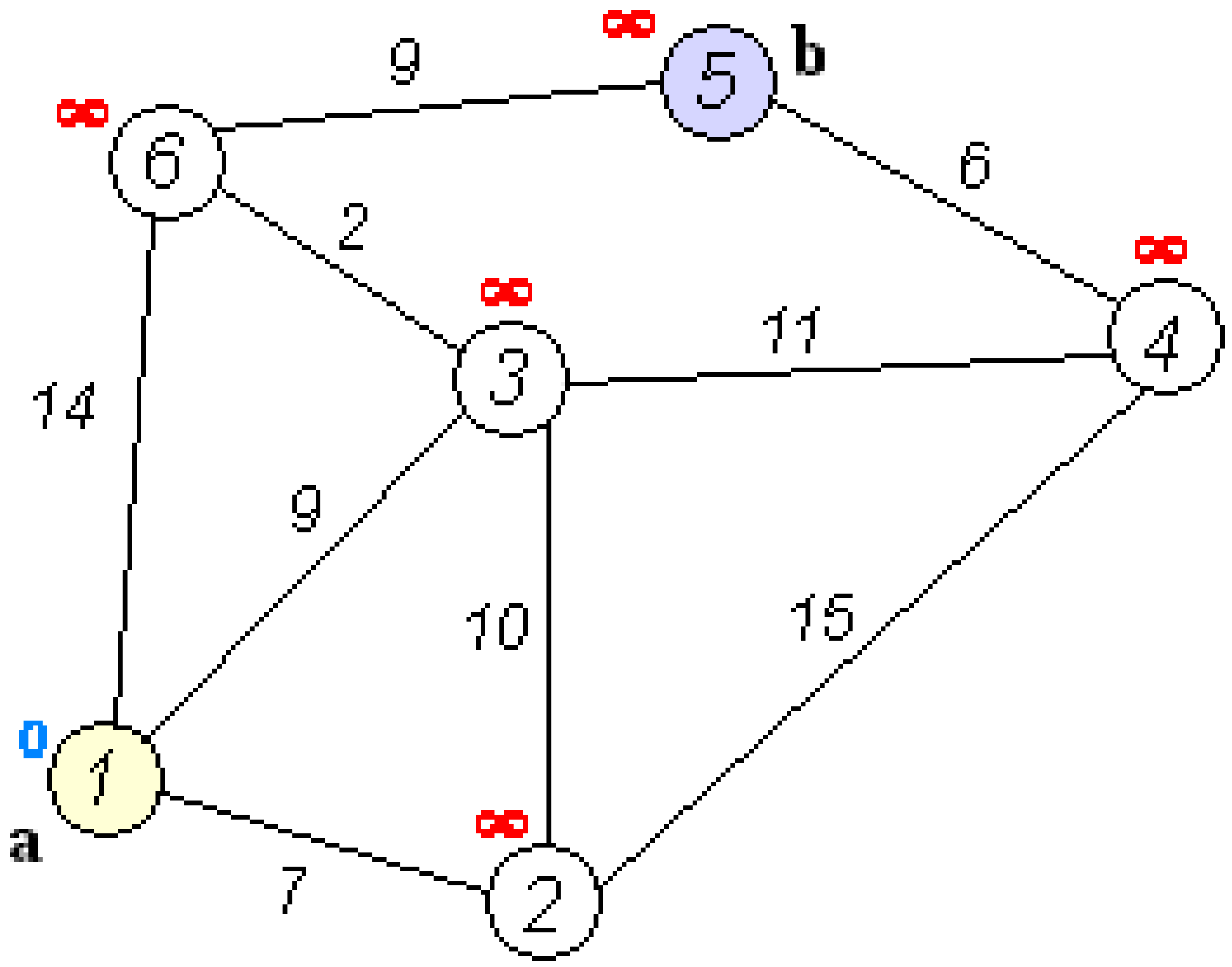
- While the priority queue is not empty:
    - Extract the vertex (u) with the minimum distance from the priority queue.
    - If (u) is already visited, skip it.
    - Mark (u) as visited.
    - For each neighbor (v) of (u):
        - Compute the potential new distance to (v) as distance[u]+weight(u,v)
    - .

