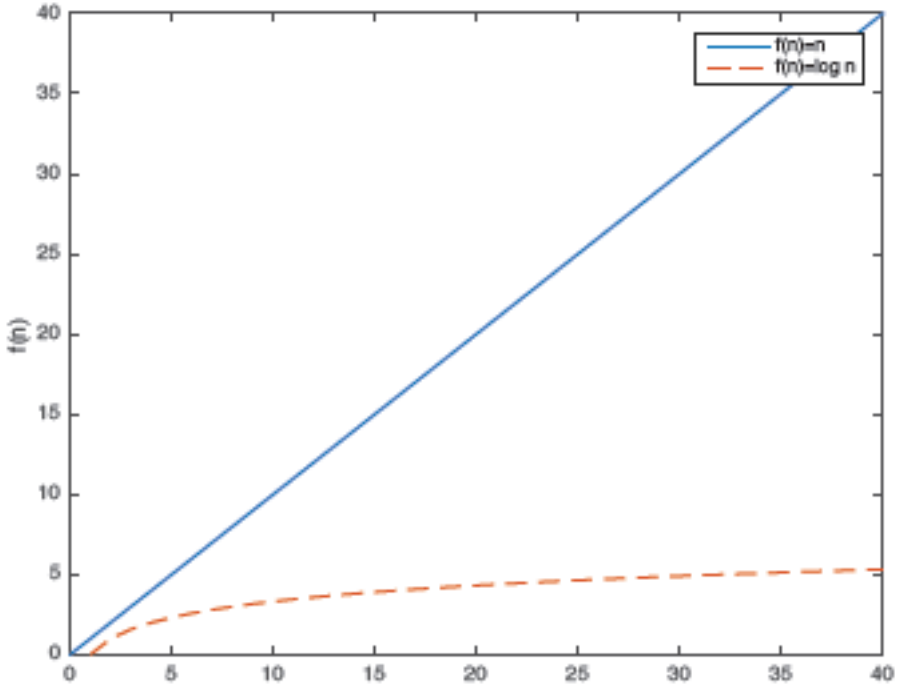
Graph Algorithm

Graphs

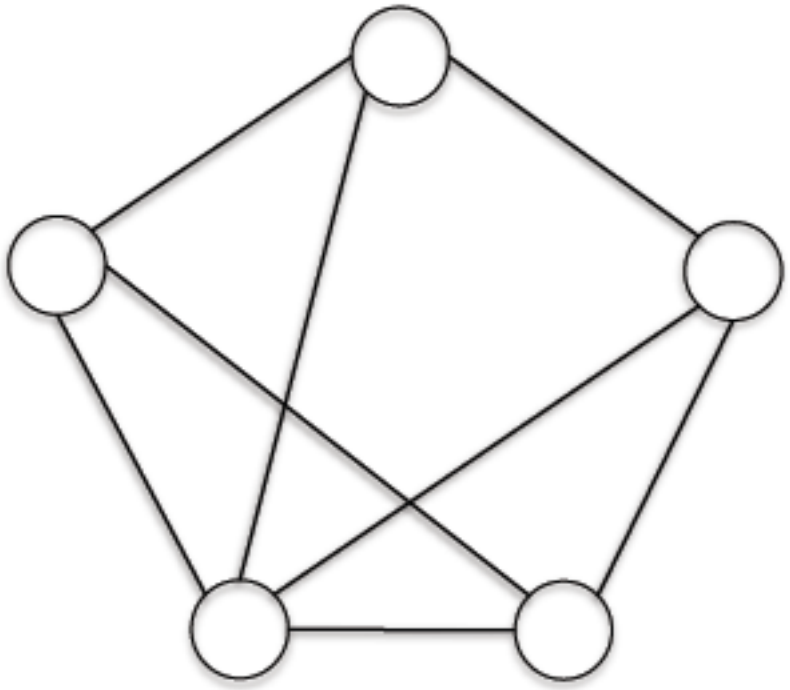
- When you hear the word "graph," you probably think about an x-axis, a y-axis, and so on.



(a) A graph (to most of the world)

Graphs

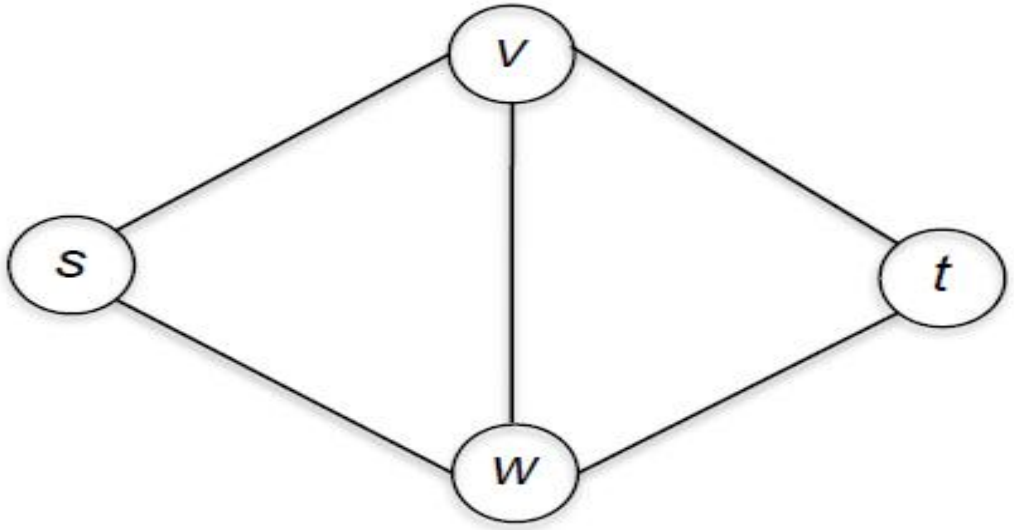
- In algorithms, a graph is a representation of a s of objects (such as people) and the pairwi relationships between them (such as friendships).
- The vertices (singular: vertex) or the nodes of t graph.
- The pairwise relationships translate to the edges the graph.
- We usually denote the vertex and edge sets of graph by and E, respectively,
- G = (V,E) to mean the graph G with vertices V a edges E.



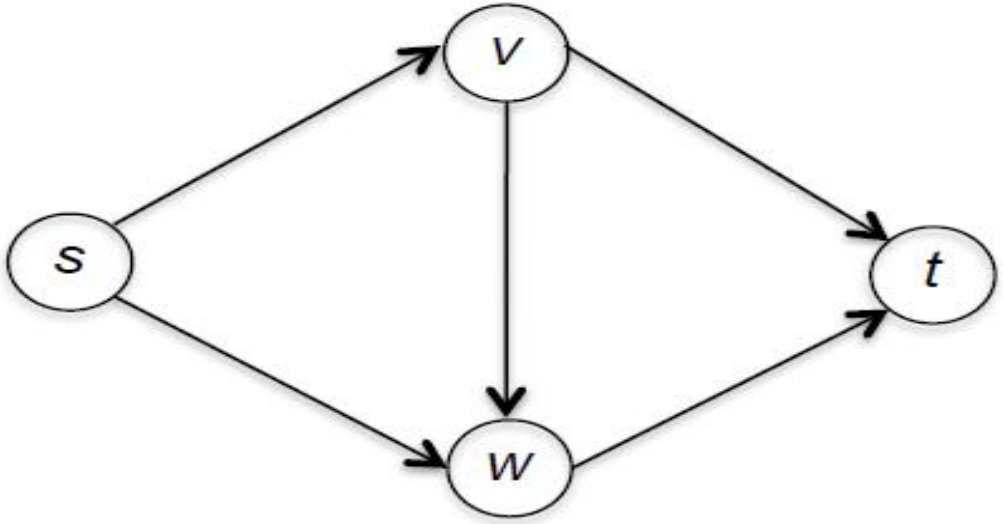
(b) A graph (in algorithms)

Directed and Undirected Graphs

- In an undirected graph, each edge corresponds to an unordered pair {v,w} of vertices there is no difference between an edge (v,w) and an edge (w, v).
- In a directed graph, each edge (v,w) is an ordered pair, with the edge traveling from the first vertex v (called the tail) to the second w (the head)



(a) An undirected graph



(b) A directed graph

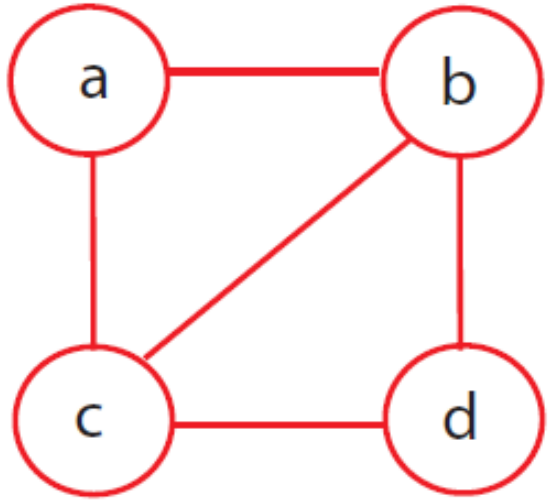
Directed and Undirected Graphs

- Graph G = (V, E)
- V = set of vertices
- E = set of edges i.e. vertex pairs (v, w)
  - ordered pair ⇒ directed edge of graph (v, w)
  - unordered pair ⇒ undirected {v, w}

Directed vs Undirected Graph

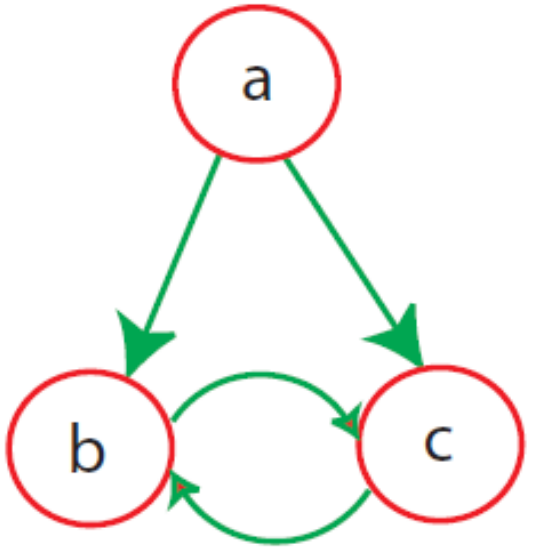
Directed Graph

- V = {a,b,c,d}
- E = {{a,b},{a,c},{b,c},{b,d},{c,d}}



Undirected Graph

- V = {a,b,c}
- E = {(a,c),(b,c),(c,b),(b,a)}



https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/MIT6_006F11_lec13.pdf

A Few Applications

- Road networks. When your smartphone's software computes driving directions, it searches through a graph that represents the road network, with vertices corresponding to intersections and edges corresponding to individual road segments.
- The World Wide Web. The Web can be modeled as a directed graph, with the vertices corresponding to individual Web pages, and the edges corresponding to hyperlinks, directed from the page containing the hyperlink to the destination page.
- Social networks. A social network can be represented as a graph whose vertices correspond to individuals and edges to some type of relationship. For example, an edge could indicate a friendship between its endpoints, or that one of its endpoints is a follower of the other.

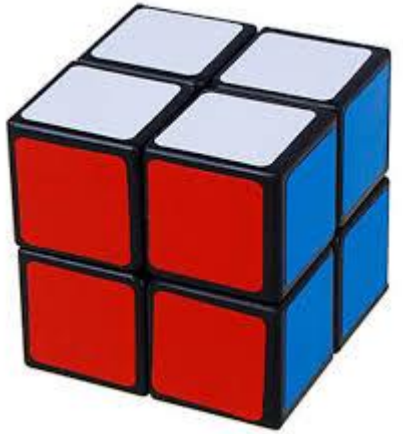
A Few Applications

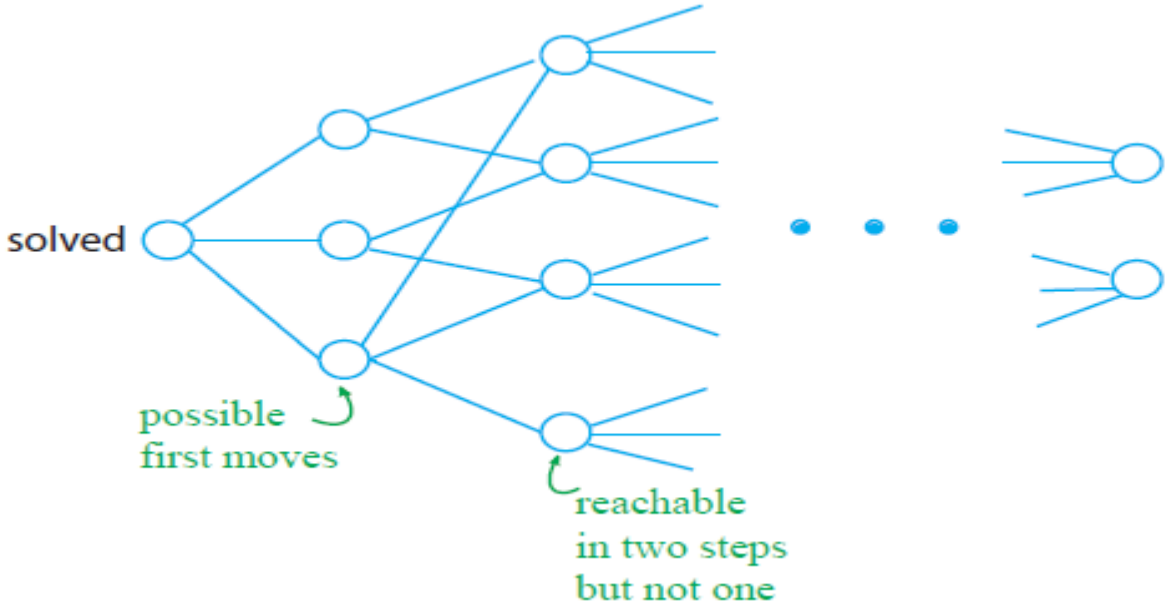
- The Internet is a graph [vertices = end hosts + routers, directed edges = direct physical or wireless connections].
- web crawling (how Google finds pages)
- social networking (Facebook friend Finder)
- network broadcast routing
- garbage collection
- model checking (Finite state machine)
- solving puzzles and games
- Precedence constraints. Graphs are also useful in problems that lack an obvious network structure. For example, imagine that you have to complete a bunch of tasks, subject to precedence constraints— perhaps you're a first-year university student, planning which courses to take and in which order.

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/MIT6_006F11_lec13.pdf

Rubik's cube

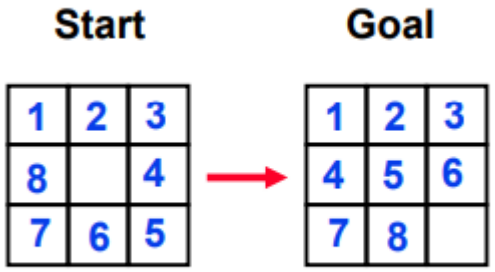
- Consider a 2*2*2 Rubik's cube
- Configuration Graph:
  - vertex for each possible state
  - edge for each basic move

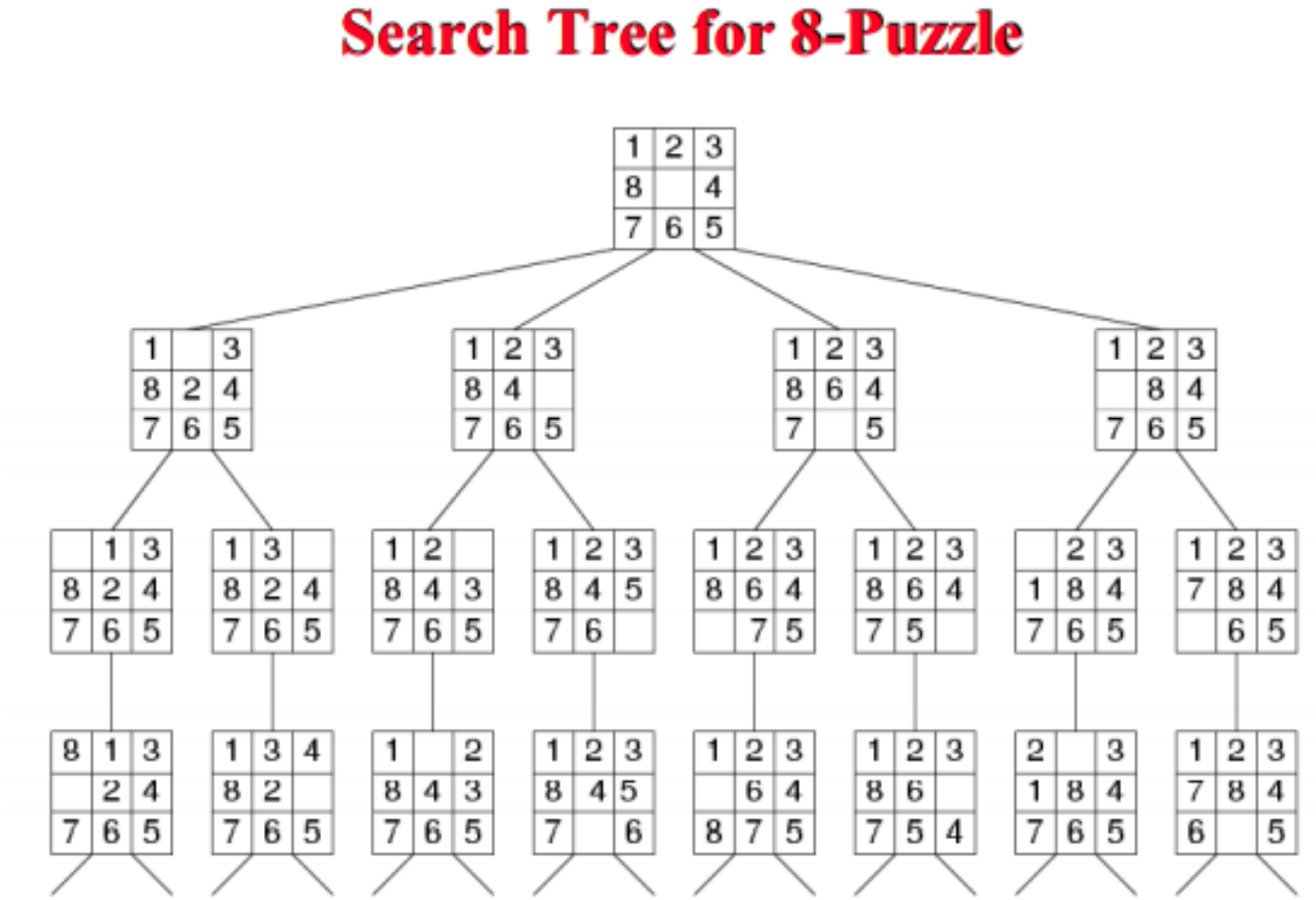




- BFS
  - 2*2*2
  - 7*7*7 →more than life time of the universe

8-Puzzle





[Search (washington.edu)]

Measuring the Size of a Graph

- For a graph G = (V,E) with vertex set V and edge set E:
- n = |V | denotes the number of vertices.
- m = |E| denotes the number of edges.

