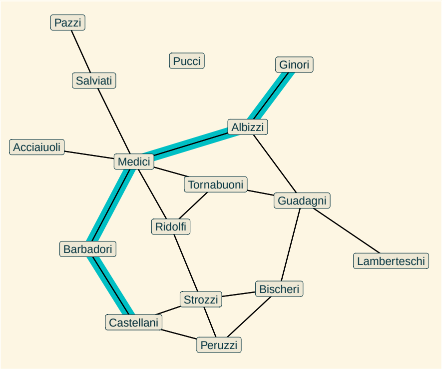
Lesson 01: Graph Representations and Breadth-First Search

Michael T. Gastner (21 February 2023)



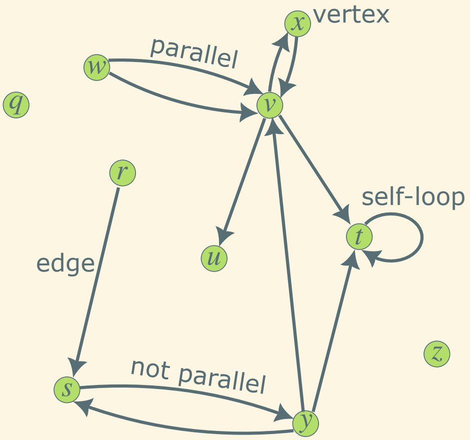
Disclaimer: These slides are based on and occasionally quote from 'Introduction to Algorithms' (3rd ed.) by Cormen et al. (2009), MIT Press.

What is a Graph?
A Mathematical Representation of a Network!

A graph G is a pair (V, E) of two sets:

- V: Set of **vertices** (singular: vertex), also called 'nodes'.
- E: Set of **edges**, also called 'links'.

To emphasise that V and E are attributes of G, we sometimes write V as $G.V$ and E as $G.E$.



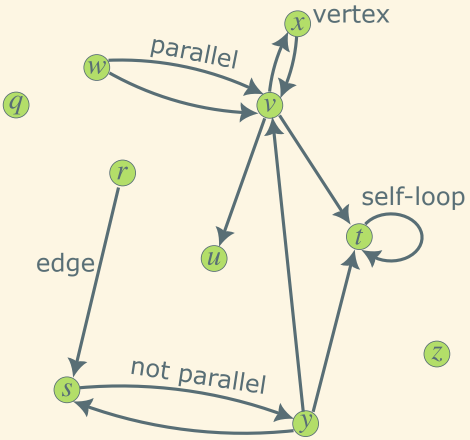
2 / 36

What is a Graph?

A Mathematical Representation of a Network!

In a **directed graph**, also called a 'digraph', the edges are ordered pairs of vertices. That is, $(r, s) \in E$ does not necessarily imply $(s, r) \in E$. In plots, the order is usually indicated by an arrow from r to s.

In an **undirected graph**, the edges are unordered pairs of vertices. That is, $(r, s) = (s, r)$.

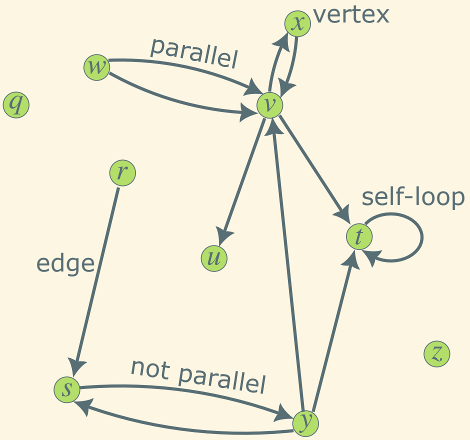


What is a Graph?

A Mathematical Representation of a Network!

A graph is **simple** if it does not contain parallel edges or self-loops.

In this course, we only work with simple graphs. For brevity, the adjective 'simple' will be omitted but is always implicit unless otherwise stated.



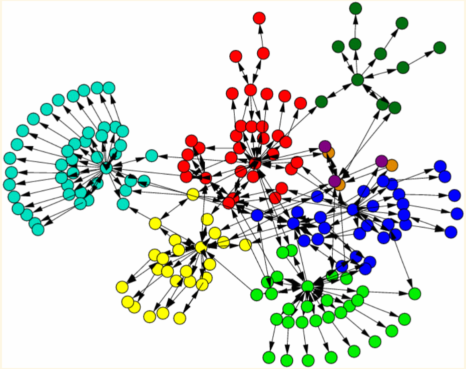
2 / 36

Learning Objectives

By the end of this lesson, you should be able to ...

- Name real-world examples of graphs in different domains (e.g. transportation, social and computer networks).
- Recall important definitions related to graphs (e.g. vertex, edge, directedness and path).
- Construct adjacency-list and adjacency-matrix representations of a given graph.
- Determine the reachability of a vertex v from a vertex u.
- Find the shortest path from u to v if v is reachable from u.

Example
World Wide Web

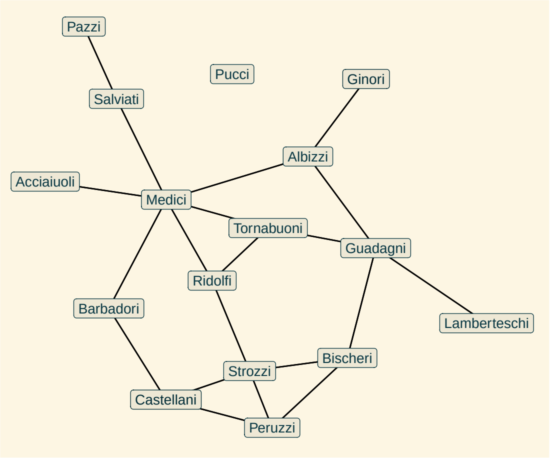


Network of 180 web pages of a large corporation. Colours depict an automatically detected division into communities.

Image by M. E. J. Newman and M. Girvan.
DOI: 10.1103/PhysRevE.69.026113

- Directed graph.
- Vertices are websites.
- Edges are hyperlinks.

Example
Social Network

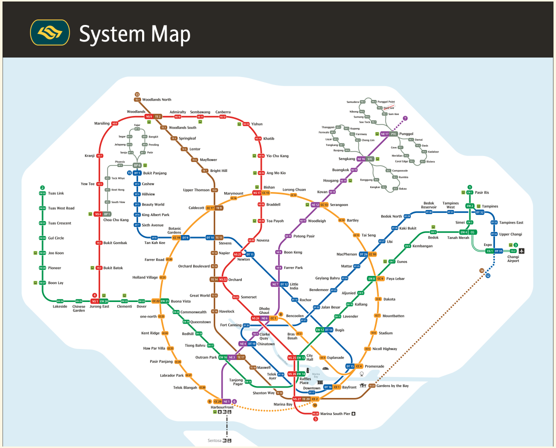


Florentine marriage network

- Undirected graph.
- Vertices are influential families in Renaissance Florence.
- Edges are marriages.