If this distance is less than the current distance[v] then

- Update distance[v] with the new distance.
- Update the priority queue with the new distance for (v).
- Update the predecessor of (v) to (u).

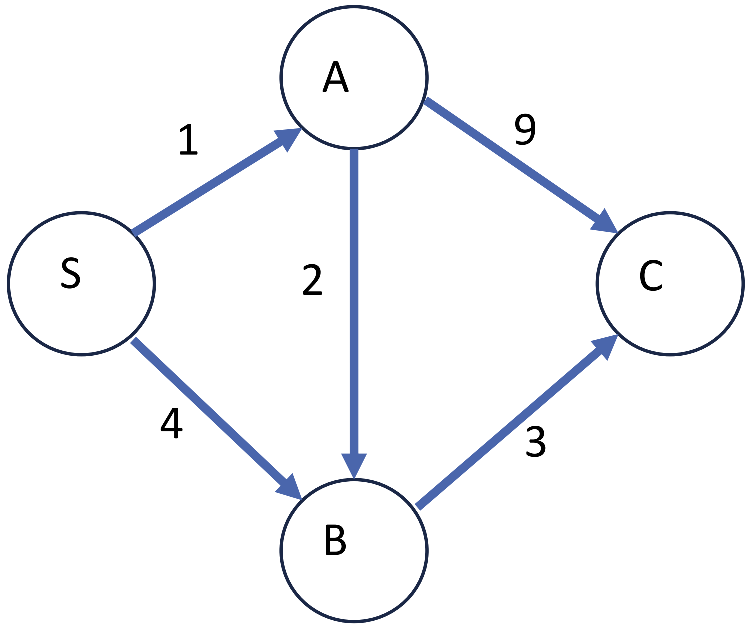
Output:

- The distance array contains the shortest path distances from the source to all vertices.
- The predecessor array (if used) allows reconstruction of the shortest paths.



∞
5
b
∞
6
9
2
6
∞
∞
3
11
4
14
9
10
15
0
1
a
∞
7
2

Grow Vertex(D,P) where D=minimum distance from source to vertex, only using explored vertices, P= Parent



Explored	Unexplored
	S(0, Nil), A(inf),B(inf),C(inf)
S(0)	A(1,S),B(4,S),C(inf)
S(0), A(1,S)	B(3,A), C(7,B)
S(0),A(1,S), B(3,A)	C(6,B)
S(0),A(1,S), B(3,A), C(6,B)	

Run Time

O((V + E) log V) with a binary heap, where V is the number of vertices and E is the number of edges.

O(V^2) with a simple array-based priority queue (less efficient for sparse graphs).

Time Complexity

Needs to maintain:

- A priority queue to select the vertex with the smallest tentative distance.
- A distance array to track the shortest known distance to each vertex.
- A visited set to avoid reprocessing vertices.