Quiz

- Consider an undirected graph with n vertices and no parallel edges. Assume that the graph is connected, meaning "in one piece." What are the minimum and maximum numbers of edges, respectively, that the graph could have?

- If the graph is not be connected➔ It may has zero edges.
- If parallel edges are allowed, a graph with at least two vertices can have an arbitrarily large number of edges.

Minimum ➔n – 1 ➔$m = \Omega(n)$.
Maximum➔n(n–1)/2 ➔$m = \Theta(n^2)$

Sparse And Dense Graphs

- Informally, a graph is sparse if the number of edges is relatively close to linear in the number of vertices, and dense if this number is closer to quadratic in the number of vertices.
- For example, graphs with n vertices and O(n log n) edges are usually considered sparse, while those with $\Omega(n2/\log n)$ edges are considered dense. “Partially dense” graphs, like those with $\approx n^{3/2}$ edges, may be considered either sparse or dense, depending on the specific application.

Representing a Graph

1) Adjacency Lists
    1. An array containing the graph's vertices.
    2. An array containing the graph's edges.
    3. For each edge, a pointer to each of its two endpoints.
    4. For each vertex, a pointer to each of the incident edges

Hash tables

- A hash table supports the following operations:
  - Insert(k): Insert key k into the hash table.
  - Lookup(k): Check if key k is present in the table.
  - Delete(k): Delete the key k from the table.
- Each operation will take constant time (in expectation).

Time complexity

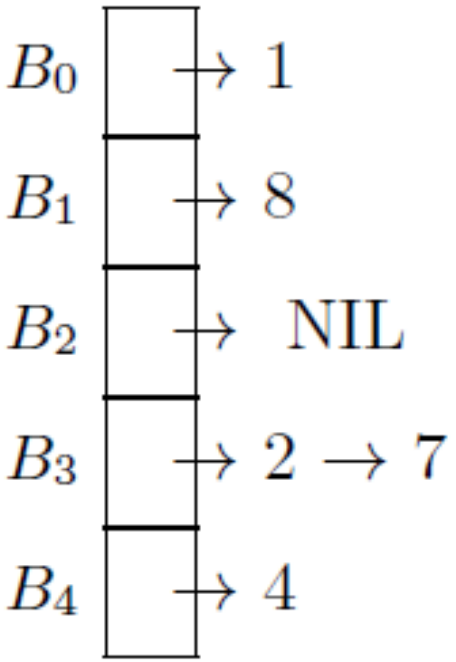
- Insert, Lookup or Delete operation on key k is linear in the length of the linked list for the bucket that key k maps to.
- an Insert could be performed in constant time by always inserting at the head of the list.

Hash tables

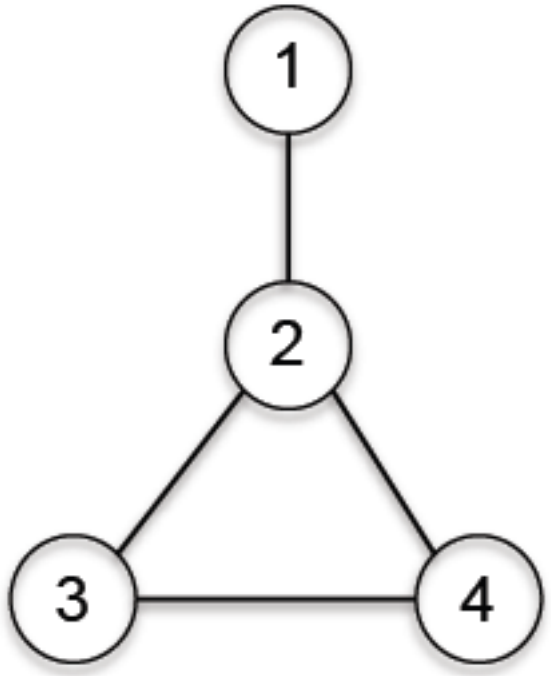
- A hash table is a commonly used data structure to store an unordered set of items, allowing constant time inserts, lookups and deletes (not always).
- Every item consists of a unique identifier called a key and values.
- The key might be a Social Security Number, a driver's license number, or an employee ID number.
- Python’s dict, Java’s HashSet/HashMap, C++’s unordered_map.

Example

- Suppose we have a hash table of size n = 5 with hash function h(x) = 13x + 2 mod 5. After inserting the elements {1; 2; 4; 7; 8}



Representing a Graph-The Adjacency Matrix



(a) A graph...

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

(b) ...and its adjacency matrix

Representing a Graph

2) an undirected graph G = (V,E) with n vertices and no parallel edges, and label its vertices 1, 2, 3, . . . ,n. The adjacency matrix representation of G is a square n×n matrix

a two-dimensional array—with only zeroes and ones as entries. Each entry Aij is defined as

$$A_{ij} = \begin{cases} 1 & \text{if edge } (i,j) \text{ belongs to } E \\ 0 & \text{otherwise.} \end{cases}$$

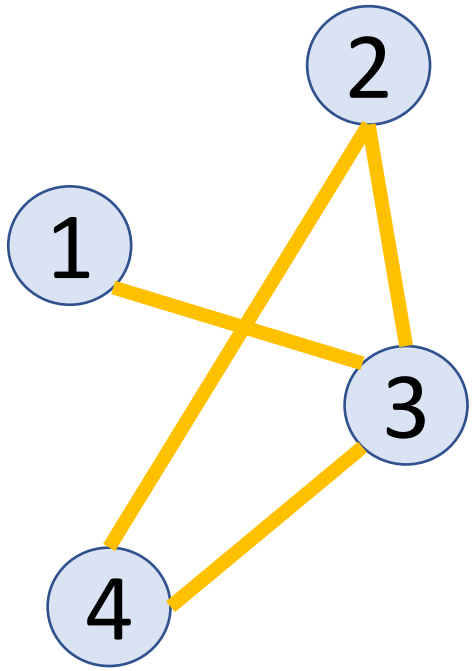
Quiz

- How much space does the adjacency list representation of a graph require, as a function of the number n of vertices and the number m of edges?
- $\Theta(m + n)$

How do we represent graphs?

- Option 1: adjacency matrix

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

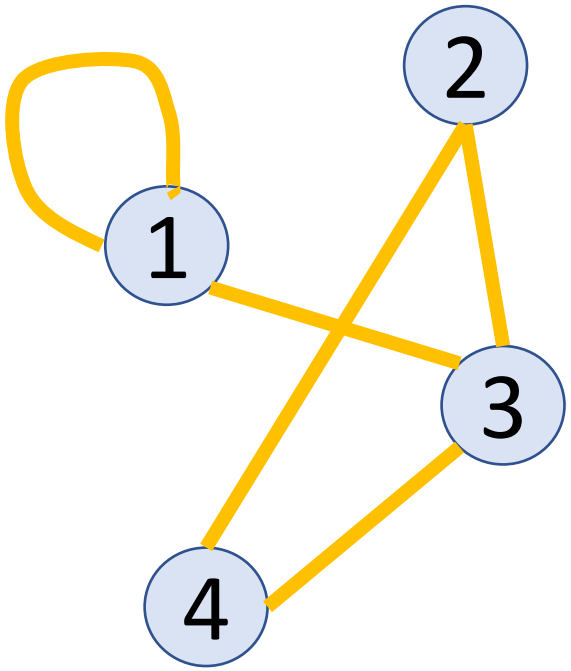


http://web.stanford.edu/class/archive/cs/cs161/cs161.1182/schedule.html

How do we represent graphs?

- Option 1: adjacency matrix

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

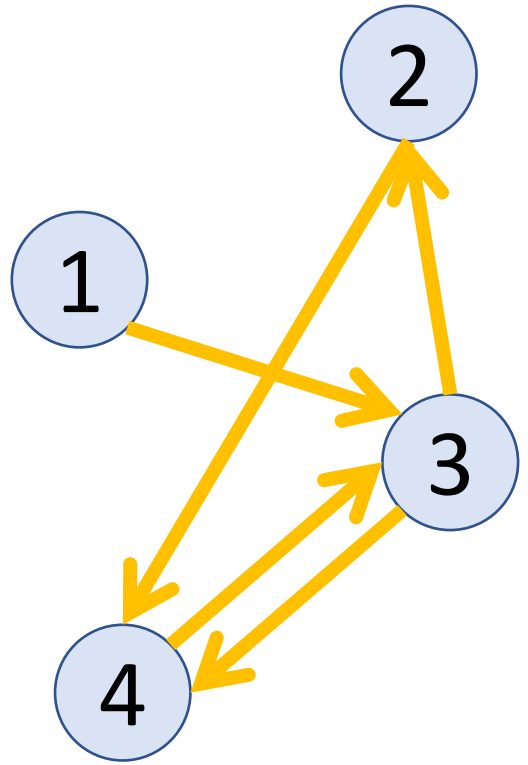


http://web.stanford.edu/class/archive/cs/cs161/cs161.1182/schedule.html

How do we represent graphs?

- Option 1: adjacency matrix

Source \ Destination	1	2	3	4
1	0	0	1	0
2	0	0	0	1
3	0	1	0	1
4	0	0	1	0



http://web.stanford.edu/class/archive/cs/cs161/cs161.1182/schedule.html

Quiz

- How much space does the adjacency Matrix representation of a graph require, as a function of the number n of vertices and the number m of edges?

$\Theta(n^2)$

Sparse vs Dense Graph

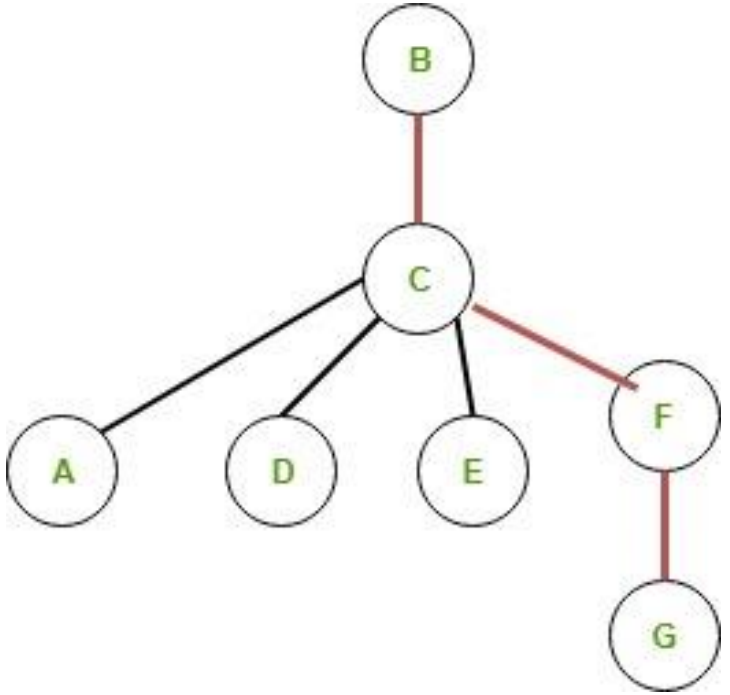
- Informally, a graph is sparse if the number of edges is relatively close to linear in the number of vertices, and dense if this number is closer to quadratic in the number of vertices.

Which is better?

- “it depends.”
- The density of graph :how the number m of edges compares to the number no of vertices.
  - the adjacency matrix is an efficient way to encode a dense graph but is wasteful for a sparse graph.
- It depends on which operations you want to support. adjacency lists make more exploring a graph applications.
- Graphs O(n^2) or more considered Dense.
- Note : the Web graph, where vertices correspond to Web pages and directed edges the Web graph where vertices correspond to Web pages and directed edges to hyperlinks. A conservative Lower bound for estimating the number of vertices is 10 billion.
- The size of the adjacency matrix of this graph, however, is proportional to 100 quintillion (10^{20}). This is way too big to store or process with today’s technology.

Terminology

- The distance between two vertices in a graph is the number of edges in a shortest or minimal path.
- The diameter of graph is the maximum distance between the pair of vertices. It can also be defined as the maximal distance between the pair of vertices.

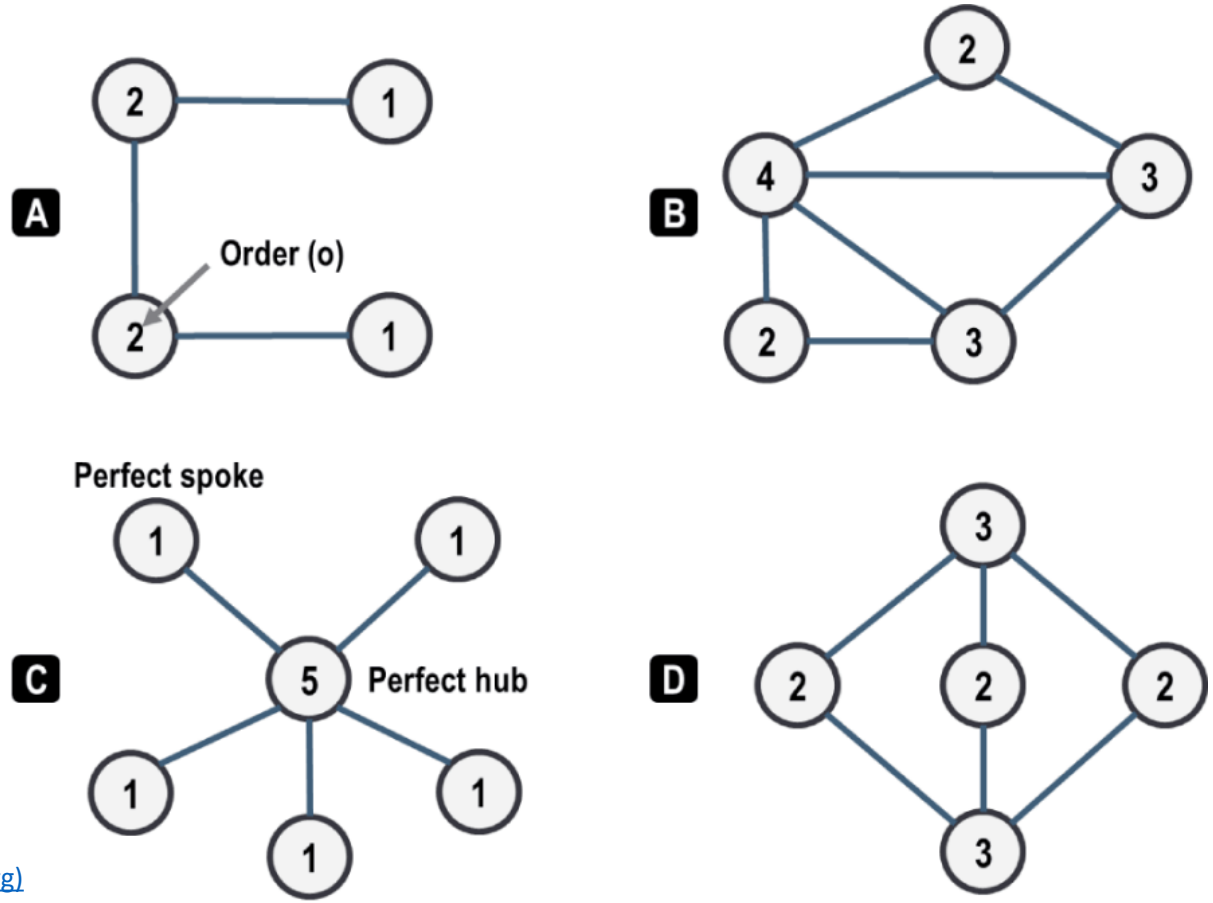


Terminology

- Koenig number (or associated number, eccentricity). A measure of farness based on the number of links needed to reach the most distant node in the graph. The eccentricity ecc(v)ecc(v) of vv in GG is the greatest distance from vv to any other node.

Terminology

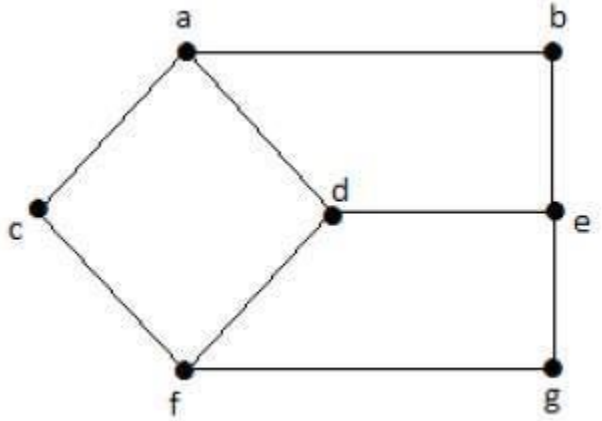
- Order (degree) of a Node (o). The number of its attached links and is a simple, but effective measure of nodal importance.