Data from Pagett and Ansell.
DOI: 10.1086/230190

Example
Public Transport



Network of Singapore Mass Rapid Transit. Image from Land Transport Guru.

- Undirected graph.
- Vertices are stations.
- Edges are railway tracks.

Paths

Contiguous Sequences of Vertices

A **path** of **length** k from a vertex u to a vertex u' in a graph $G = (V, E)$ is a sequence $\langle v_0, v_1, \dots, v_k \rangle$ of vertices such that all of the following conditions are satisfied:

- $u = v_0$
- $u' = v_k$
- $(v_{i-1}, v_i) \in E \quad \forall i \in \{1, 2, \dots, k\}$

There is always a path from u to u of length 0 for all $u \in V$.

If there is a path p from u to u', we say that u' is **reachable** from u via p.

A path is **simple** if all vertices in the path are distinct.

7 / 36

Application

Reachability and Shortest Path

Is Dhoby Ghaut reachable from Bencoolen?

If yes, what is the shortest path, measured by the number of visited stations?

How can we program a computer to find the answers?

8 / 36

Maximum Number of Possible Edges
$O(V^2)$

If a simple graph has $|V|$ vertices, what is the maximum possible number of edges?

It depends on whether the graph is directed or undirected:

- In a directed graph, there are $|V|^2 - |V|$ possible edges, which equals the number of all ordered pairs $|V|^2$ of vertices minus the number of self-loops $|V|$.
- In an undirected graph, there are only $\frac{1}{2}(|V|^2 - |V|)$ possible edges because an edge (r, s) always implies that (s, r) is also in E.

However, in both cases, the maximum number of edges is $O(|V|^2)$, which we will write as $O(V^2)$ for the sake of brevity.

9 / 36

Sparse and Dense Graphs

Comparing Maximum to Actual Number of Edges

- A graph is **sparse** if $|E|$ is much less than $|V|^2$.
  Often, a sparse graph is assumed to satisfy $|E| = O(V)$.
- A graph is **dense** if $|E|$ is close to $|V|^2$.
  Often, a dense graph is assumed to satisfy $|E| = O(V^2)$.

Note that $|V|$ and $|E|$ cannot be varied in most real-world networks. They are usually fixed numbers given by the input data.

Thus, statements such as $|E| = O(V)$ need to be taken with a grain of salt because we cannot predict how many edges a network would have if the number of vertices were to change.

10 / 36

Graph Representations

Adjacency List and Adjacency Matrix

We will learn two standard ways to represent a graph $G = (V, E)$:

- Adjacency list
- Adjacency matrix

Rules of thumb:

- The adjacency-list representation tends to be better when a graph is sparse.
- The adjacency-matrix representation tends to be better when a graph is dense.

For a given graph, vertex v is **adjacent** to vertex u if (u, v) is an edge in the graph.

11 / 36

Adjacency List

Array of Lists of Adjacent Vertices

An adjacency list of a graph $G = (V, E)$, is an array Adj of $|V|$ lists with one list for each vertex in V.

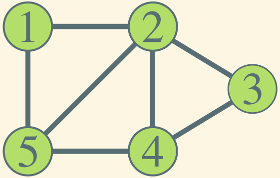
For each $u \in V$, the list $Adj[u]$ contains all the vertices v that are adjacent to u. The order of the elements in $Adj[u]$ does not matter.

To emphasise that Adj is an attribute of G, we sometimes write the adjacency list explicitly as $G.Adj$.

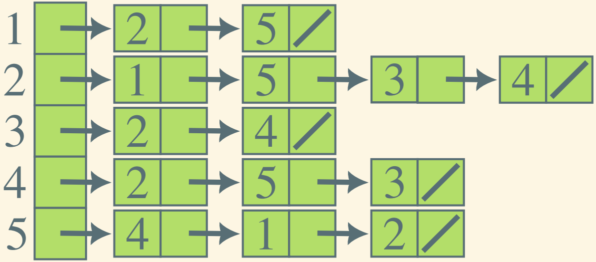
Adjacency List

Example: Undirected Graph

Graph



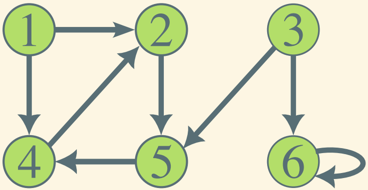
Adjacency list



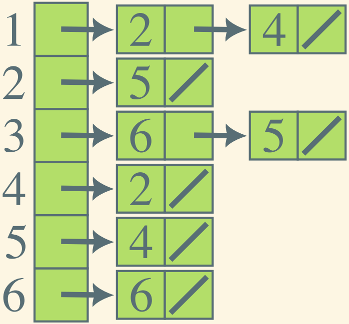
Adjacency List

Example: Directed Graph

Graph (not simple because of self-loop)



Adjacency list



Memory Needed for Adjacency List
$\Theta(V+E)$

What is the sum Σ of the lengths of all sublists in an adjacency list?

- In a directed graph, every edge corresponds to one element in one of the sublists.
  Hence, $\Sigma = |E|$.
- In an undirected graph, every edge $(u, v) \in E$ corresponds to one element in $Adj[u]$ and one element in $Adj[v]$.
  Hence, $\Sigma = 2|E|$.

In addition to Σ, we also must reserve memory for the array of $|V|$ pointers to the sublists. Consequently, an adjacency list requires $\Theta(V+E)$ memory.

- For sparse graphs, $\Theta(V+E) = \Theta(V)$.
- For dense graphs, $\Theta(V+E) = \Theta(V^2)$.