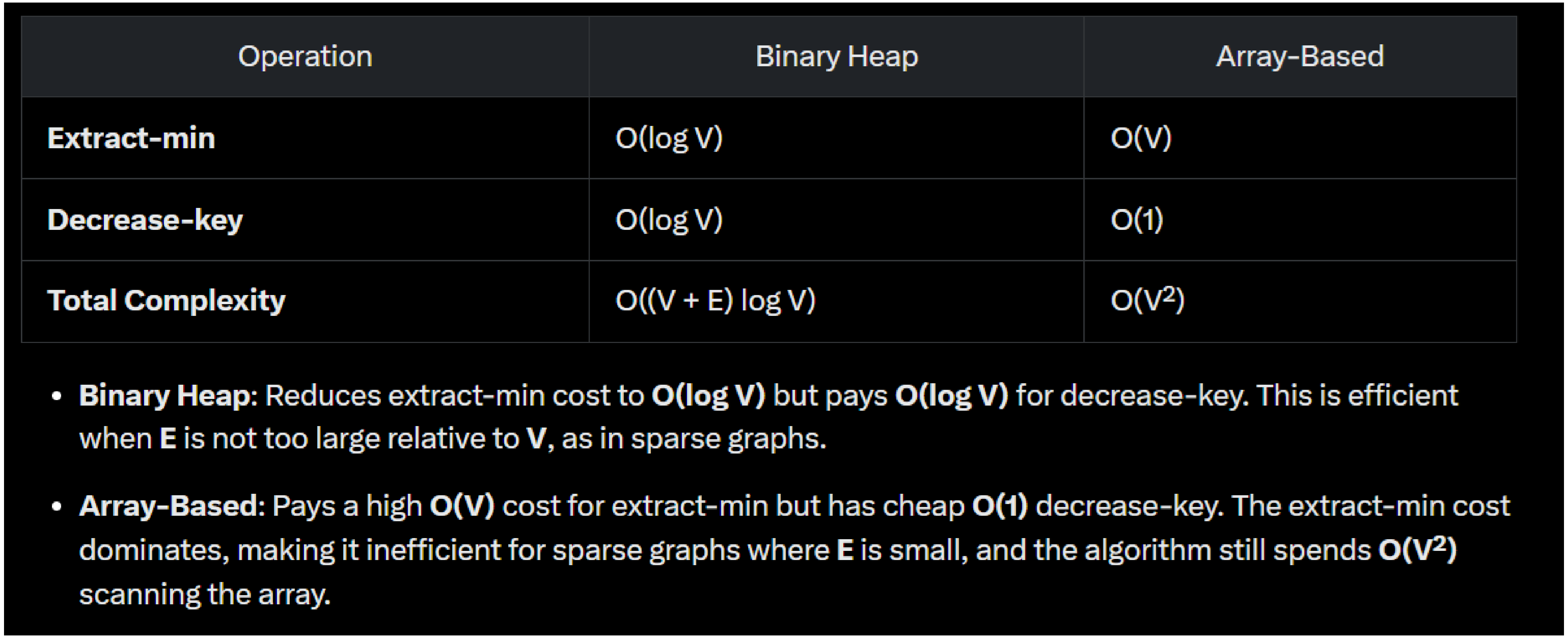
The key operations affecting time complexity are:

- Extracts the vertex with the minimum distance from the priority queue.
- Relaxes all edges from that vertex, updating distances to its neighbors if a shorter path is found.

The key operations affecting time complexity are:

- Extract-min: Removing the vertex with the smallest distance from the priority queue.
- Decrease-key: Updating the priority queue when a shorter path to a vertex is found.

Heap vs 2d array



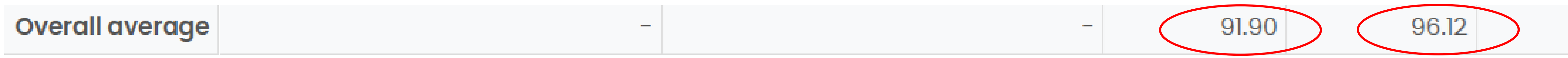
Operation	Binary Heap	Array-Based
Extract-min	$O(\log V)$	$O(V)$
Decrease-key	$O(\log V)$	$O(1)$
Total Complexity	$O((V + E) \log V)$	$O(V^2)$

- **Binary Heap:** Reduces extract-min cost to **$O(\log V)$** but pays **$O(\log V)$** for decrease-key. This is efficient when **E** is not too large relative to **V**, as in sparse graphs.
- **Array-Based:** Pays a high **$O(V)$** cost for extract-min but has cheap **$O(1)$** decrease-key. The extract-min cost dominates, making it inefficient for sparse graphs where **E** is small, and the algorithm still spends **$O(V^2)$** scanning the array.

Labs – Continual assessment

“the labs are very hard, could you give us more feedback on them”

A. Are they hard?



			CA1	CA2
Overall average	-	-	91.90	96.12

So…..either, they are hard but you are all geniuses (possible) or…
..they arnt too hard.

I think the latter.
Mainly because: You receive TA support, templates, time to finish offline (open book time)

Labs - Continual Assessment

B. More feedback and support

Lecture on Sunday 27th will be entirely a review, including review of Coursework answers so far, sample exam question and any other questions (email me them so I can prepare answers and appear clever). For example, how would I answer this question?

Rank the following functions by order of growth.