Diameter of a Graph | The Geography of Transport Systems (transportgeography.org)

Terminology

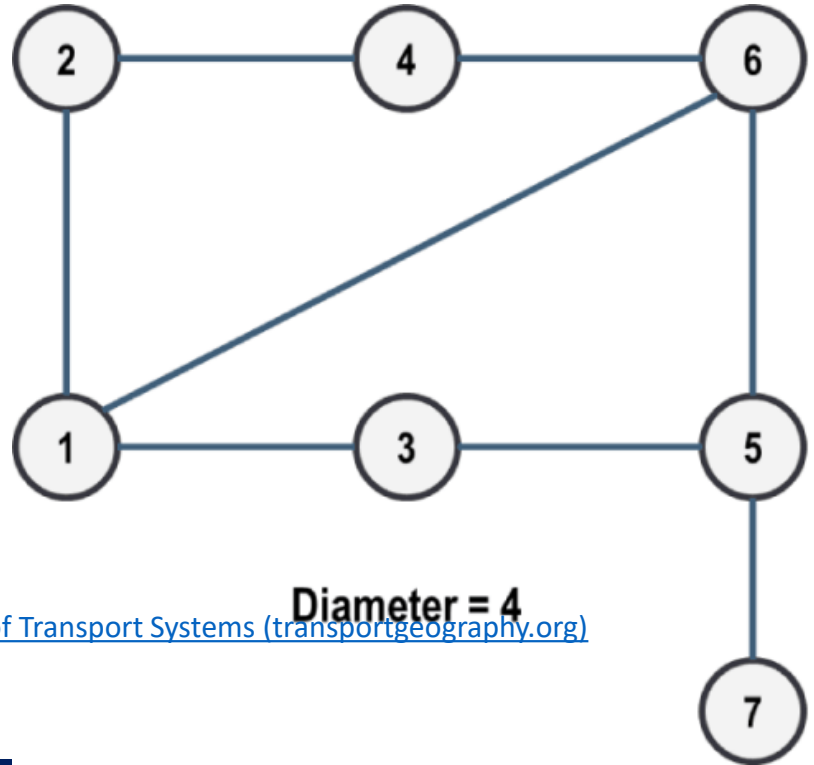
- Eccentricity of a Vertex **Example**
- e(a)=3
- e(b) = 3
- e(c) = 3
- e(d) = 2
- e(e) = 3
- e(f) = 3
- e(g) = 3



Graph Theory - Basic Properties - Tutorialspoint

Terminology

- The maximum eccentricity from all the vertices is considered as the diameter of the Graph G. The maximum among all the distances between a vertex to all other vertices is considered as the diameter of the Graph G. **d(G)**



Diameter = 4

Diameter of a Graph | The Geography of Transport Systems (transportgeography.org)

Shimbel Distance

v	1	2	3	4	5	6	7
1	0	1	1	2	2	1	3
2	1	0	2	1	3	2	4
3	1	2	0	3	1	2	2
4	2	1	3	0	2	1	3
5	2	3	1	2	0	1	1
6	1	2	2	1	1	0	2
7	3	4	2	3	1	2	0

Terminology

- Radius of a Connected Graph
- The minimum eccentricity from all the vertices is considered as the radius of the Graph G. The minimum among all the maximum distances between a vertex to all other vertices is considered as the radius of the Graph G.
- Notation − r(G)
- From all the eccentricities of the vertices in a graph, the radius of the connected graph is the minimum of all those eccentricities.
- **Example**
- In the above graph r(G) = 2, which is the minimum eccentricity for ‘d’.

Terminology

- The radius rad(G)rad(G) of GG is the value of the smallest eccentricity.
  - r(G) = 2, which is the minimum eccentricity
- The diameter diam(G)diam(G) of GG is the value of the greatest eccentricity.
  - d(G) = 3; which is the maximum eccentricity.

- The center of GG is the set of nodes vv such that ecc(v)=rad(G)

Graph Search and Its Applications

Motivation

- Checking connectivity. In a physical network, such as a road network or a network of computers, an important sanity check is that you can get anywhere from anywhere else.
- the movie network, where vertices correspond to movie actors, and two actors are connected by an undirected edge whenever they appeared in the same movie. The most famous statistic of this type is the Bacon number, which is the minimum number of hops through the movie network needed to reach the fairly ubiquitous actor Kevin Bacon. So, Kevin Bacon himself has a Bacon number of 0, every actor who has appeared in a movie with Kevin Bacon has a Bacon number of 1, every actor who has appeared with an actor whose Bacon number is 1 but who is not Kevin Bacon himself has a Bacon number of 2, and so on.
  - The Bacon number is a riff on the older concept of the Erdös number, named after the famous mathematician Paul Erdös, which measures the number of degrees of separation from Erdös in the co-authorship graph (where vertices are researchers, and there is an edge between each pair of researchers who have co-authored apaper).

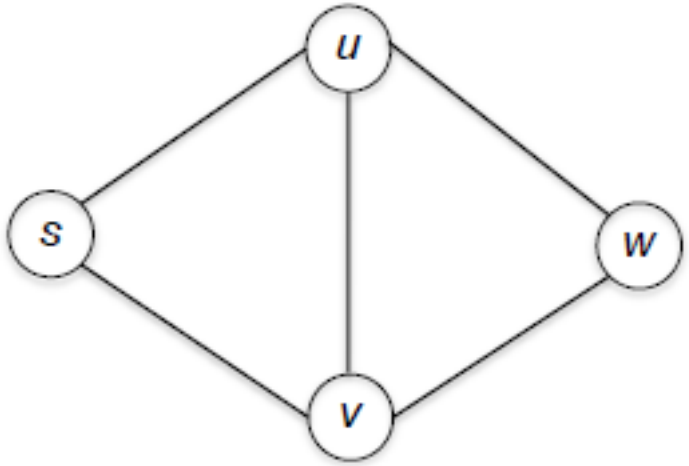
https://oracleofbacon.org/

Problem: Graph Search

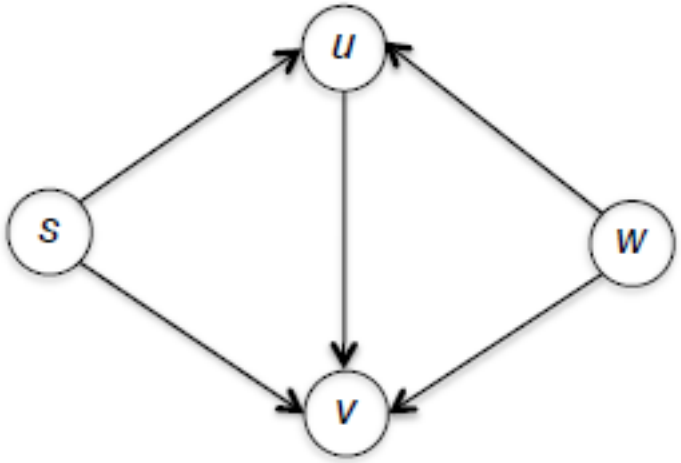
- Input: An undirected or directed graph G = (V,E), and a starting vertex $s \in V$.
- Goal: Identify the vertices of V reachable from s in G.
- The graph search strategies
  - breadth-first search
  - depth-first search
- The goal is to finds all the reachable vertices, taking care to avoid exploring anything twice.

Problem: Graph Search

- A vertex v being "reachable," means that there is a sequence of edges in G that travels from s to v.
-



An undirected graph
the set of vertices reachable from s
is {s, u, v,w}.



A directed graph
the set of vertices reachable from s
is {s, u, v}.

Generic Search

- Input: graph G = (V,E) and a vertex $s \in V$.
- Post condition: a vertex is reachable from s if and only if it is marked as "explored."

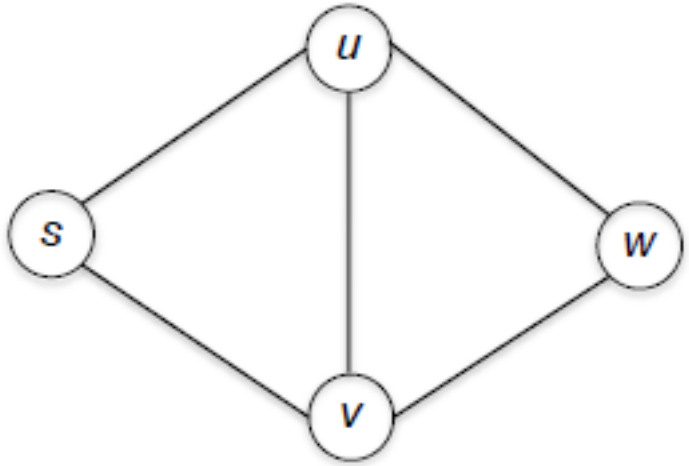
1. mark s as explored, all other vertices as unexplored
2. while there is an edge (v,w) ∈ E with v explored and w unexplored do

   choose some edge (v,w)

   mark w as explored

Generic Search-**Example**

- initially s is marked as explored.
- In the first iteration of the while loop, two edges meet the loop condition: (s, u) and (s, v). The Generic Search algorithm chooses one of these edges—(s, u), say—and marks u as explored.
- In the second iteration of the loop, there are again two choices: (s, v) and (u,w). The algorithm might choose (u,w), in which case w is marked as explored.
- With one more iteration (after choosing either (s, v) or (w, v)), v is marked as explored.

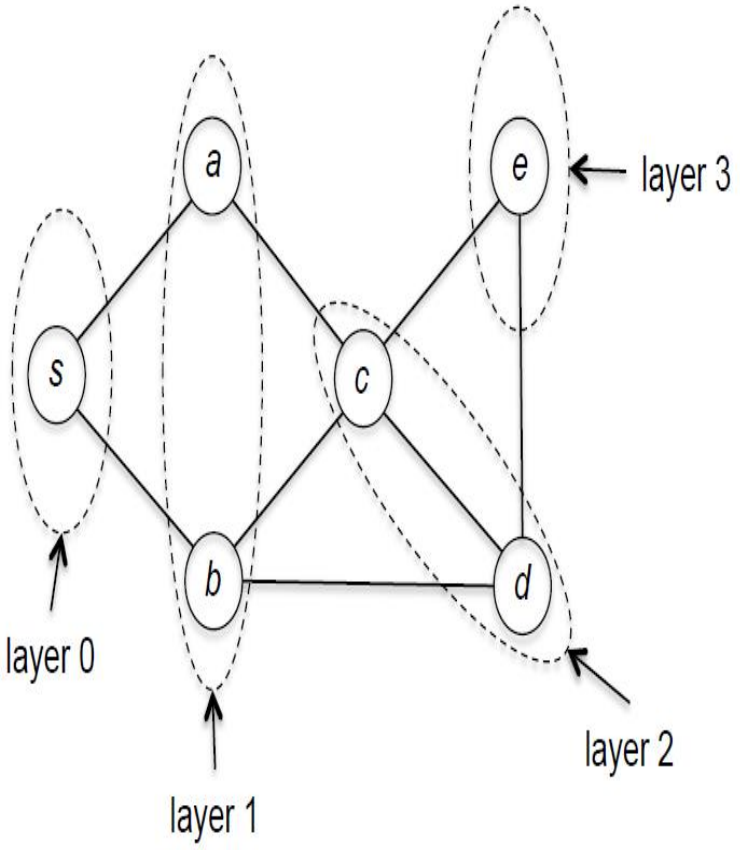


Generic Search

- Generic graph search algorithm is underspecified, as multiple edges (v,w) can be eligible for selection in an iteration of the while loop.
- Breadth-first search and depth-first search correspond to two specific decisions about which edge to explore next.

Breadth-First Search

- Breadth-first search explores the vertices of a graph in layers. Layer 0 contains the starting vertex s and nothing else. Layer 1 is the set of vertices that are one hop away from s—that is, s's neighbors. These are the vertices that are explored immediately after s.
- Breadth-first search explores all of layer-i vertices immediately after completing its exploration of layer-(i – 1) vertices. (Vertices not reachable from s do not belong to any layer.)



Quiz

- Consider an undirected graph with n (2 vertices. What are the minimum and maximum number of different layers that the graph could have, respectively?

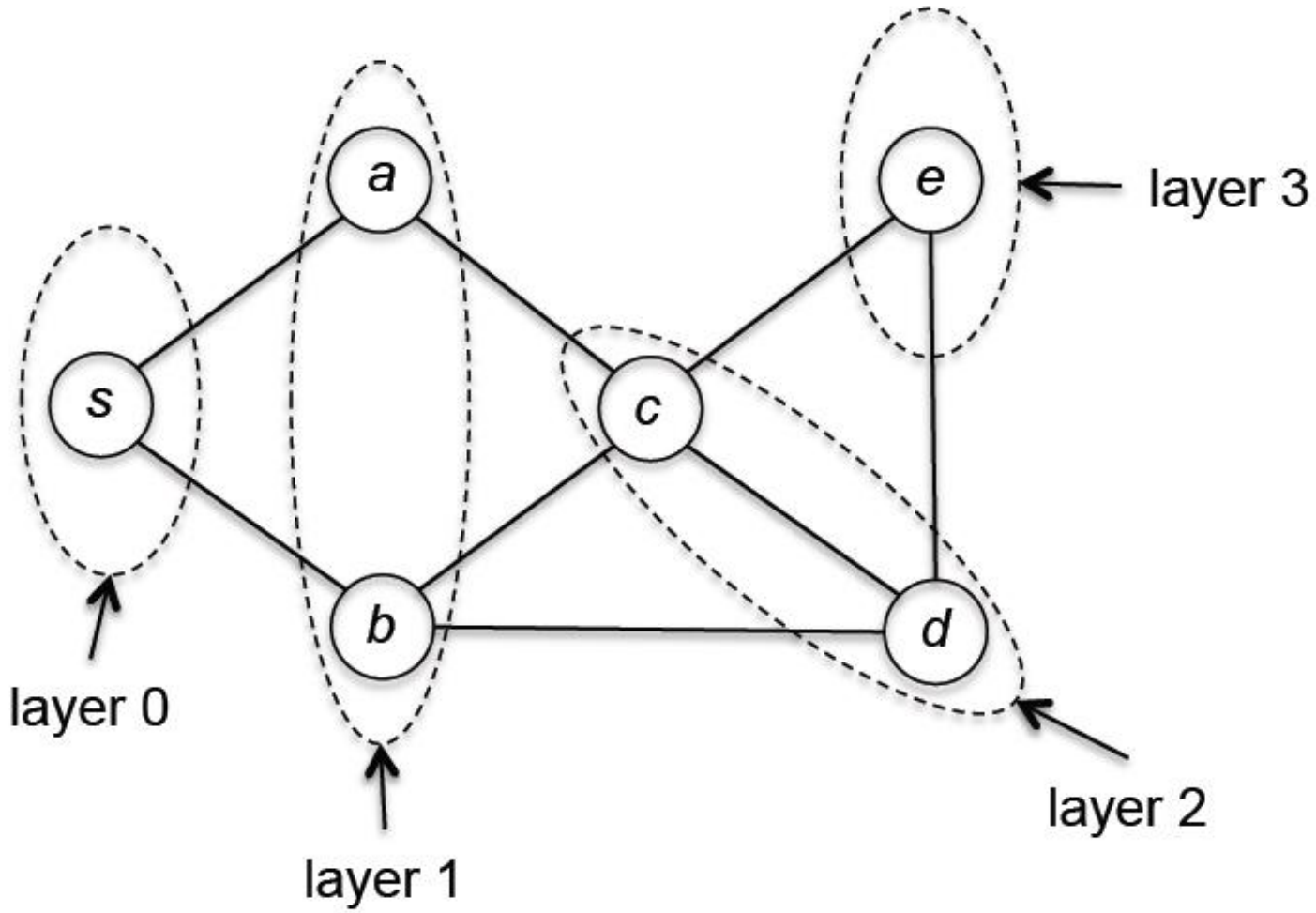
2 and n

Pseudocode for BFS

- Implementing breadth-first search in linear time requires a queue (simple "first-in first-out" data structure known as a queue).
- Input: graph G = (V,E) in adjacency-list representation, and a vertex $s \in V$.
- Postcondition: a vertex is reachable from s if and only if it is marked as "explored."