15 / 36

Adjacency Matrix

Binary $|V| \times |V|$ Matrix

The adjacency matrix of a graph $G = (V, E)$ with vertices $V = \{1, 2, \ldots, |V|\}$ is a $|V| \times |V|$ matrix $A = (a_{ij})$ such that

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

G is undirected if and only if A is symmetric; that is,

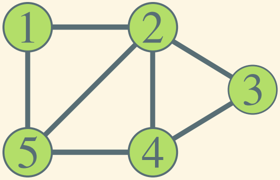
$$a_{ij} = a_{ji} \qquad \forall i, j \in \{1, 2, \ldots, |V|\}.$$

16 / 36

Adjacency Matrix

Example: Undirected Graph

Graph



Adjacency matrix

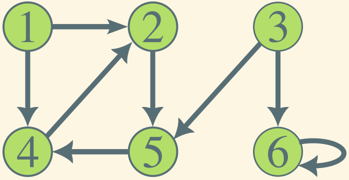
$$
\begin{array}{c|ccccc|}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 0 & 0 & 1 \\
2 & 1 & 0 & 1 & 1 & 1 \\
3 & 0 & 1 & 0 & 1 & 0 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 1 & 1 & 0 & 1 & 0 \\
\hline
\end{array}
$$

17 / 36

Adjacency Matrix

Example: Directed Graph

Graph (not simple because of self-loop)



Adjacency matrix

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

Properties of the Adjacency Matrix

Memory and Usage

The adjacency matrix A requires $\Theta(V^2)$ memory, regardless of the number of edges.

The adjacency-list representation is asymptotically at least as space-efficient as the adjacency-matrix representation, and the difference in required memory is largest for sparse graphs.

Because most real-world networks are sparse, graph algorithms usually assume that the graph is represented by the adjacency list.

19 / 36

Breadth-First Search
Graph Traversal Algorithm

We begin our study of graph algorithms with **breadth-first search** for the following reasons:

- Breadth-first search is one of the simplest algorithms for searching a graph.
- Breadth-first search returns information about the reachability of one node from another.
- If vertex v is reachable from u, breadth-first search computes a **shortest path** (i.e. a path containing the smallest number of edges) from u to v.
- Breadth-first search is a prototype for many important graph algorithms (e.g. Prim's minimum-spanning-tree algorithm and Dijkstra's single-source shortest-paths algorithm).

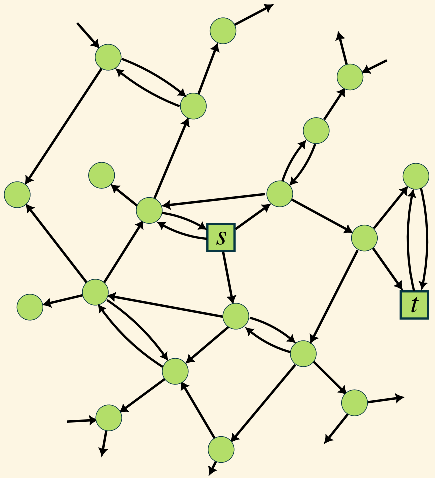
20 / 36

Idea Behind Breadth-First Search

Illustrative Example

Consider graph on the right.

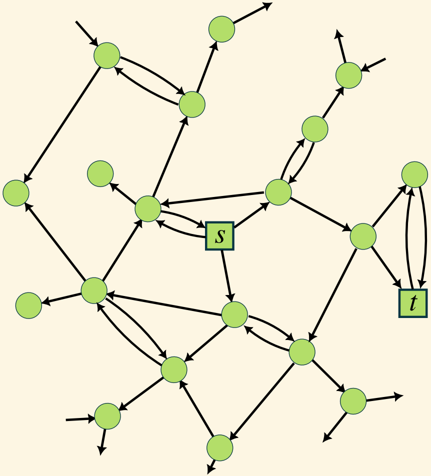
Objective: find **distance** (i.e. length of shortest path) from source vertex s to target vertex t



Idea Behind Breadth-First Search

Illustrative Example

Mark s as 'active' (red).

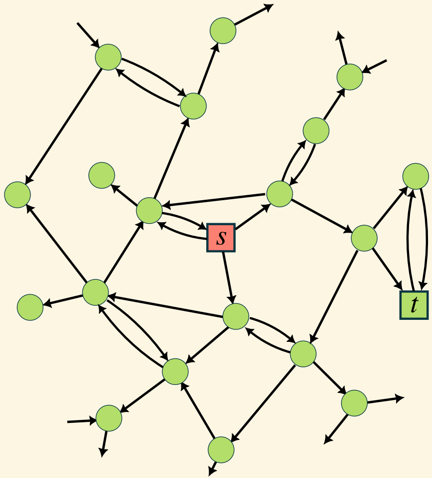


22 / 36

Idea Behind Breadth-First Search

Illustrative Example

Mark s as 'active' (red).



22 / 36

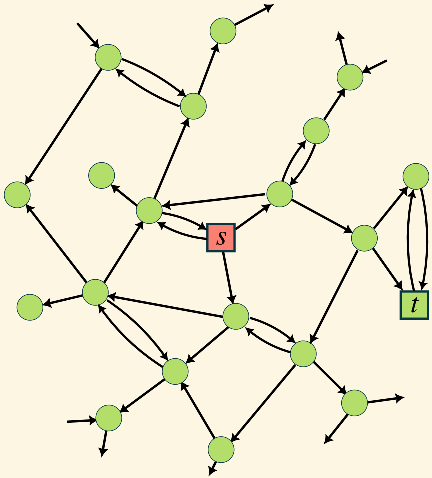
Idea Behind Breadth-First Search

Illustrative Example

It is guaranteed that s has distance 0 from itself.

We also know that no other vertex has distance 0 from s.

Indicate all vertices with distance 0 (i.e. only s) with a dark grey circle.



23 / 36

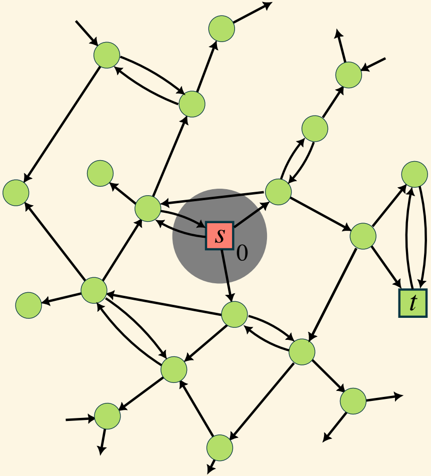
Idea Behind Breadth-First Search

Illustrative Example

It is guaranteed that s has distance 0 from itself.

We also know that no other vertex has distance 0 from s.

Indicate all vertices with distance 0 (i.e. only s) with a dark grey circle.



23 / 36

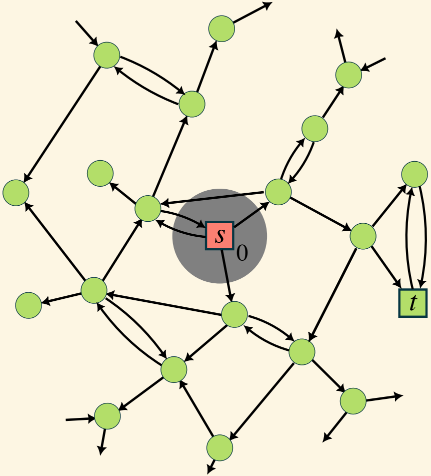
Idea Behind Breadth-First Search

Illustrative Example

Suppose the active vertex s has a virus that can spread to all adjacent vertices.