		$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$(3/2)^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	
$\ln \ln n$		$n \cdot 2^n$	$n^{\lg \lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
	$2^{\sqrt{2 \lg n}}$	n	2^n	$n \lg n$	$2^{2^{n+1}}$

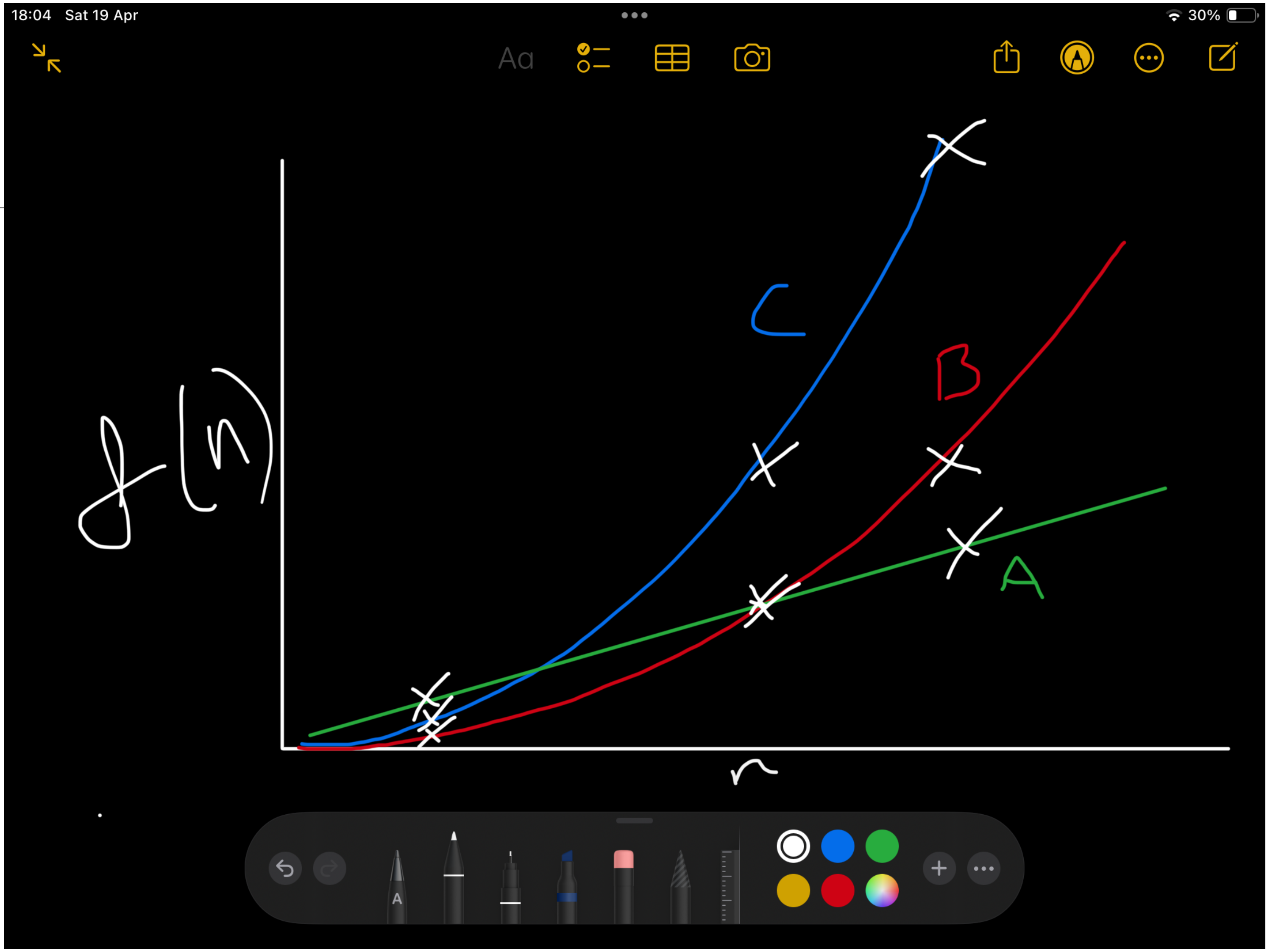
Top Tip

Approximate!!!