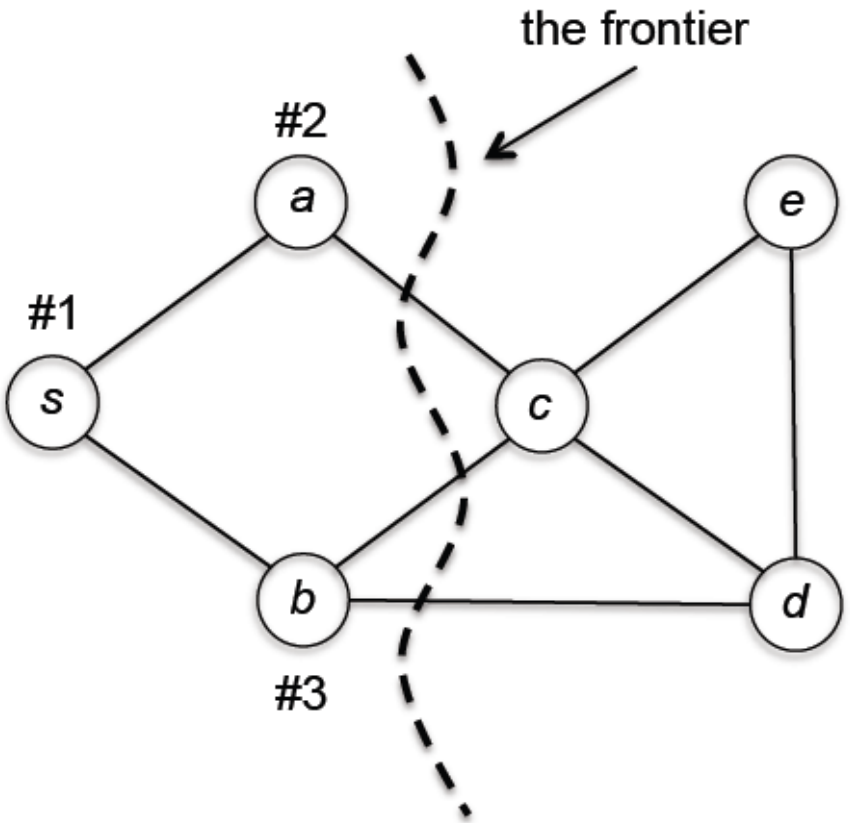
```
1.    mark s as explored, all other vertices as unexplored
2.     Q := a queue data structure, initialized with s
3.    while Q is not empty do
4.                remove the vertex from the front of Q, call it v
5.                 for each edge (v,w) in v's adjacency list do
6.                          if w is unexplored then
7.                                    mark w as explored
8.                                   add w to the end of Q
```

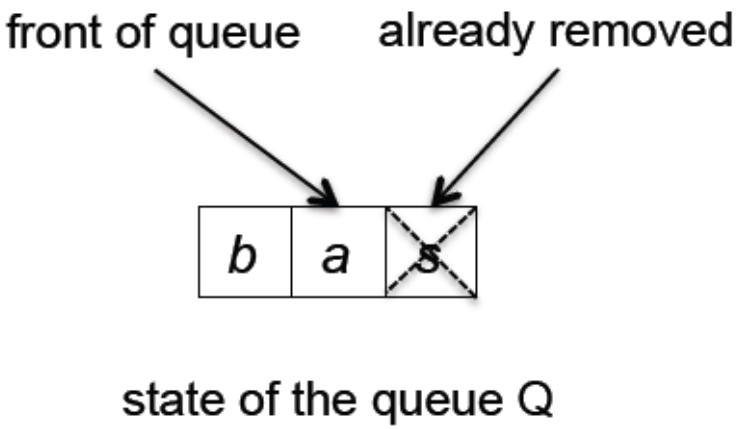
Example (1)



Example (2)

- The first iteration of the while loop extracts s from the queue Q and the subsequent for loop examines the edges (s, a)and (s, b)

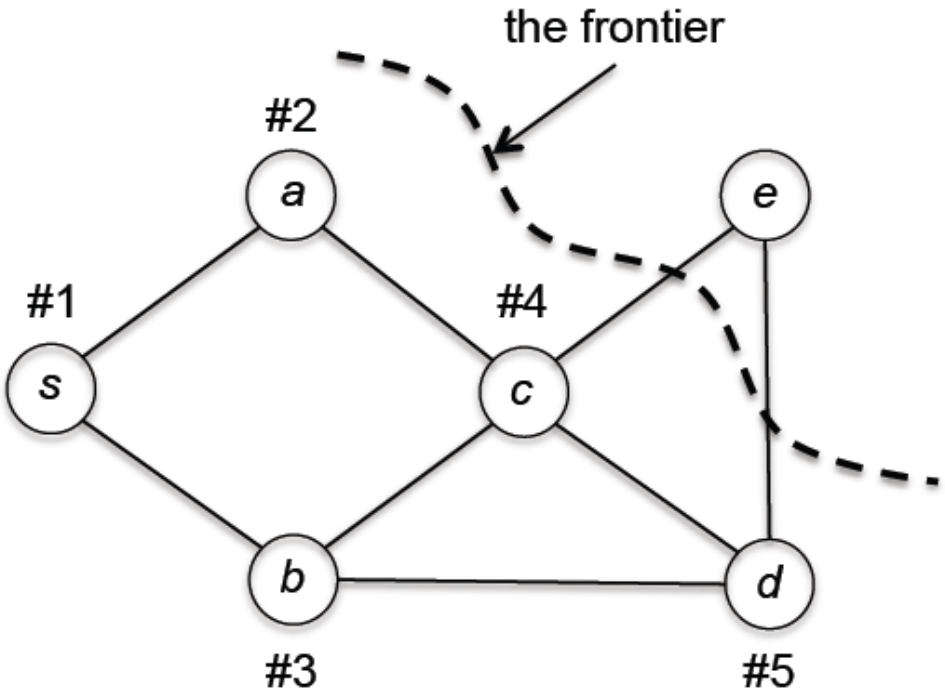


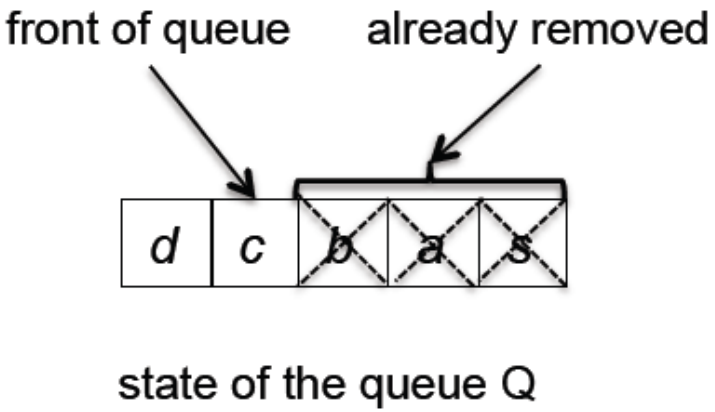


state of the queue Q

Example (3)

- The next iteration extracts the vertex a, incident edges (s, a) and (a, c). It skips (s, a)➔ s is already marked as explored, and adds vertex c to the end of the queue. The third iteration extracts b and adds vertex d (because s and c are already marked as explored, they are skipped over)





state of the queue Q

Example (4)

- In the fourth iteration, the vertex c is removed from the front of the queue. Of its neighbors, the vertex e is added to the end of the queue.
- The final two iterations extract d and then e from the queue, and verify that all of their neighbors have already been explored.
- The queue is then empty, and the algorithm halts.
- Breadth-first search discovers all the vertices reachable from the starting vertex, and it runs in linear time.

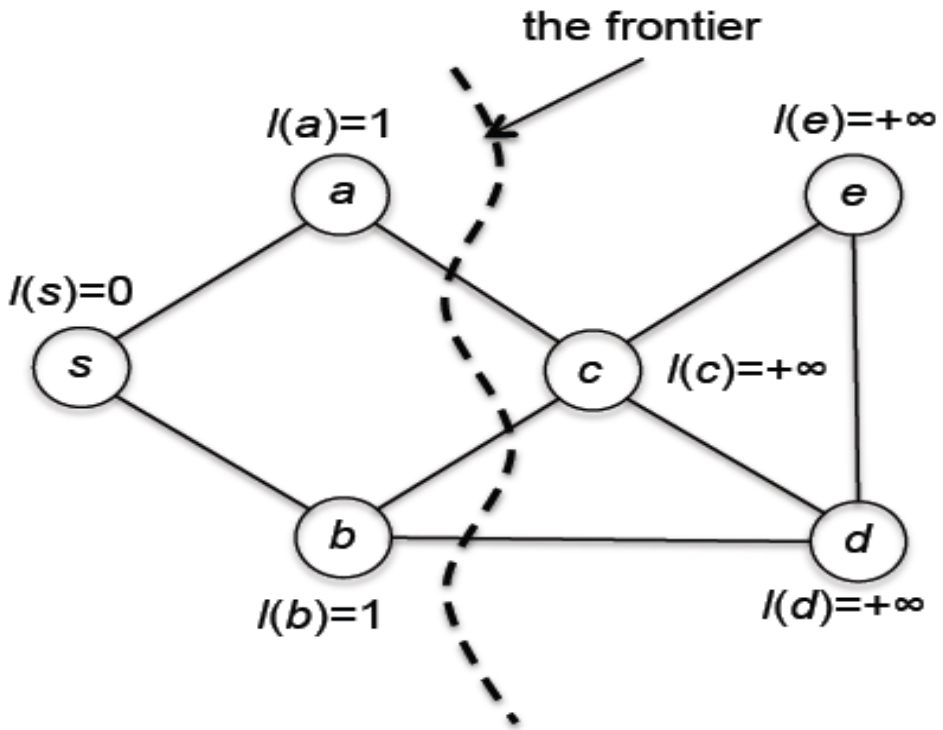
Complexity Analysis of BFS

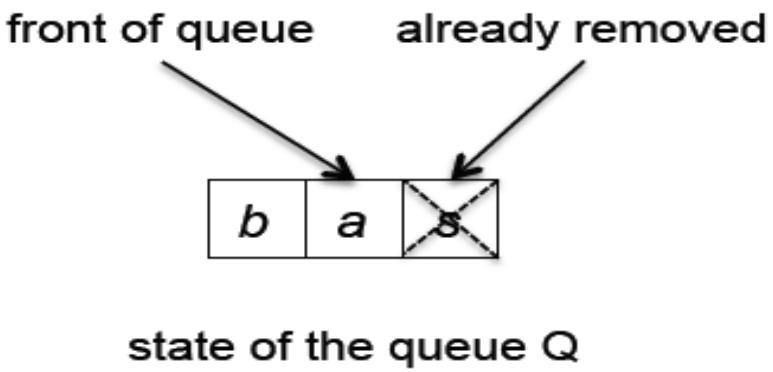
```
1.  mark s as explored, all other vertices as unexplored
2.   Q := a queue data structure, initialized with s
3.  while Q is not empty do
4.                  remove the vertex from the front of Q, call
    it v
5.                   for each edge (v,w) in v's adjacency list do
6.                          if w is unexplored then
7.                                  mark w as explored
8.                                 add w to the end of Q
```

- O(m + n)

Example and Analysis(1)

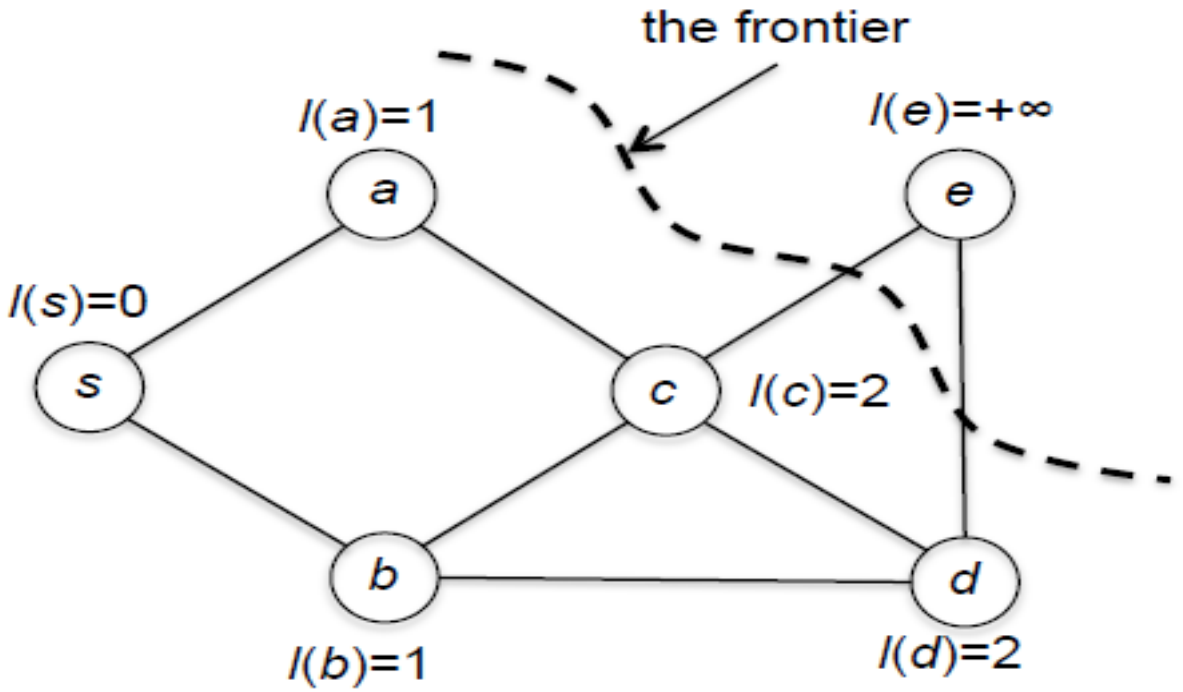
- The first iteration of the while loop discovers the vertices a and b.
- Because s triggered their discovery and l(s) = 0, the algorithm reassigns l(a) and l(b) from +∝ to 1

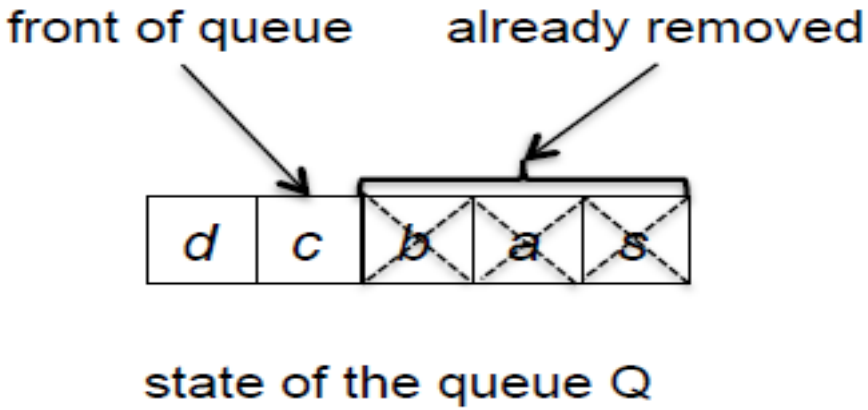




Example and Analysis(2)

- The second iteration of the while loop processes the vertex a, leading to c's discovery. The algorithm reassigns l(c) from +1 to l(a) + 1, which is 2. Similarly, in the third iteration, l(d) is set to l(b) + 1, which is also 2.





Example and Analysis(3)

- The fourth iteration discovers the final vertex e via the vertex c, and sets l(e) to l(c) + 1, which is 3. At this point, for every vertex v, l(v) equals the true shortest-path distance dist(s, v), which also equals the number of the layer that contains v.
- For every undirectedor directed graph G = (V,E) in adjacency-list representation and for every starting vertex s ∈ V :
- (a) At the conclusion of Augmented-BFS, for every vertex v ∈ V , the value of l(v) equals the length dist(s, v) of a shortest path from s to v in G (or +1, if no such path exists).
- (b) The running time of Augmented-BFS is O(m+n), where m = |E| and n = |V |.

Quiz

True or false

1-BFS is a linear time algorithm.

1-TRUE BFS runs in time O(n +m)

2-Every undirected connected graph on n vertices has exactly $n-1$ edges.

2-FALSE (G that is a cycle)

The running time of UCC

- Each call to BFS from a vertex I runs in O(mi + n) time, where mi and nİ denote the number of edges and vertices, respectively, in i's connected component. As BFS is called only once for each connected component, and each vertex or edge of G participates in exactly one component, the combined
- running time of all the BFS calls is

$$O(\sum_i mi + \sum_i ni) = \text{O(m+n).}$$

The initialization and additional bookkeeping performed by the algorithm requires only O(n) time, so the final running time is O(m+n).

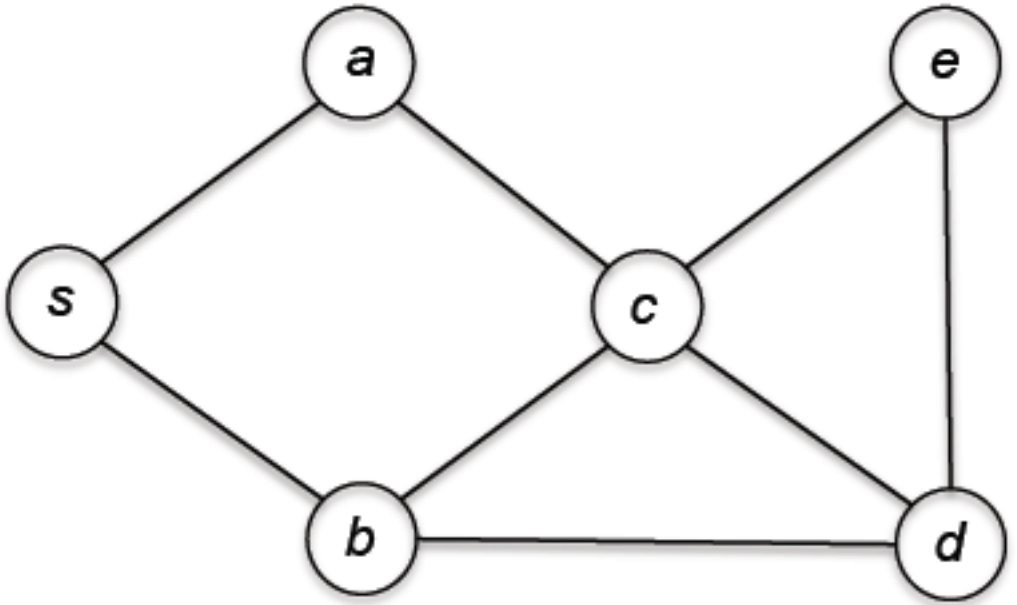
Depth-First Search

Depth-First Search

- Why do we need another graph search strategy? After all, breadth-first search seems pretty awesome—it finds all the vertices reachable from the starting vertex in linear time, and can even compute shortest-path distances along the way.
- There's another linear-time graph search strategy, depth-first search (DFS), which comes with its own impressive catalog of applications (not already covered by BFS). For example, we'll see how to use DFS to compute in linear time a topological ordering of the vertices of a directed acyclic graph, as well as the connected components (appropriately defined) of a directed graph.