All adjacent vertices become active (red).

Meanwhile, the source develops immunity and becomes white.



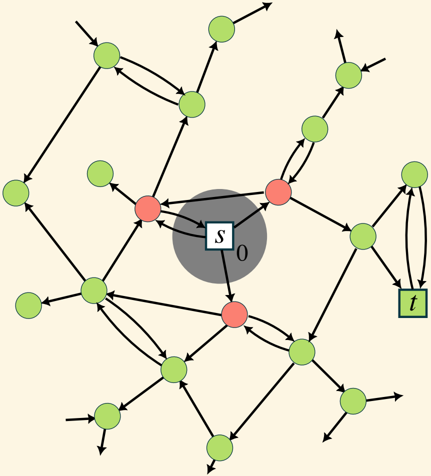
Idea Behind Breadth-First Search

Illustrative Example

Suppose the active vertex s has a virus that can spread to all adjacent vertices.

All adjacent vertices become active (red).

Meanwhile, the source develops immunity and becomes white.

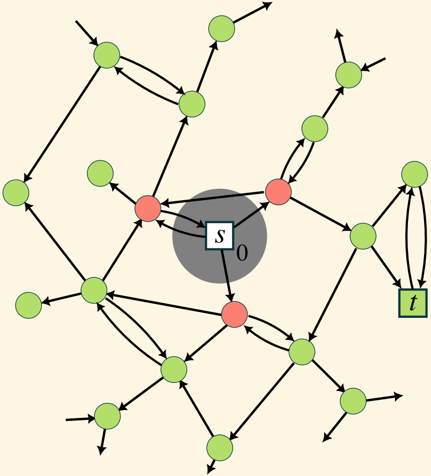


24 / 36

Idea Behind Breadth-First Search

Illustrative Example

Currently active (red) vertices have distance 1 from s because they can be reached from s in one step.

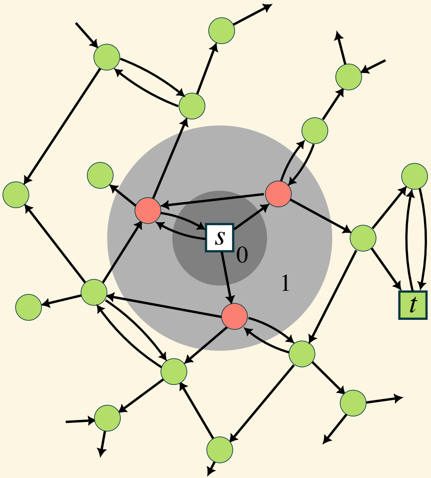


25 / 36

Idea Behind Breadth-First Search

Illustrative Example

Currently active (red) vertices have distance 1 from s because they can be reached from s in one step.



25 / 36

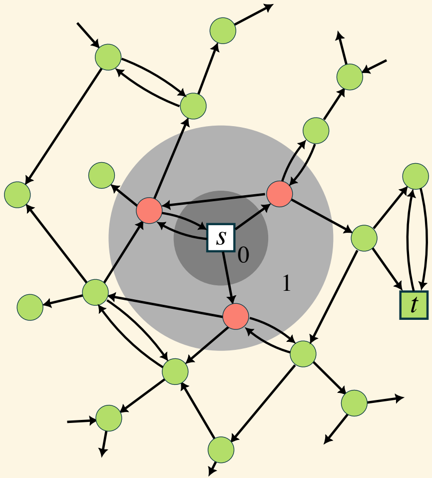
Idea Behind Breadth-First Search
Illustrative Example

Currently active (red) vertices spread the virus to adjacent susceptible (green) vertices.

Adjacent green vertices become active.

The immune (white) vertex s has perpetual immunity and stays white.

Vertices that are currently active also develop immunity.



26 / 36

Idea Behind Breadth-First Search

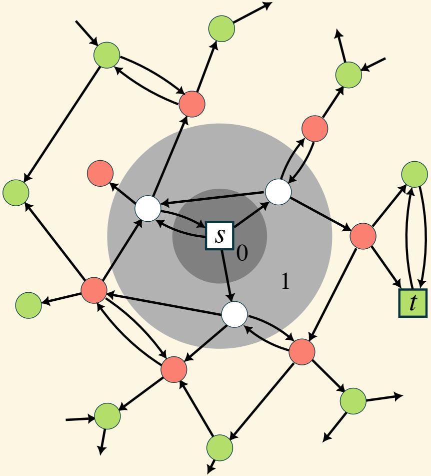
Illustrative Example

Currently active (red) vertices spread the virus to adjacent susceptible (green) vertices.

Adjacent green vertices become active.

The immune (white) vertex s has perpetual immunity and stays white.

Vertices that are currently active also develop immunity.

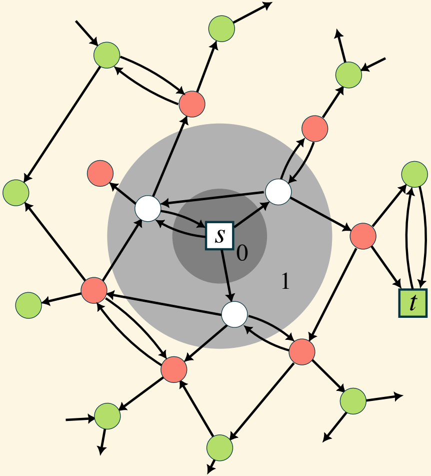


26 / 36

Idea Behind Breadth-First Search
Illustrative Example

Currently active (red) vertices have distance 2 from s.

They cannot have a shorter distance because they were unreachable in zero or one steps.



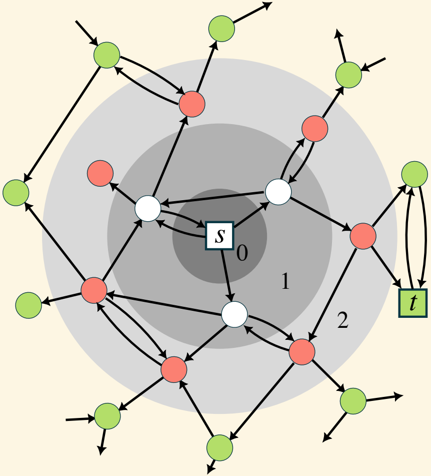
27 / 36

Idea Behind Breadth-First Search

Illustrative Example

Currently active (red) vertices have distance 2 from s.