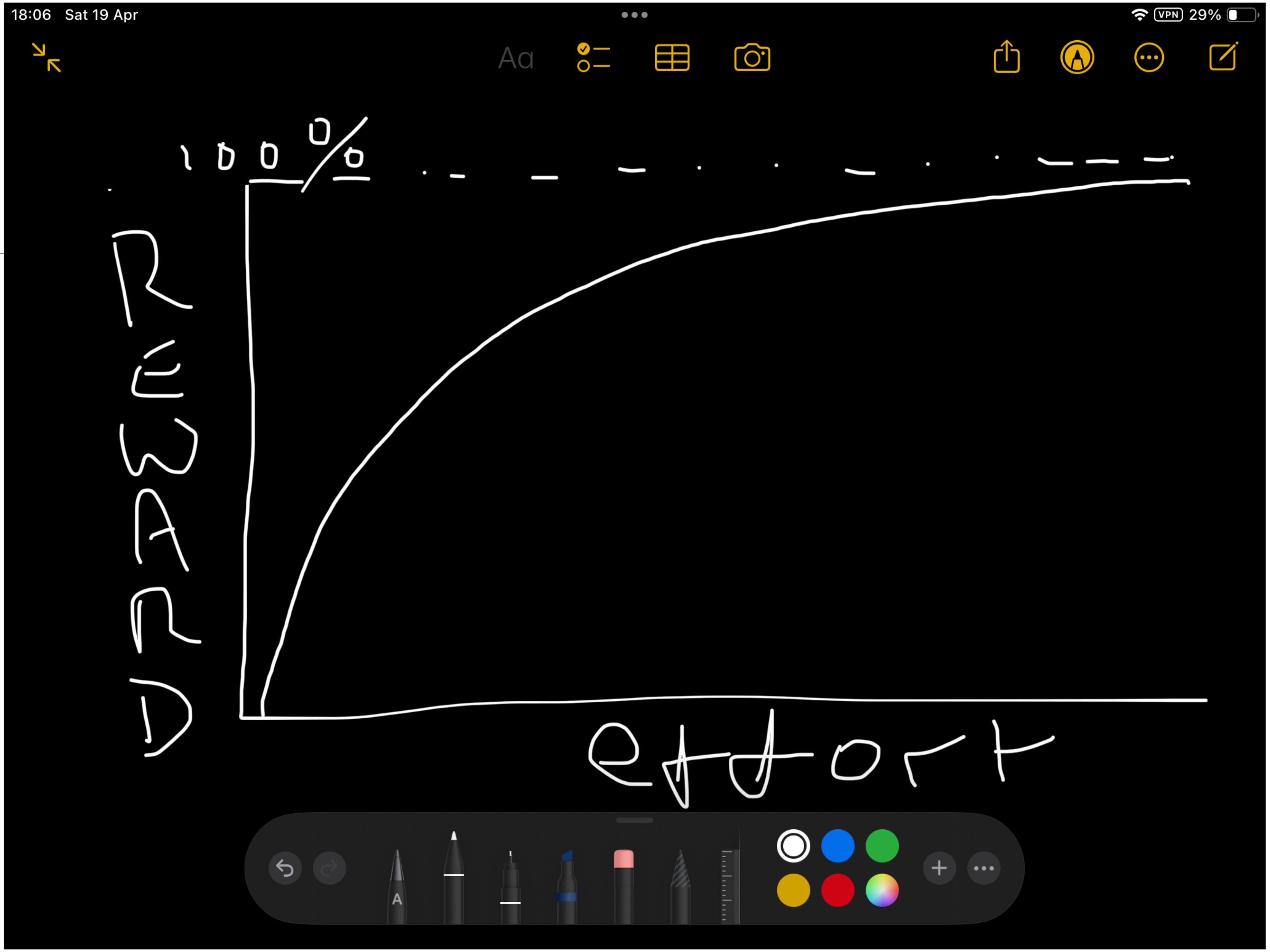
Unless a question specifically asks for exact quantities (and give no marks for being close)

……. approximate

This graph is an approximation



Question design


18:06 Sat 19 Apr
REWARD
effort