An Example

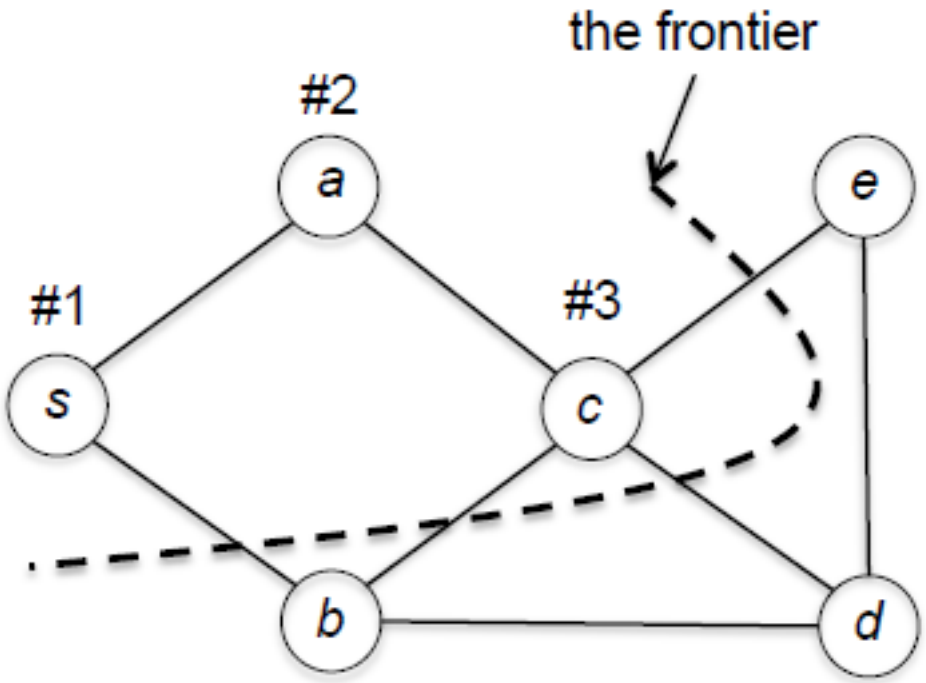
- The depth-first search is its more aggressive cousin, always exploring from the most recently discovered vertex and backtracking only when necessary (like exploring a maze)



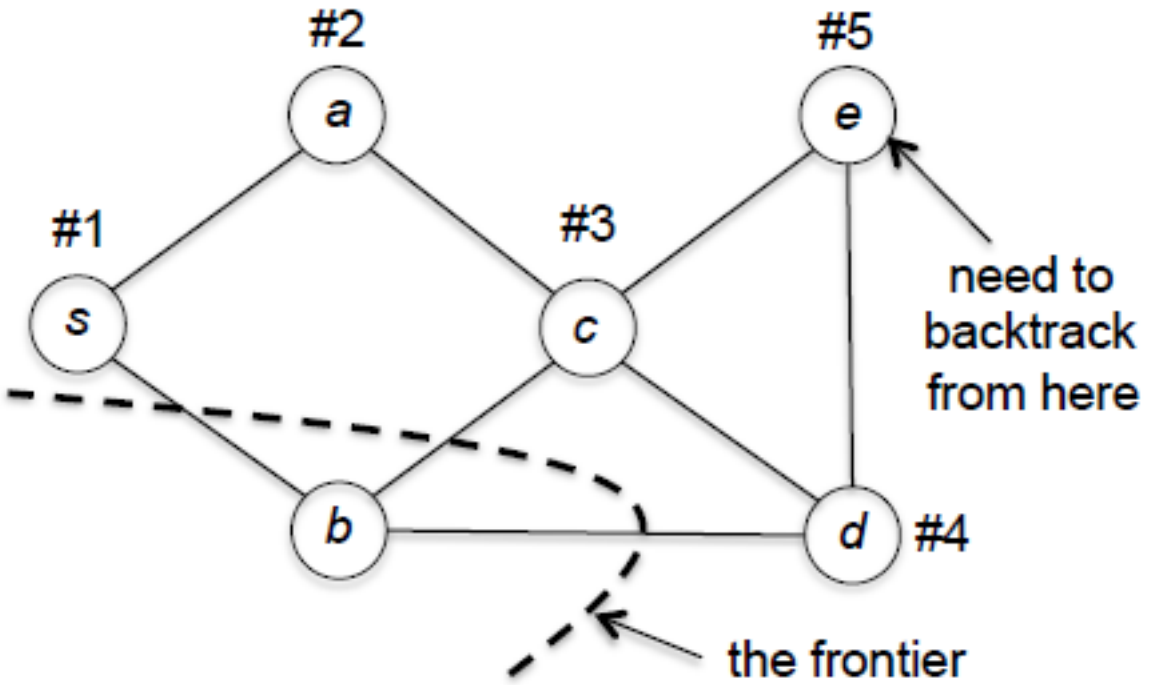
An Example

- Like BFS, DFS marks a vertex as explored the first time it discovers it.
- The first iteration of DFS examines the edges (s, a) and (s, b), in whatever order these edges appear in s's adjacency list. Let's say (s, a) comes first, leading DFS to discover the vertex a and mark it as explored.
- The second iteration of DFS is where it diverges from BFS—rather than considering next s's other layer-1 neighbor b, DFS immediately proceeds to exploring the neighbors of a.
- Then discovers the vertex c, which is where it travels next. Then DFS examines in some order the neighbors of c, the most recently discovered vertex. let's say that DFS discovers d next, followed by e.
- From e, DFS has nowhere to go—both of e's neighbors are already marked as explored. DFS is forced to retreat to the previous vertex, namely d, and resume exploring the rest of its neighbors. From d, DFS will discover the final vertex b (perhaps after checking c and finding it marked as explored).
- Once at b, DFS discovers that all of b's neighbors have already been explored, and must backtrack to the previously visited vertex, which is d. Similarly, because all of d's remaining neighbors are already marked as explored, DFS must rewind further, to c. DFS then retreats further to a (after checking that all of c's remaining neighbors are marked as explored), then to s. It finally stops once it checks s's remaining neighbor (whicis b) and finds it marked as explored

An Example



An Example



Pseudocode for DFS

- One way to think about and implement DFS is to start from the code for BFS and make two changes:
- (i) swap in a stack data structure (which is last-in first-out) for the queue (which is first-in first-out);
- (ii) postpone checking whether a vertex has already been explored until after removing it from the data structure

DFS

DFS (Iterative Version)

Input: graph $G = (V, E)$ in adjacency-list representation, and a vertex $s \in V$.
Postcondition: a vertex is reachable from s if and only if it is marked as "explored."

```
mark all vertices as unexplored
S := a stack data structure, initialized with s
while S is not empty do
    remove ("pop") the vertex v from the front of S
    if v is unexplored then
        mark v as explored
        for each edge (v, w) in v's adjacency list do
            add ("push") w to the front of S
```

Pseudocode for DFS

Recursive Implementation

Depth-first search also has an elegant recursive implementation.[22]

`DFS` (Recursive Version)

Input: graph $G = (V, E)$ in adjacency-list representation, and a vertex $s \in V$.
Postcondition: a vertex is reachable from s if and only if it is marked as "explored."

```
// all vertices unexplored before outer call
mark s as explored
for each edge (s, v) in s's adjacency list do
    if v is unexplored then
        DFS (G, v)
```

Stack for DFS

- In the previous example , the first iteration of DFS's while loop pops the vertex s and pushes its two neighbors onto the stack in some order.
- say, with b first and a second. Because a was the last to be pushed, it is the first to be popped, in the second iteration of the while loop.
- This causes s and c to be pushed onto the stack.
- let's say with c first. The vertex s is popped in the next iteration; since it has already been marked as explored, the algorithm skips it. Then c is popped, and all of its neighbors (a, b, d, and e) are pushed onto the stack, joining the first occurrence of b.

Strongly Connected Components

undirected Graphs

- How to get the number of the connected components in undirected graph(G)?
- Input: G(V,E) in adjacency list representation
- Output: vertices labelled by connected components number
- Run DFS and keep track of Components numbers.

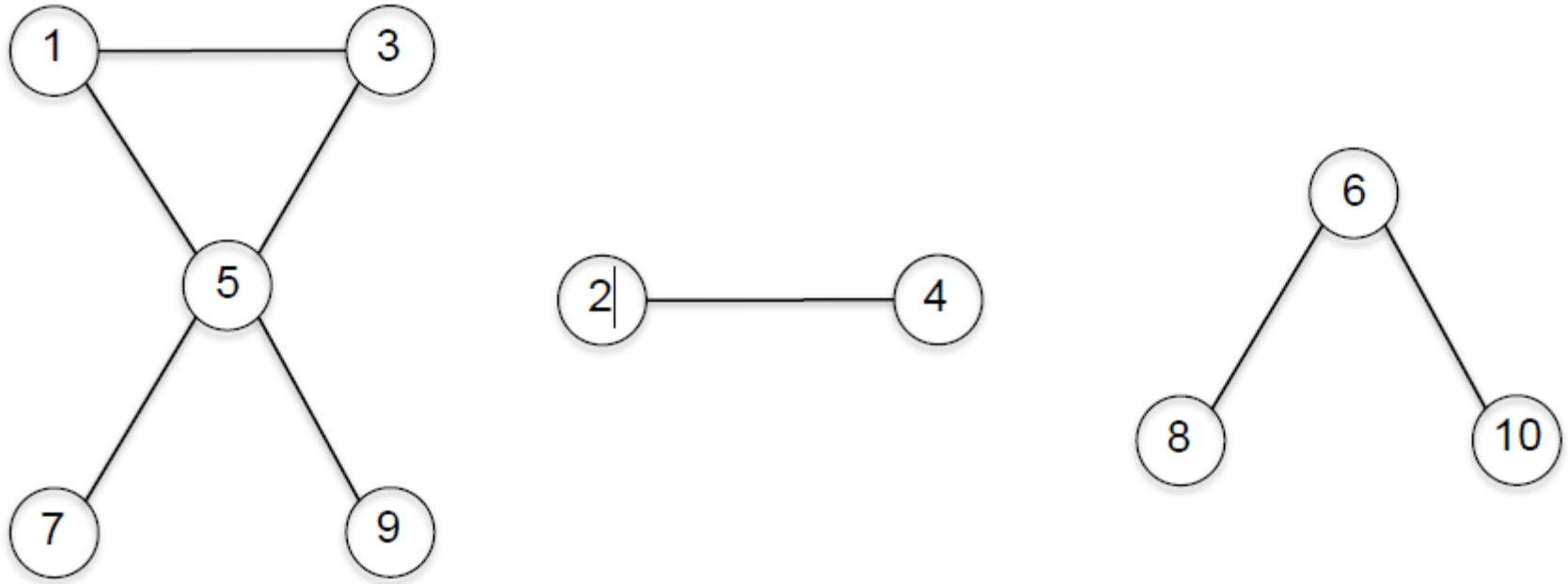
```
DFS − cc(G):
    ccnum = 0
    For all v ∈ V
        visited(v) = False
    For allv ∈:
        If not visited(v) then
            ccnum++
            Explore(v)
```

```
Explore(z):
    visited(z) = True
    cc(z) = ccnum
    for all(z,w) ∈ E //Neibors of z
        if not visited(w) then
            Explore(w)
```

- The above algorithm runs in O(n+m) time where n=|V| and m=|E|.

Connected Components

Connected Components



- {1, 3, 5, 7, 9}, {2, 4}, and {6, 8, 10}.

Undirected Connected Components(UCC)

- Problem: Undirected Connected Components
- Input: An undirected graph G = (V,E).
- Goal: Identify the connected components of G.

Question

- Consider an undirected graph with n vertices and m edges.
- What are the minimum and maximum number of connected components that the graph could have, respectively?
- a) 1 and n − 1
- b) 1 and n
- c) 1 and max{m, n}
- d) 2 and max{m, n}

- (b)

Applications

- Detecting network failures.
- Clustering. Suppose you have a collection of objects that you care about, with each pair annotated as either "similar" or "dissimilar." For example, the objects could be documents (like crawled Web pages or news stories), with similar objects corresponding to near-duplicate documents (perhaps differing only in a timestamp or a headline). Or the objects could be genomes, with two genomes deemed similar if a small number of mutations can transform one into the other.
- Now form an undirected graph G = (V,E), with vertices corresponding to objects and edges corresponding to pairs of similar objects. Intuitively, each connected component of this graph represents a set of objects that share much in common. For example, if the objects are crawled news stories, one might expect the vertices of a connected component to be variations on the same story reported on different Web sites. If the objects are genomes, a connected component might correspond to different individuals belonging to the same species.

The UCC Algorithm

- Computing the connected components of an undirected graph easily reduces to breadth-first search (or other graph search algorithms, such as depth-first search). The idea is to use an outer loop to make a single pass over the vertices, invoking BFS as a subroutine whenever the algorithm encounters a vertex that it has never seen before. This outer loop ensures that the algorithm looks at every vertex at least once. Vertices are initialized as unexplored before the outer loop, and not inside a call to BFS. The algorithm also maintains a field cc(v) for each vertex v, to remember which connected component contains it. By identifying each vertex of V with its position in the vertex array, we can assume that V = {1, 2, 3, . . . ,n}.

UCC Algorithm

Input: undirected graph $G = (V, E)$ in adjacency-list representation, with $V = \{1, 2, 3, \ldots, n\}$.
Postcondition: for every $u, v \in V$, $cc(u) = cc(v)$ if and only if u, v are in the same connected component.

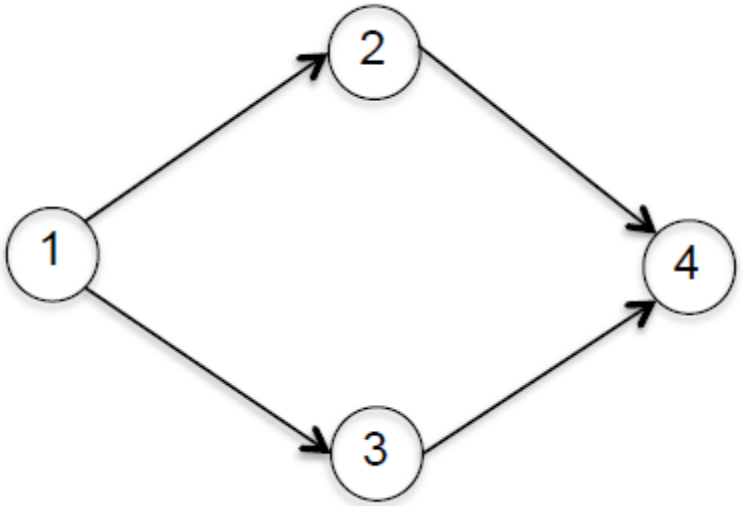
```
mark all vertices as unexplored
numCC := 0
for i := 1 to n do                    // try all vertices
    if i is unexplored then           // avoid redundancy
        numCC := numCC + 1            // new component
        // call BFS starting at i (lines 2-8)
        Q := a queue data structure, initialized with i
        while Q is not empty do
            remove the vertex from the front of Q, call it v
            cc(v) := numCC
            for each (v, w) in v's adjacency list do
                if w is unexplored then
                    mark w as explored
                    add w to the end of Q
```

Running time

- The running time of UCC is O(m + n),

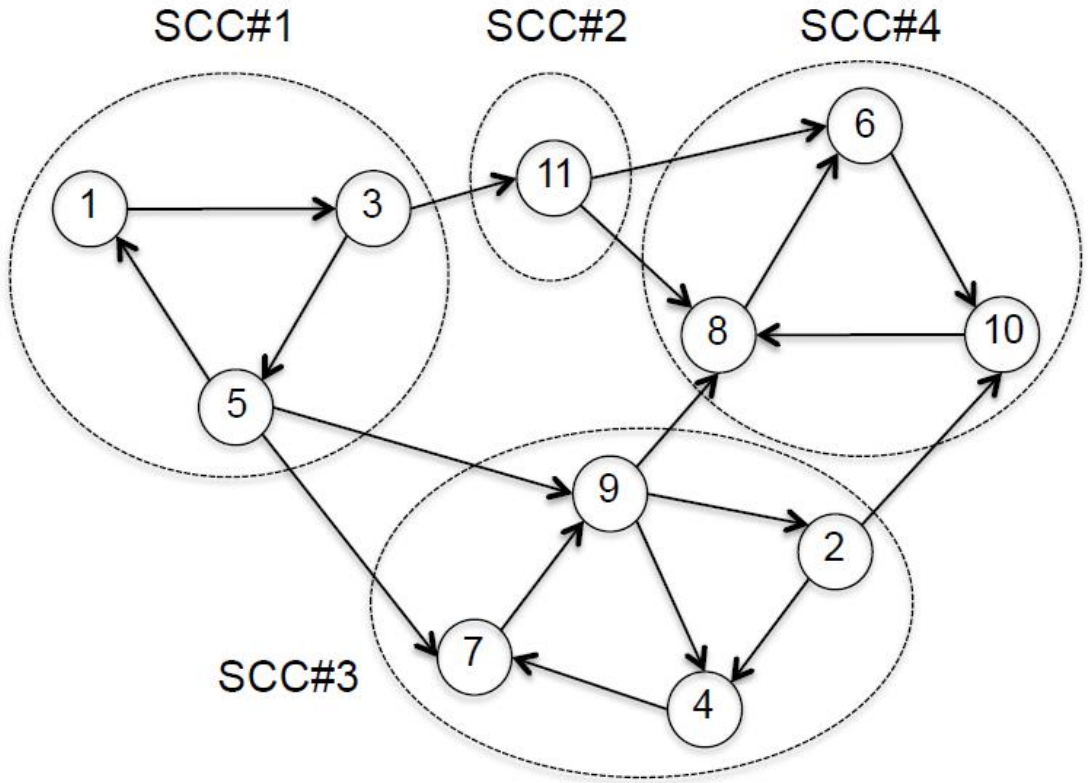
Computing Strongly Connected Components

- computing the strongly connected components of a directed graph



Computing Strongly Connected Components

- A strongly connected component or SCC of a directed graph is a maximal subset $S \subseteq V$ of vertices such that there is a directed path from any vertex in S to any other vertex in S