They cannot have a shorter distance because they were unreachable in zero or one steps.



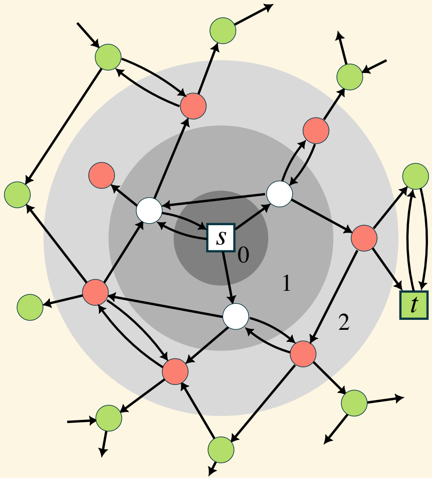
27 / 36

Idea Behind Breadth-First Search

Illustrative Example

Repeat the previous steps:

1. Currently red vertices
   - infect adjacent green vertices, which become red.
   - become immune (white).

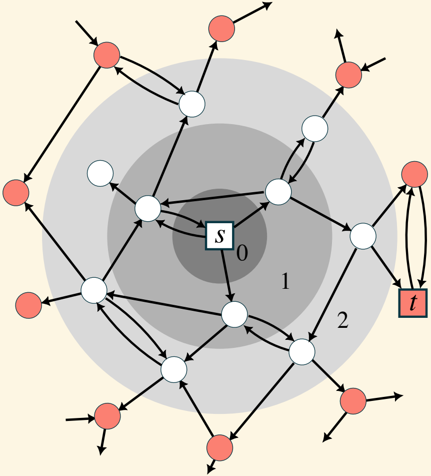


Idea Behind Breadth-First Search

Illustrative Example

Repeat the previous steps:

1. Currently red vertices
   - infect adjacent green vertices, which become red.
   - become immune (white).



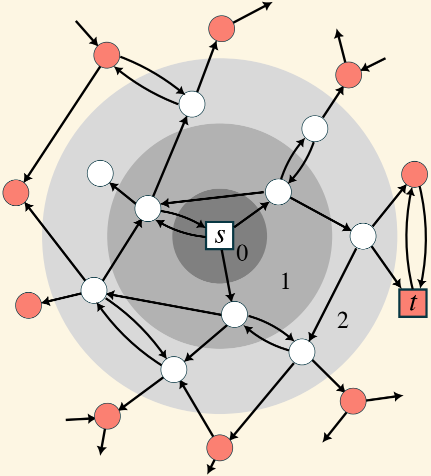
28 / 36

Idea Behind Breadth-First Search

Illustrative Example

Repeat the previous steps:

1. Currently red vertices
   - infect adjacent green vertices, which become red.
   - become immune (white).
2. Distances of new red vertices equals 1 plus the distance of the vertex that infected them.

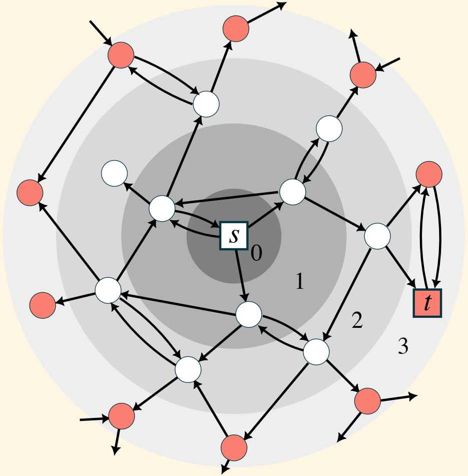


Idea Behind Breadth-First Search

Illustrative Example

Repeat the previous steps:

1. Currently red vertices
   - infect adjacent green vertices, which become red.
   - become immune (white).
2. Distances of new red vertices equals 1 plus the distance of the vertex that infected them.



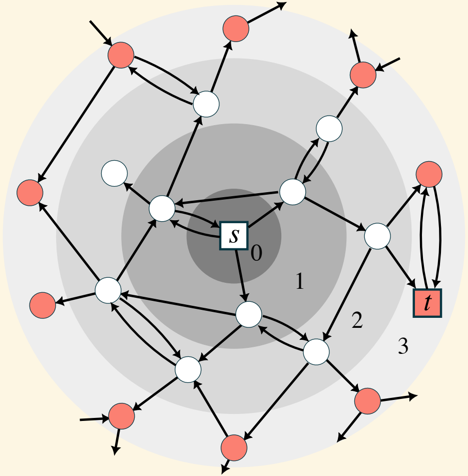
Idea Behind Breadth-First Search

Illustrative Example

Repeat the previous steps:

1. Currently red vertices
   - infect adjacent green vertices, which become red.
   - become immune (white).
2. Distances of new red vertices equals 1 plus the distance of the vertex that infected them.

Stop when assigning distance to target vertex t.



28 / 36

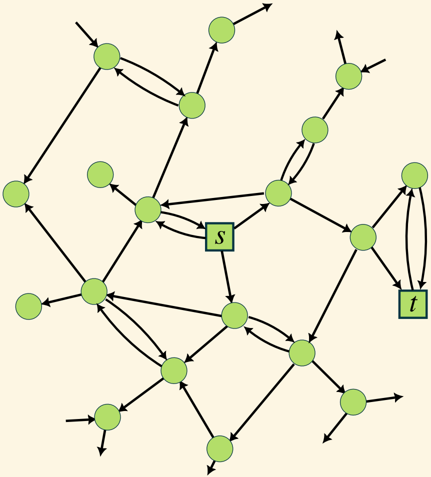
Idea Behind Breadth-First Search

Illustrative Example

In the animation on the right (viewable in Acrobat Reader), note how the red vertices spread from s like burning trees in a forest fire:

- green: healthy tree
- red: tree on fire
- white: tree has burnt down and cannot be reignited

Because of this metaphor, in physics 'breadth-first search' is often called 'burning algorithm'.



Idea Behind Breadth-First Search

Illustrative Example

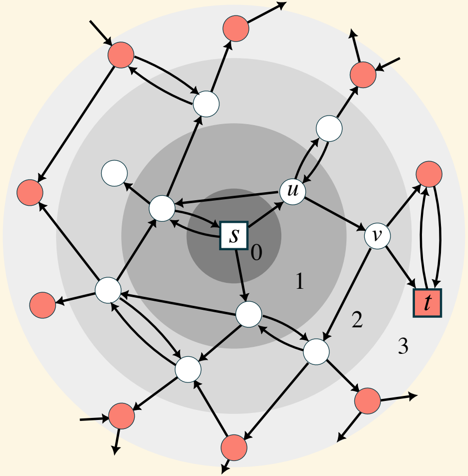
We now know that the distance from s to t is 3.

What if we want to find a path of length 3 (e.g. $s \to u \to v \to t$)?

We would need to determine:

- which vertex v infected t.
- which vertex u infected v.

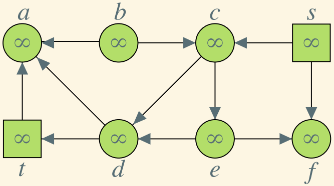
Next, we study a refinement of the algorithm to solve this problem.



Breadth-first search with queue

Assigning parent vertex

Number inside vertex = Estimated distance from s

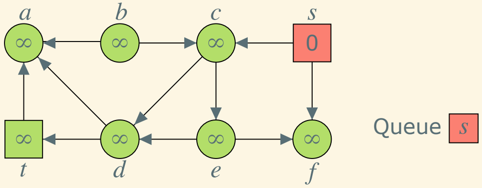


Queue empty

Breadth-first search with queue

Assigning parent vertex

Number inside vertex = Estimated distance from s

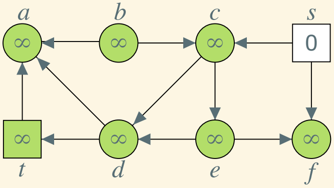


31 / 36

Breadth-first search with queue

Assigning parent vertex

Number inside vertex = Estimated distance from s



Pop first vertex in queue: s

Queue empty

For each green vertex adjacent to popped vertex:

- Push into queue
- Assign distance as 1 plus distance of popped vertex
- Mark popped vertex as 'parent'

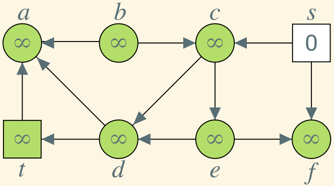
Stop if t has finite distance or queue is empty.
Otherwise, repeat from the top.

31 / 36

Breadth-first search with queue

Assigning parent vertex

Number inside vertex = Estimated distance from s



Pop first vertex in queue: s

Queue empty

For each green vertex adjacent to popped vertex:

- **Push into queue**
- Assign distance as 1 plus distance of popped vertex
- Mark popped vertex as ‘parent’

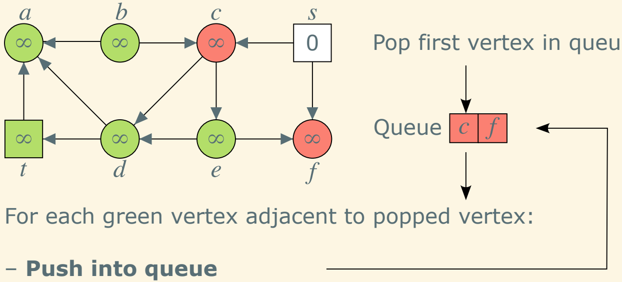
Stop if t has finite distance or queue is empty.
Otherwise, repeat from the top.

31 / 36

Breadth-first search with queue

Assigning parent vertex

Number inside vertex = Estimated distance from s



For each green vertex adjacent to popped vertex:

- **Push into queue**
- Assign distance as 1 plus distance of popped vertex
- Mark popped vertex as ‘parent’

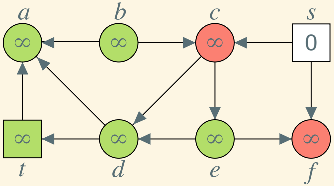
Stop if t has finite distance or queue is empty.
Otherwise, repeat from the top.

31 / 36

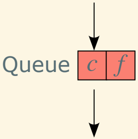
Breadth-first search with queue

Assigning parent vertex

Number inside vertex = Estimated distance from s



Pop first vertex in queue: s



For each green vertex adjacent to popped vertex:

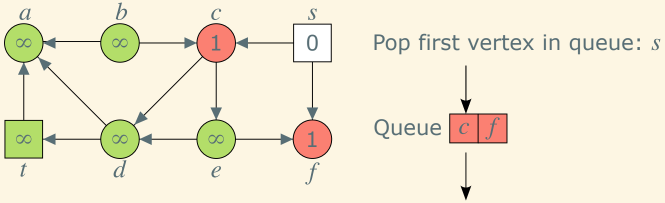
- Push into queue
- **Assign distance as 1 plus distance of popped vertex**
- Mark popped vertex as 'parent'

Stop if t has finite distance or queue is empty.
Otherwise, repeat from the top.

Breadth-first search with queue

Assigning parent vertex

Number inside vertex = Estimated distance from s



For each green vertex adjacent to popped vertex:

- Push into queue
- **Assign distance as 1 plus distance of popped vertex**
- Mark popped vertex as ‘parent’

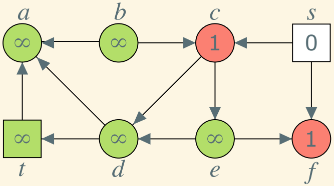
Stop if t has finite distance or queue is empty.
Otherwise, repeat from the top.

31 / 36

Breadth-first search with queue

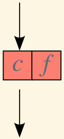
Assigning parent vertex

Number inside vertex = Estimated distance from s



Pop first vertex in queue: s

Queue



For each green vertex adjacent to popped vertex:

- Push into queue
- Assign distance as 1 plus distance of popped vertex
- **Mark popped vertex as 'parent'**

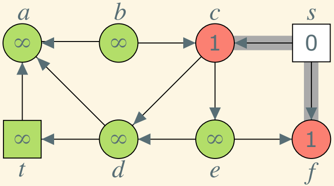
Stop if t has finite distance or queue is empty.
Otherwise, repeat from the top.

31 / 36

Breadth-first search with queue

Assigning parent vertex

Number inside vertex = Estimated distance from s



Pop first vertex in queue: s

Queue c f

For each green vertex adjacent to popped vertex:

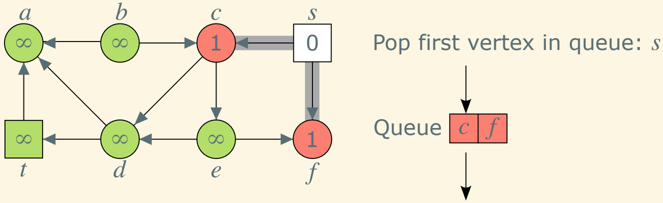
- Push into queue
- Assign distance as 1 plus distance of popped vertex
- **Mark popped vertex as 'parent'**

Stop if t has finite distance or queue is empty.
Otherwise, repeat from the top.