- BFS or DFS start at 1

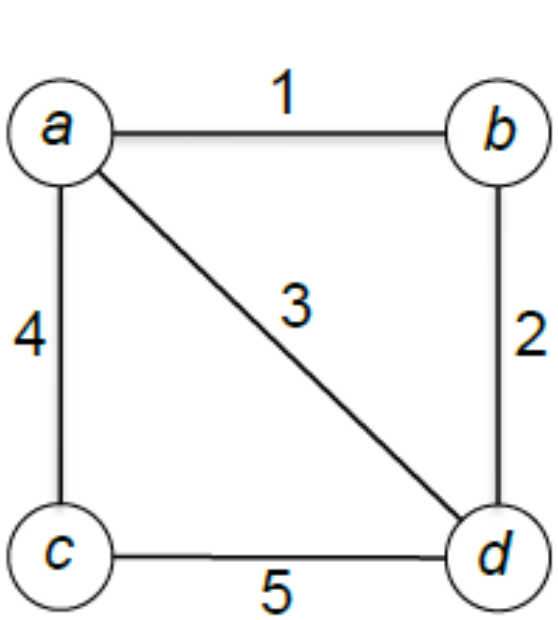
Spanning tree

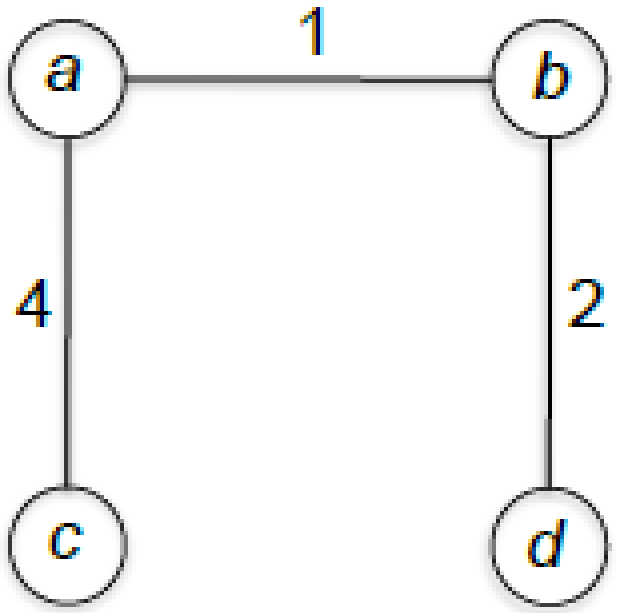
Problem: Minimum Spanning Tree (MST)

Input: A connected undirected graph $G = (V, E)$ and a real-valued cost c_e for each edge $e \in E$.

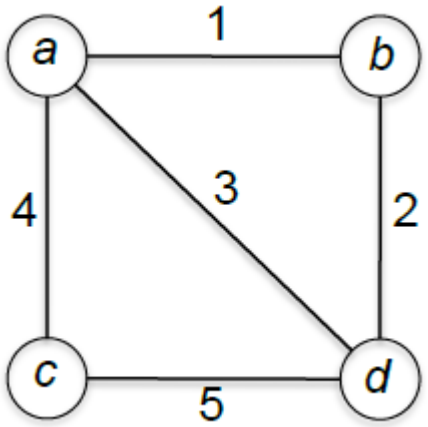
Output: A spanning tree $T \subseteq E$ of G with the minimum-possible sum $\sum_{e \in T} c_e$ of edge costs.

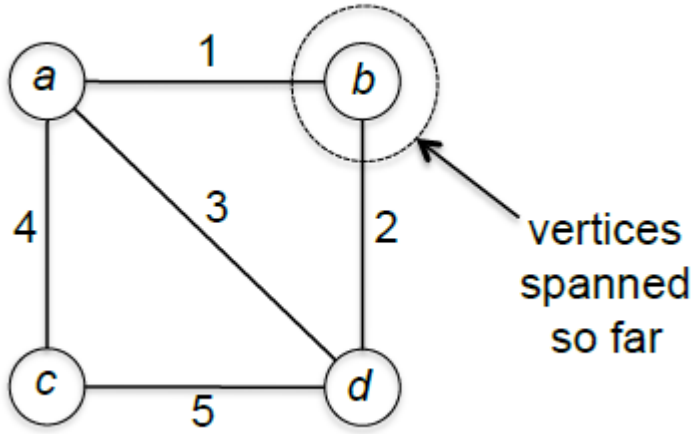
Example

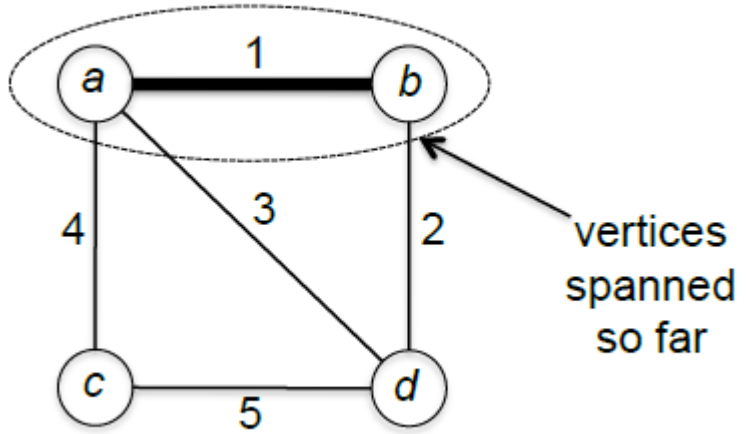


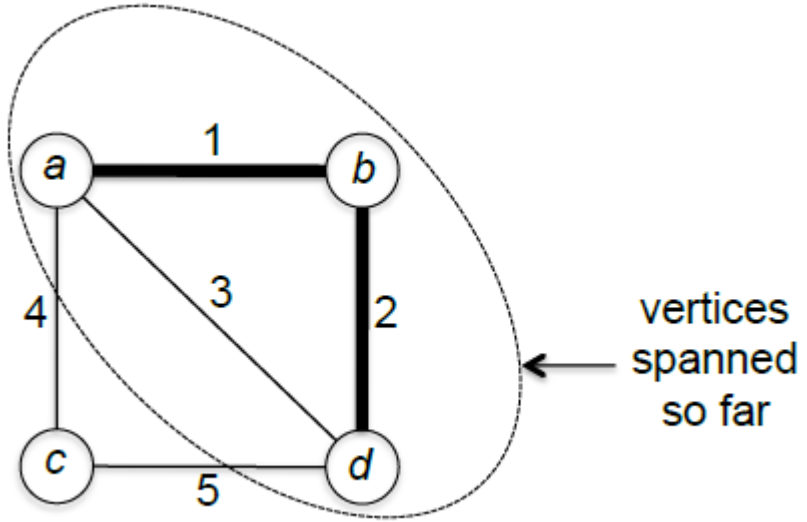


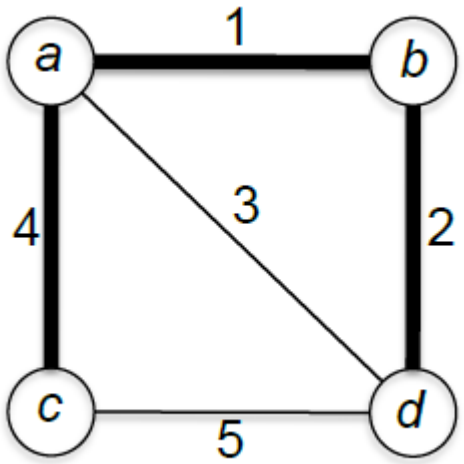
Prim's Algorithm











Prim Algorithm

Prim

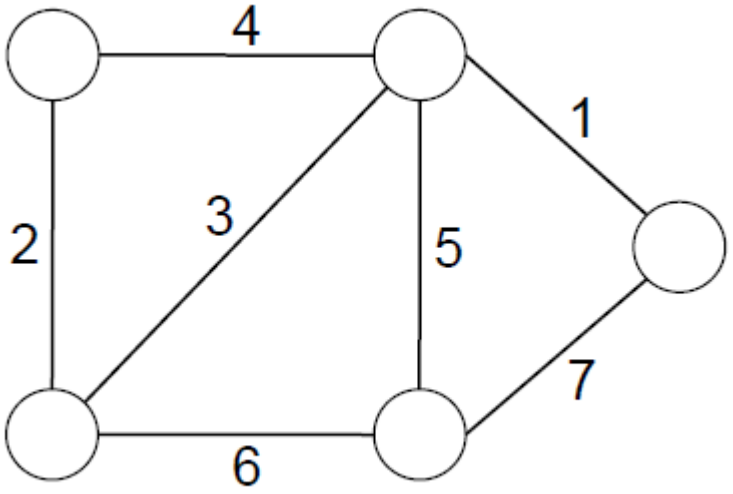
Input: connected undirected graph $G = (V, E)$ in adjacency-list representation and a cost c_e for each edge $e \in E$.

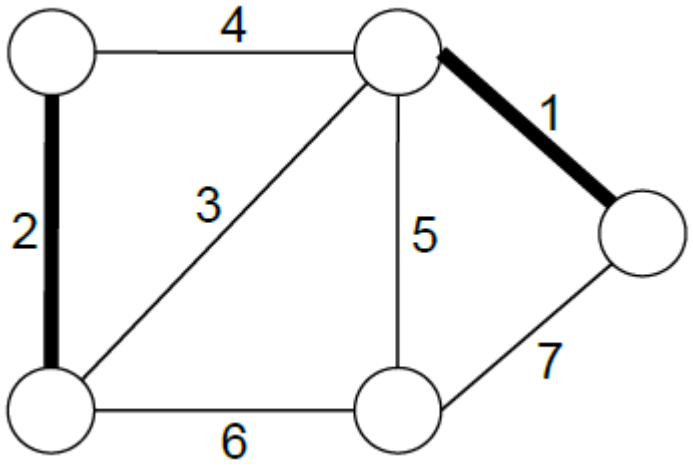
Output: the edges of a minimum spanning tree of G.

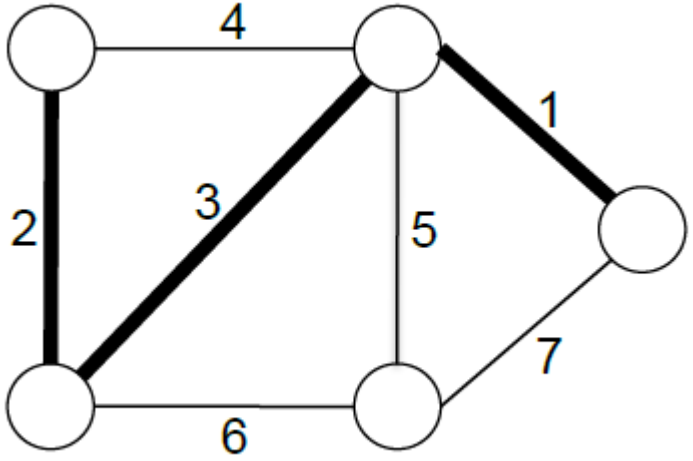
```
// Initialization
X := {s}    // s is an arbitrarily chosen vertex
T := ∅      // invariant:  the edges in T span X
// Main loop
while there is an edge (v, w) with v ∈ X, w ∉ X do
    (v*, w*) := a minimum-cost such edge
    add vertex w* to X
    add edge (v*, w*) to T
return T
```

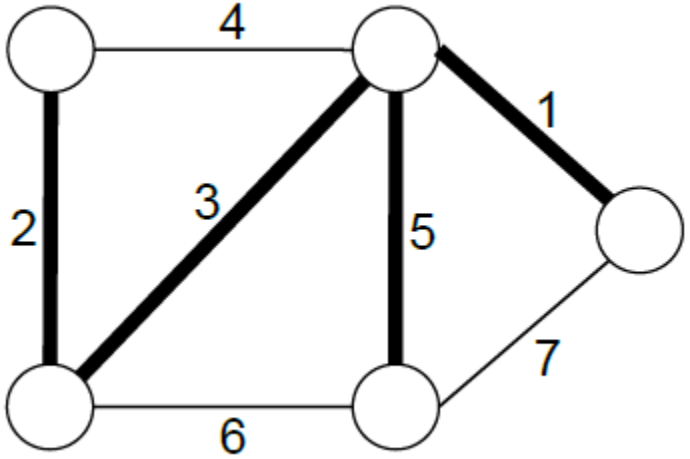
- Prim Algorithm is _____ algorithm.
- Is it Correct?
- Perfect implementation

Kruskal's Algorithm









Algorithm

Kruskal

Input: connected undirected graph $G = (V, E)$ in adjacency-list representation and a cost c_e for each edge $e \in E$.

Output: the edges of a minimum spanning tree of G.

```
// Preprocessing
T := ∅
sort edges of E by cost   // e.g., using MergeSort[26]
// Main loop
for each e ∈ E, in nondecreasing order of cost do
    if T ∪ {e} is acyclic then
        T := T ∪ {e}
return T
```

Running Time

- O(mn)
- In the preprocessing step, the algorithm sorts the edge array of the input graph, which has m entries. With a good sorting algorithm (like MergeSort), this step contributes O(mlog n)
- The main loop has m iterations. Each iteration is responsible for checking whether the edge e = (v,w) under examination can be added to the solution-so-far T without creating a cycle
- adding e to T creates a cycle if and only if T already contains a v-w path. This condition can be checked in linear time using any reasonable graph search algorithm, like breadth- or depth-first search starting from v as an acyclic graph with n vertices, has at most n − 1 edges. The per-iteration running time is therefore O(n), for an overall running time of O(mn).

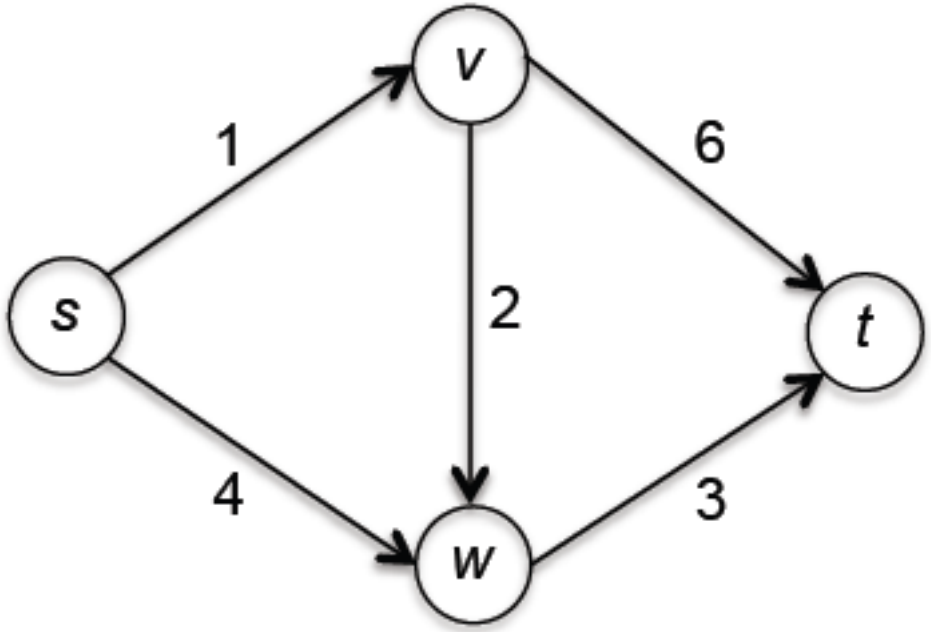
Dijkstra’s Shortest-Path Algorithm

Problem Definition

- Discovered by Edsger W. Dijkstra in 1956
- Dijkstra's algorithm solves the single-source shortest path problem(The term "source" in the name of the problem refers to the given starting vertex. We've already used the term "source vertex" to mean a vertex of a directed graph with no incoming edges).
- Input: A directed graph G = (V,E), a starting vertex s∈ V, and a nonnegative length $\mathcal{L}_e$for each edge e ∈ E.
- Output: dist(s, v) for every vertex v ∈ V.
- dist(s, v) denotes the length of a shortest path from s to v. (If there is no path at all from s to v, then dist(s, v) is +∝.) By the length of a path, we mean the sum of the lengths of its edges.. A shortest path from a vertex v to a vertex w is one with minimum length (among all v-w paths)

Quiz

- Consider the following input to the single-source shortest path problem, with starting vertex s and with each edge labeled with its length
- What are the shortest-path distances to s, v, w, and t, respectively?