31 / 36

Breadth-first search with queue
Assigning parent vertex

Number inside vertex = Estimated distance from s



For each green vertex adjacent to popped vertex:

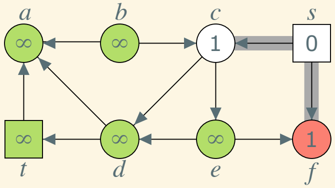
- Push into queue
- Assign distance as 1 plus distance of popped vertex
- Mark popped vertex as ‘parent’

Stop if t has finite distance or queue is empty.
Otherwise, repeat from the top.

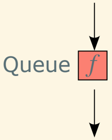
Breadth-first search with queue

Assigning parent vertex

Number inside vertex = Estimated distance from s



Pop first vertex in queue: c



For each green vertex adjacent to popped vertex:

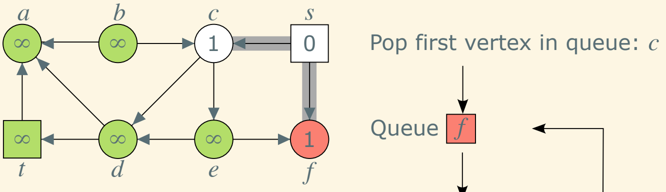
- Push into queue
- Assign distance as 1 plus distance of popped vertex
- Mark popped vertex as 'parent'

Stop if t has finite distance or queue is empty.
Otherwise, repeat from the top.

Breadth-first search with queue

Assigning parent vertex

Number inside vertex = Estimated distance from s



For each green vertex adjacent to popped vertex:

- **Push into queue**
- Assign distance as 1 plus distance of popped vertex
- Mark popped vertex as ‘parent’

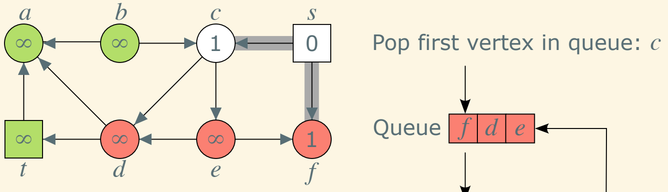
Stop if t has finite distance or queue is empty.
Otherwise, repeat from the top.

31 / 36

Breadth-first search with queue

Assigning parent vertex

Number inside vertex = Estimated distance from s



For each green vertex adjacent to popped vertex:

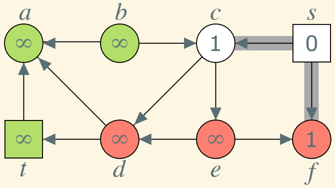
- **Push into queue**
- Assign distance as 1 plus distance of popped vertex
- Mark popped vertex as ‘parent’

Stop if t has finite distance or queue is empty.
Otherwise, repeat from the top.

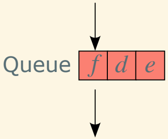
31 / 36

Breadth-first search with queue
Assigning parent vertex

Number inside vertex = Estimated distance from s



Pop first vertex in queue: c



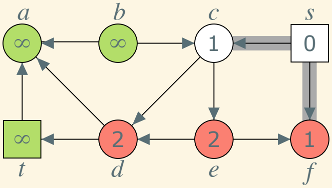
For each green vertex adjacent to popped vertex:

- Push into queue
- **Assign distance as 1 plus distance of popped vertex**
- Mark popped vertex as 'parent'

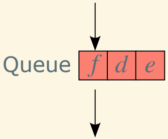
Stop if t has finite distance or queue is empty.
Otherwise, repeat from the top.

Breadth-first search with queue
Assigning parent vertex

Number inside vertex = Estimated distance from s



Pop first vertex in queue: c



For each green vertex adjacent to popped vertex:

- Push into queue
- **Assign distance as 1 plus distance of popped vertex**
- Mark popped vertex as 'parent'

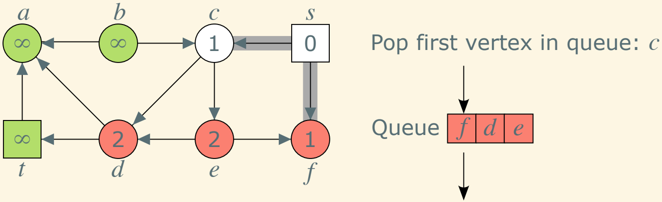
Stop if t has finite distance or queue is empty.
Otherwise, repeat from the top.

31 / 36

Breadth-first search with queue

Assigning parent vertex

Number inside vertex = Estimated distance from s



For each green vertex adjacent to popped vertex:

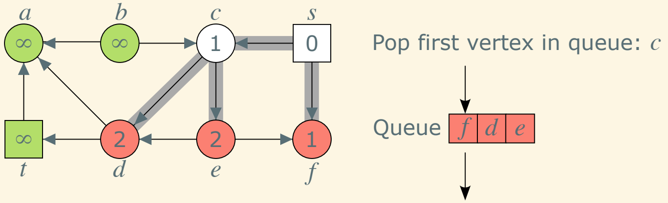
– Push into queue
– Assign distance as 1 plus distance of popped vertex
– **Mark popped vertex as 'parent'**

Stop if t has finite distance or queue is empty.
Otherwise, repeat from the top.

Breadth-first search with queue

Assigning parent vertex

Number inside vertex = Estimated distance from s



For each green vertex adjacent to popped vertex:

- Push into queue
- Assign distance as 1 plus distance of popped vertex
- **Mark popped vertex as 'parent'**

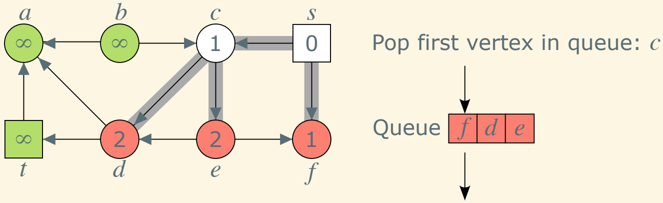
Stop if t has finite distance or queue is empty.
Otherwise, repeat from the top.

31 / 36

Breadth-first search with queue

Assigning parent vertex

Number inside vertex = Estimated distance from s



For each green vertex adjacent to popped vertex:

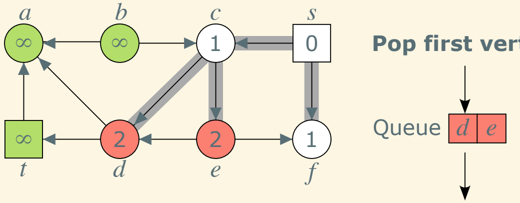
- Push into queue
- Assign distance as 1 plus distance of popped vertex
- Mark popped vertex as 'parent'

Stop if t has finite distance or queue is empty. Otherwise, repeat from the top.