0, 1, 3, 6

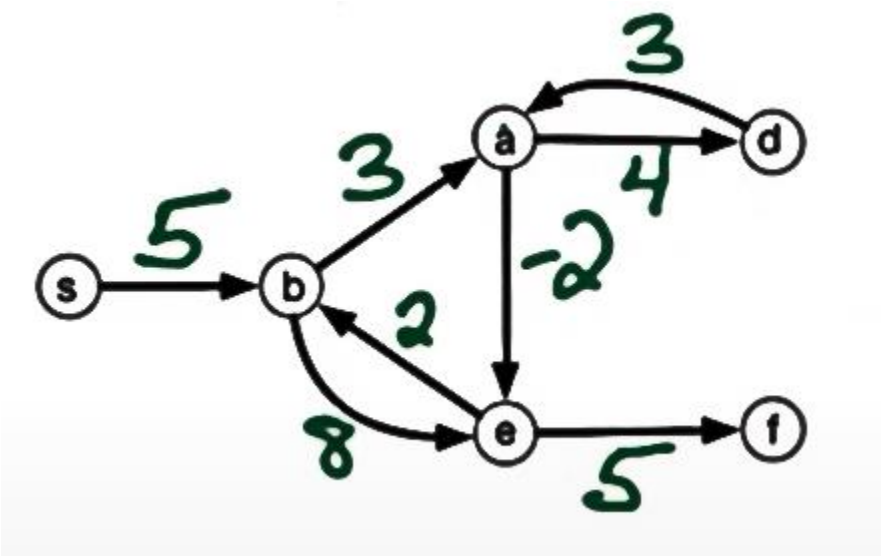
Quiz

you want to compute a profitable sequence of financial transactions that involves both buying and selling. This problem corresponds to a shortest path in a graph with edge lengths that are both positive and negative. Is Dijkstra's algorithm is the best algorithm for this problem.

You should not use Dijkstra's algorithm in applications with negative edge lengths

Shortest path algorithms using DP

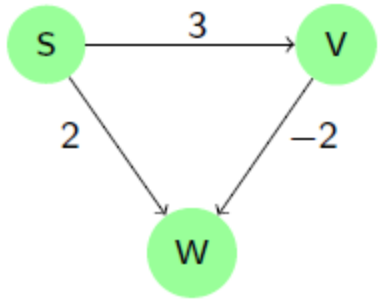
- Input: Directed $G = (V, E)$ with edge weights $w(e)$.
- Dijkstra's algorithm finds for all $z \in V$, $dist(z) =$ length of shortest path from s to z
- *dist(s)=0*
- *dist(b)=5*
- *dist(a)=8*
- *dist(e)=6*



- *O(n=m)logn*

Question

- This question concerns the (in)correctness of Dijkstra's algorithm for graphs with negative edge lengths. Consider a directed graph with vertices s,v,ws,v,w and three edges: (s,v)(s,v) with length 3, (s,w)(s,w) with length 2, and (v,w)(v,w) with length -2. What does Dijkstra's algorithm compute as the length of a shortest ss-ww path, and what is the correct answer?



  - 2 and 2
  - 2 and 0
  - 1 and 2
  - 2 and 1
  - D

- for shortest path P from s to z, P visits every vertex at most once if no negative weight cycle. So P is of length ≤n-1 edges.

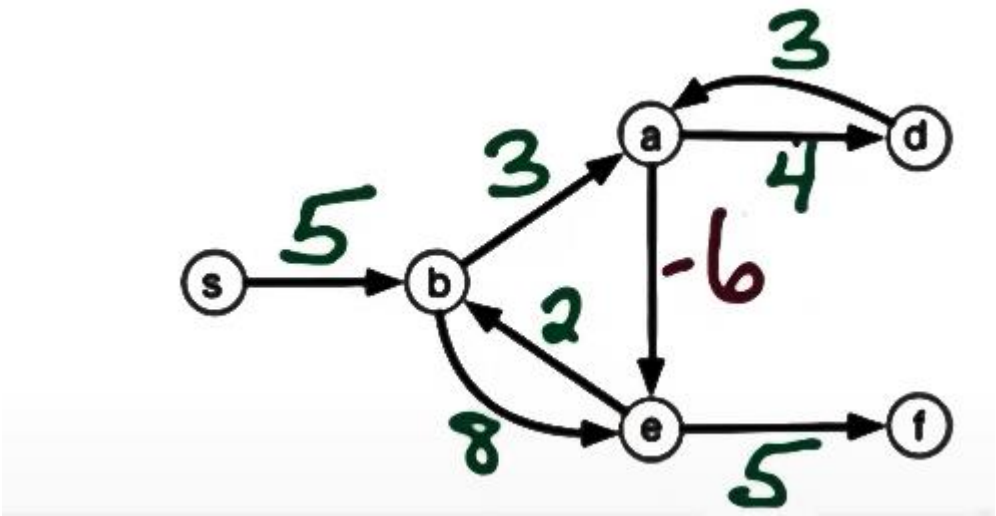
- DP idea: use $i = 0\ to\ n-1$ edges on the path.
- $For\ 0 \leq i \leq n-1, and\ z \in V,$
- let $D(i,z) =$ length of shortest path from s to z using $\leq i$ edges.

- Base case: D(0,s)=0,
- and for z≠s: D(0,z)=∞.

- For $i \geq 1$: look at a shortest path P from s to z using $\leq i$ edges.
- Thus, $D(i,z) = \min\{ D(i-1,z), min_y \{D(i-1,y) + w(y,z)\}\}$.

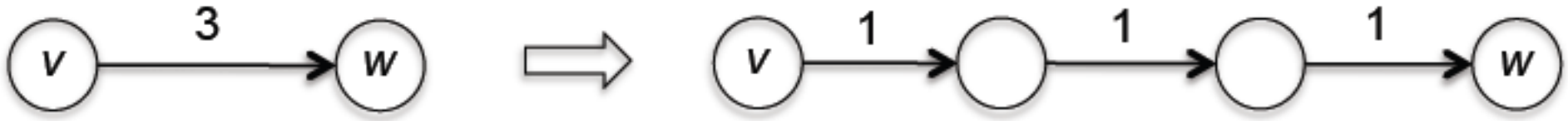
Negative weight cycle

- Shortest path to d
- $S \rightarrow b \rightarrow a \rightarrow e \rightarrow b \rightarrow a \rightarrow d$
- Repeat b e a b -1 each time
- This a walk not a path



Why Not Breadth-First Search?

- breadth-first search computes the minimum number of edges in a path from the starting vertex to every other vertex. This is the special case of the single-source shortest path problem in which every edge has length 1. Can't we just think of an edge with a longer length as a path of edges that each have length 1?



- The major problem➔it blows up the size of the graph. Breadth-first search would run in time linear in the size of the expanded graph, but this is not necessarily close to linear time in the size of the original graph.
- When all the edges have length 1, it's equivalent to breadth-first search

- Given a graph V without cycles. What is the maximum number of edges(complete Graph)
  - Undirected: $\frac{v(v-1)}{2}$
  - Directed $v(v-1)$

Dijkstra's Algorithm- Pseudocode

`Dijkstra`

Input: directed graph $G = (V, E)$ in adjacency-list representation, a vertex $s \in V$, a length $\ell_e \geq 0$ for each $e \in E$.

Postcondition: for every vertex v, the value $len(v)$ equals the true shortest-path distance $dist(s, v)$.

```
// Initialization
1 X := {s}
2 len(s) := 0, len(v) := +∞ for every v ≠ s
// Main loop
3 while there is an edge (v, w) with v ∈ X, w ∉ X do
4     (v*, w*) := such an edge minimizing len(v) + ℓ_vw
5     add w* to X
6     len(w*) := len(v*) + ℓ_v*w*
```

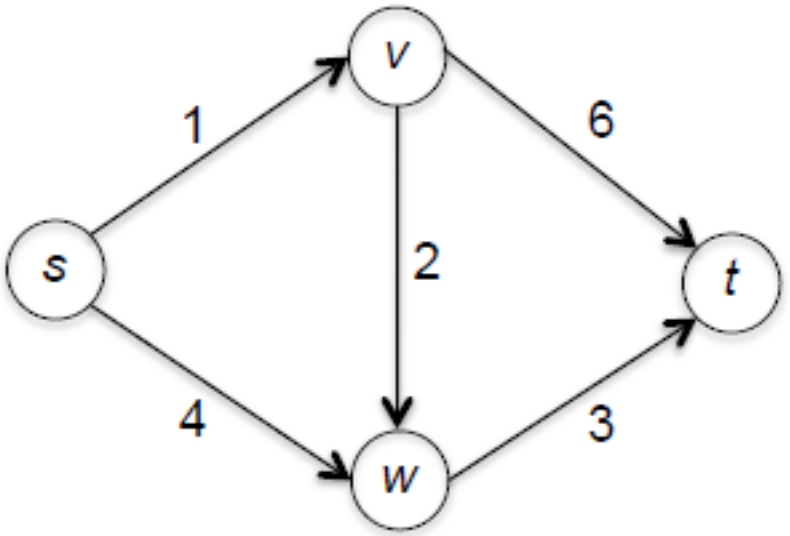
Dijkstra's Algorithm- Pseudocode

- The set X contains the vertices that the algorithm has already dealtwith. Initially, X contains only the starting vertex (and, of co urse, len(s) = 0), and the set grows like a mold until it covers all the vertices reachable from s. The algorithm assigns a finite value to the len-value of a vertex at the same time it adds the vertex to X.
- Each iteration of the main loop augments X by one new vertex, the head of some edge (v,w) crossing from X to V −X . (If there is no such edge, the algorithm halts, with len(v) = +∝ for all v ∉ X.)
- There can be many such edges; the Dijkstra algorithm chooses one (v*, w*) that minimizes the Dijkstra score, which is defined as

$$\text{len}(v) + L_{vw}$$

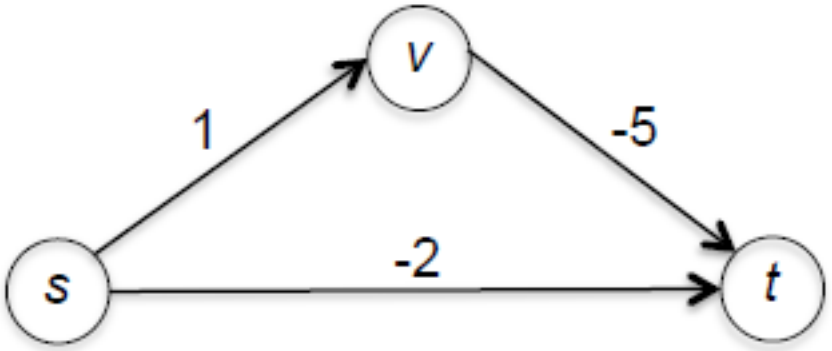
An Example

- Initially, the set X contains only s, and len(s) = 0.
- In the first iteration of the main loop, there are two edges crossing from X to V −X. the edges (s, v) and (s,w)
- The Dijkstra scores for these two edges are len(s)+ℒsv = 0+1 = 1 and len(s)+ℒsw = 0+4 = 4.
- Because the former edge has the lower score, its head v is added to X, and len(v) is assigned to the Dijkstra score of the edge (s, v), which is 1. In the
- second iteration, with X = {s, v}, there are three edges to consider: (s,w), (v,w), and (v, t). Their Dijkstra scores are 0 + 4 = 4, 1 + 2 = 3, and 1 + 6 = 7. Because (v,w) has the lowest Dijkstra score, w gets sucked into X and len(w) is assigned the value 3 ((v,w)'s Dijkstra score).
- As (v, t) and (w, t) have Dijkstra scores 1 + 6 = 7 and 3 + 3 = 6, respectively, len(t) is set to the lower score of 6. The set X now contains all the vertices, so no edges cross from X to V −X and the algorithm halts. The values len(s) = 0, len(v) = 1, len(w) = 3, and len(t) = 6



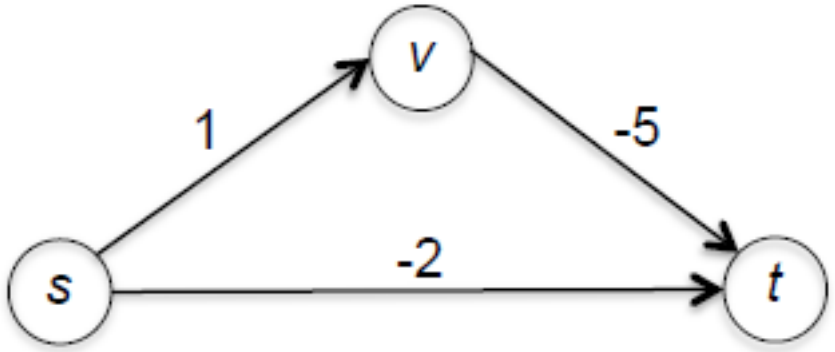
A Bad Example for the Dijkstra Algorithm

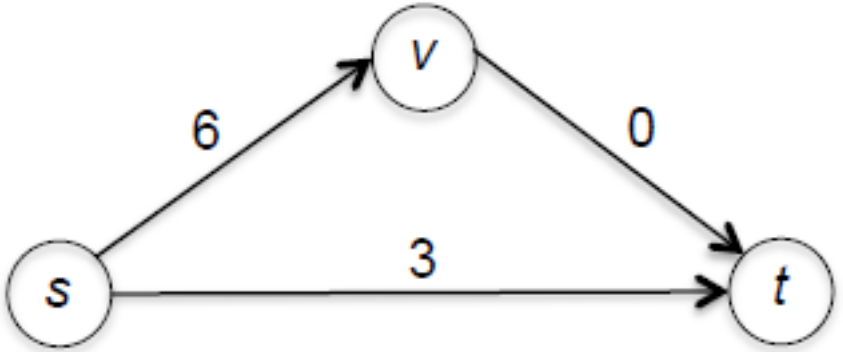
- Shortest pth s→v→t
- if we try running the Dijkstra algorithm directly on a graph with some negative edge lengths.
- initially X = {s} and len(s) = 0, all of which is fine. In the first iteration of the main loop, however, the algorithm computes the Dijkstra scores of the edges (s, v) and (s, t), which are len(s) + L_{sv} = 0 + 1 = 1 and len(s) + L_{st} = 0+(−2) = −2. The latter edge has the smaller score, and so the algorithm adds the vertex t to X and assigns len(t) to the score -2. conclude that
- the Dijkstra algorithm need not compute the correct shortest-path distances in the presence of negative edge lengths.



A Bogus Reduction

- You cannot reduce the single-source shortest path problem with general edge lengths to the special case of nonnegative edge lengths in this way. The problem is that different paths from one vertex to another might not have the same number of edges. If we add some number to the length of each edge, then the lengths of different paths can increase by different amounts, and a shortest path in the new graph might be different than in the original graph. The shortest path s →v→t

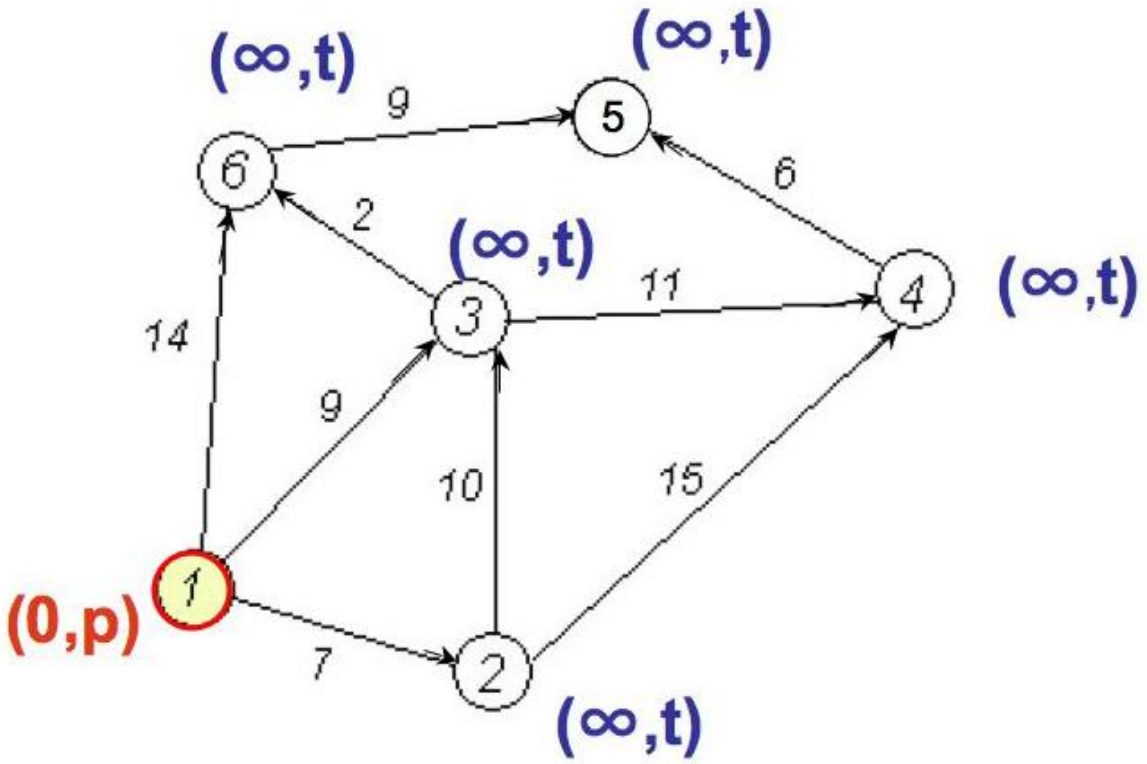




To force the graph to have nonnegative edge lengths, we could add 5 to every edge's length. The shortest path s →t

Another Example(1)

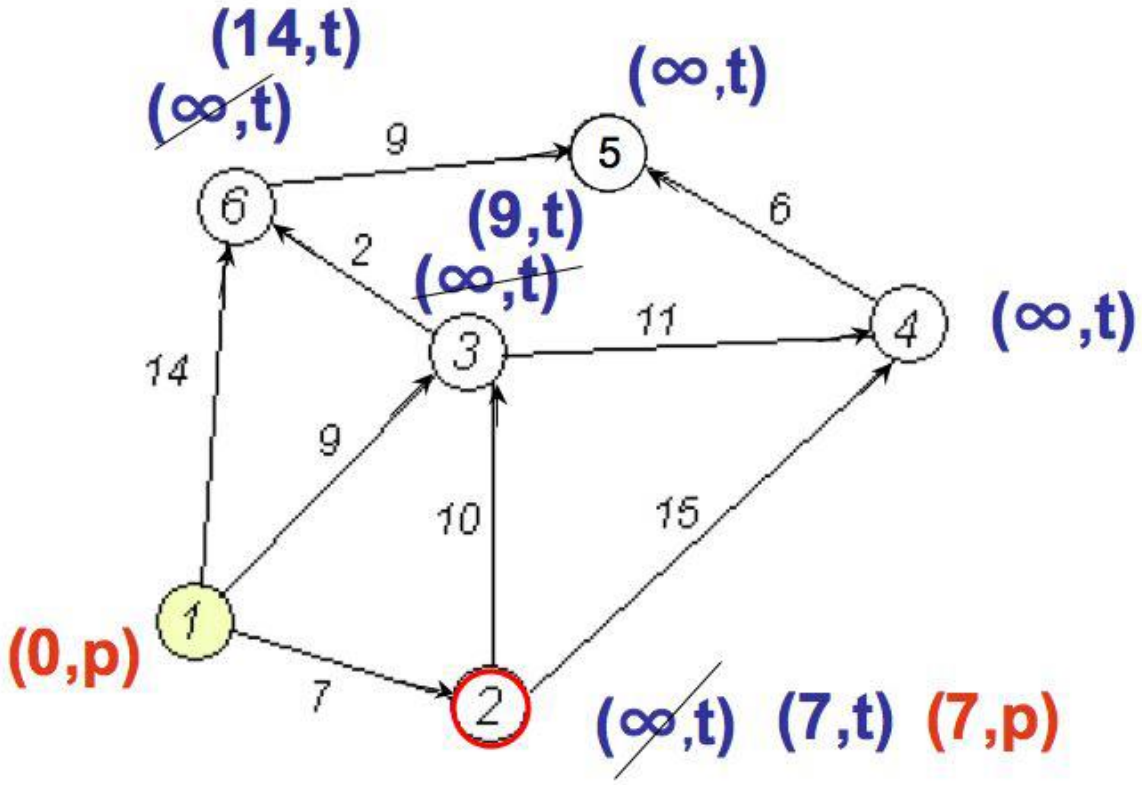
- Use Dijkstra's algorithm to calculate the single-source shortest paths from vertex 1 to every other vertex
- Initially, the set X contains only s, and len(s) = 0.
- Every other node has state ∝



Another Example(2)

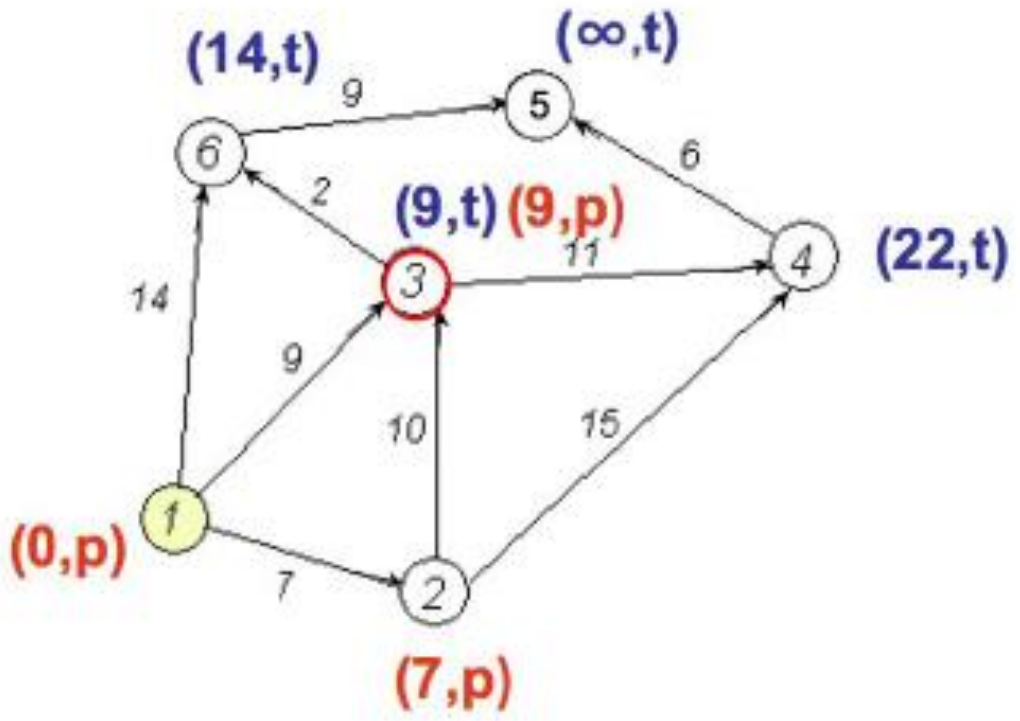
- Nodes 2, 3,and 6 can be reached
- from the current node 1
- d2 = min{1, 0+7} = 7
- d3 = min{1, 0+9} = 9
- d6 = min{1, 0+14} = 14

Now, among the nodes 2, 3, and 6, node 2 has the smallest distancevalue



Another Example(3)

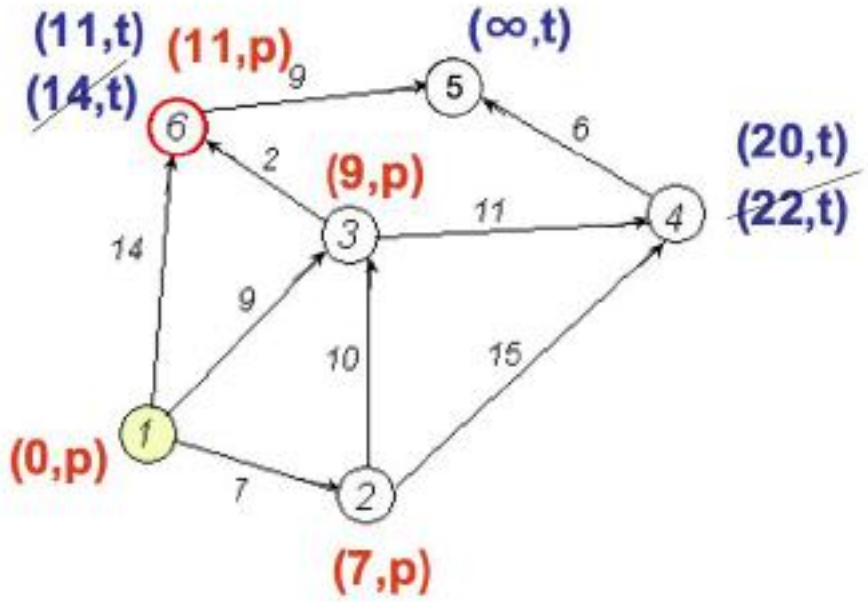
- Nodes 3 and 4 can be reached from the current node 2
- d3 = min{9, 7+10} = 9
- d6 = min{1, 7+15} = 22
- •Now, between the nodes 3 and 4 node 3 has the smallest distance value



Another Example(3)

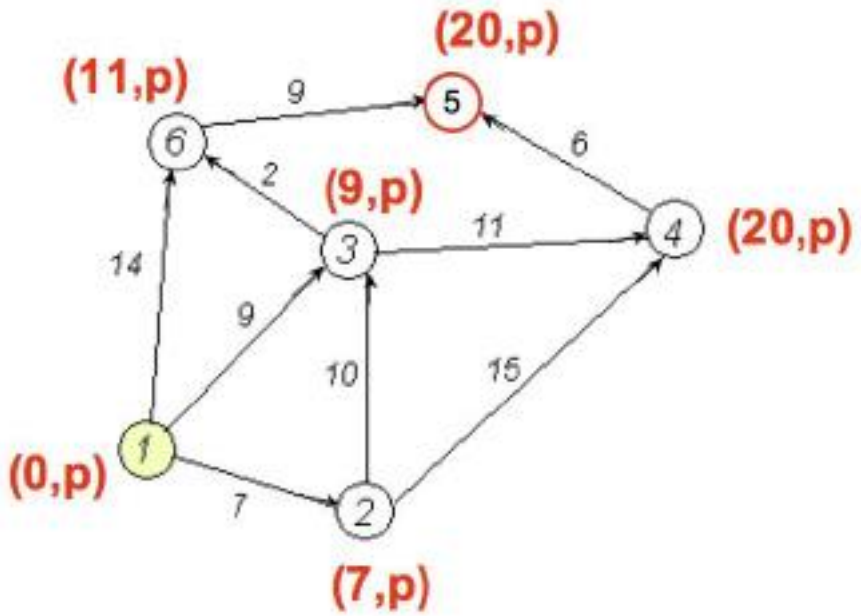
- Nodes 6 and 4 can be reached from the current node 3
- d4 = min{22, 9+11} = 20
- d6 = min{14, 9+2} = 11

Now, between the nodes 6 and 4 node 6 has the smallest distance value

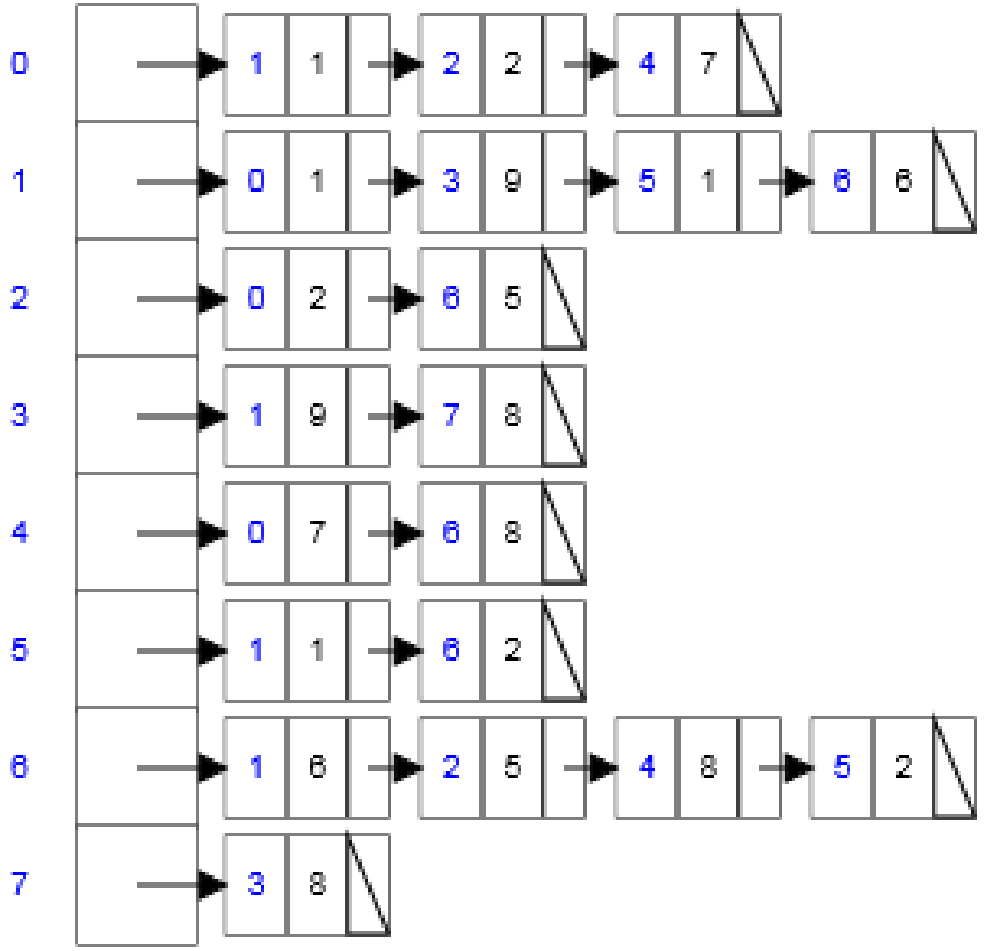


Another Example(4)

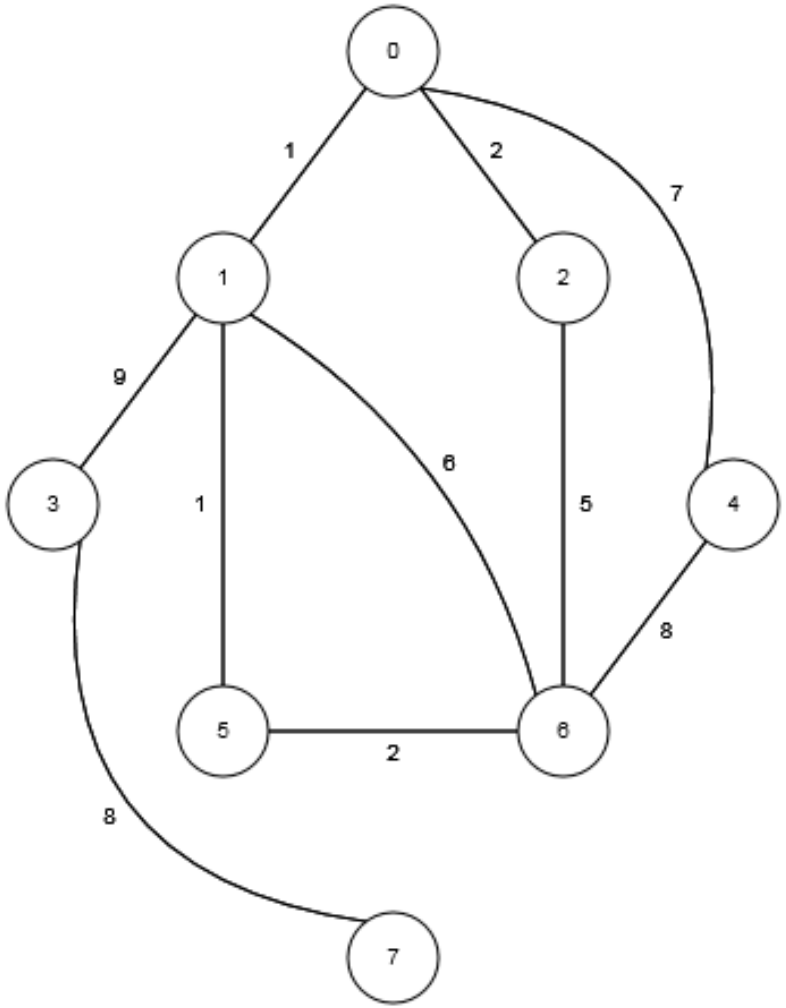
- Node 5 can be reached from the current node 6
- Update distance value for node 5
- d5 = min{1, 11+9} = 20



Visualization of Dijkstra



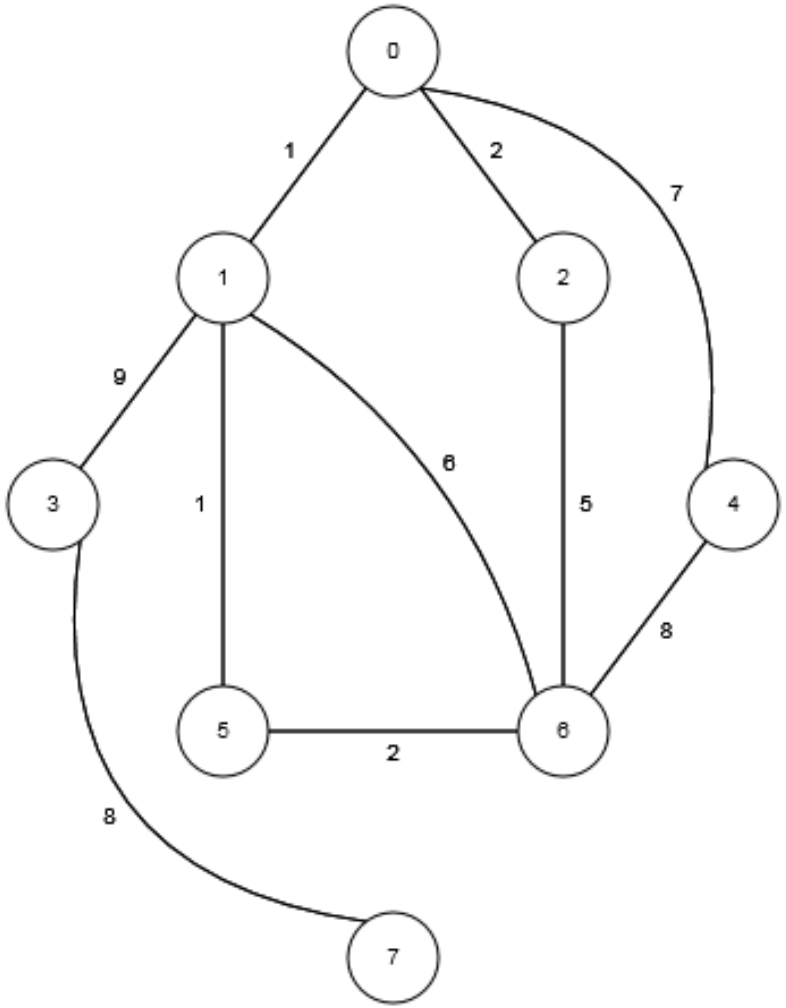
Adjacency List Representation



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

Visualization of Dijkstra

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



Adjacency MatrixRepresentation

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

Running Time of Dijkstra's

- The running times of a straightforward implementation of Dijkstra's algorithm for graphs in adjacency-list representation? As usual, n and m denote the number of vertices and edges, respectively, of the input graph.
- running time is O(mn) because the number of iterations is O(n) and each takes time O(m)