Breadth-first search with queue

Assigning parent vertex

Number inside vertex = Estimated distance from s



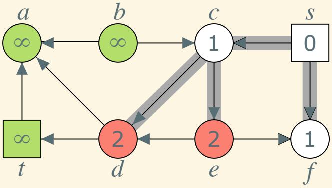
For each green vertex adjacent to popped vertex:

- Push into queue
- Assign distance as 1 plus distance of popped vertex
- Mark popped vertex as 'parent'

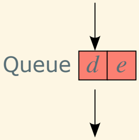
Stop if t has finite distance or queue is empty.
Otherwise, repeat from the top.

Breadth-first search with queue
Assigning parent vertex

Number inside vertex = Estimated distance from s



Pop first vertex in queue: f



For each green vertex adjacent to popped vertex:

- Push into queue
- Assign distance as 1 plus distance of popped vertex
- Mark popped vertex as 'parent'

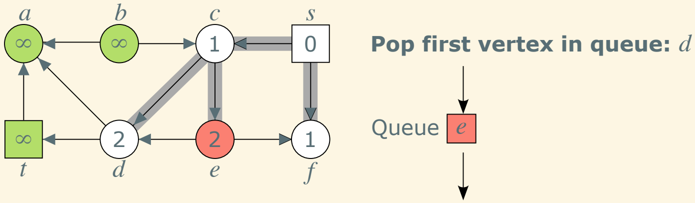
Stop if t has finite distance or queue is empty. Otherwise, repeat from the top.

31 / 36

Breadth-first search with queue

Assigning parent vertex

Number inside vertex = Estimated distance from s



For each green vertex adjacent to popped vertex:

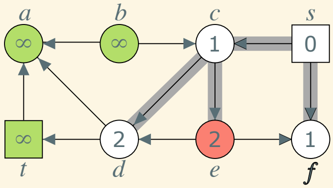
- Push into queue
- Assign distance as 1 plus distance of popped vertex
- Mark popped vertex as ‘parent’

Stop if t has finite distance or queue is empty.
Otherwise, repeat from the top.

31 / 36

Breadth-first search with queue
Assigning parent vertex

Number inside vertex = Estimated distance from s



Pop first vertex in queue: d

Queue e

For each green vertex adjacent to popped vertex:

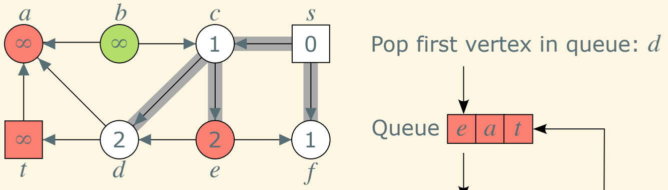
- **Push into queue**
- Assign distance as 1 plus distance of popped vertex
- Mark popped vertex as ‘parent’

Stop if t has finite distance or queue is empty.
Otherwise, repeat from the top.

Breadth-first search with queue

Assigning parent vertex

Number inside vertex = Estimated distance from s



For each green vertex adjacent to popped vertex:

- **Push into queue**
- Assign distance as 1 plus distance of popped vertex
- Mark popped vertex as ‘parent’

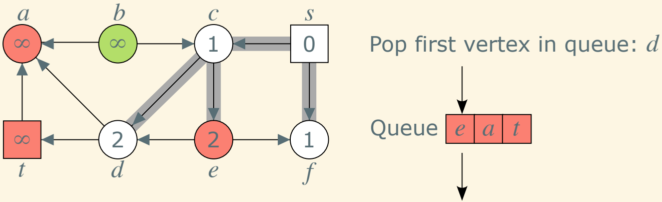
Stop if t has finite distance or queue is empty.
Otherwise, repeat from the top.

31 / 36

Breadth-first search with queue

Assigning parent vertex

Number inside vertex = Estimated distance from s



For each green vertex adjacent to popped vertex:

- Push into queue
- **Assign distance as 1 plus distance of popped vertex**
- Mark popped vertex as 'parent'

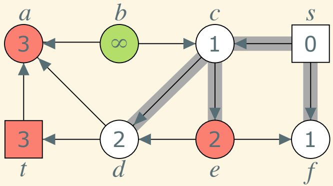
Stop if t has finite distance or queue is empty.
Otherwise, repeat from the top.

31 / 36

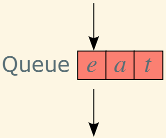
Breadth-first search with queue

Assigning parent vertex

Number inside vertex = Estimated distance from s



Pop first vertex in queue: d



For each green vertex adjacent to popped vertex:

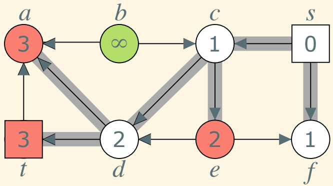
- Push into queue
- Assign distance as 1 plus distance of popped vertex
- **Mark popped vertex as 'parent'**

Stop if t has finite distance or queue is empty.
Otherwise, repeat from the top.

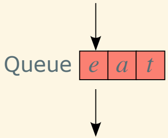
Breadth-first search with queue

Assigning parent vertex

Number inside vertex = Estimated distance from s



Pop first vertex in queue: d



For each green vertex adjacent to popped vertex:

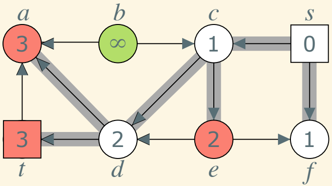
- Push into queue
- Assign distance as 1 plus distance of popped vertex
- **Mark popped vertex as 'parent'**

Stop if t has finite distance or queue is empty. Otherwise, repeat from the top.

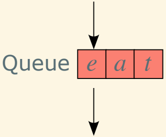
Breadth-first search with queue

Assigning parent vertex

Number inside vertex = Estimated distance from s



Pop first vertex in queue: d



For each green vertex adjacent to popped vertex:

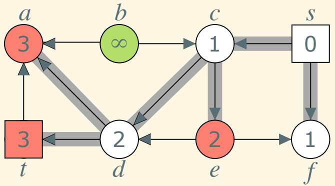
- Push into queue
- Assign distance as 1 plus distance of popped vertex
- Mark popped vertex as 'parent'

Stop if t has finite distance or queue is empty.
Otherwise, repeat from the top.

Breadth-first search with queue

Assigning parent vertex

Number inside vertex = Estimated distance from s



Conclusion:

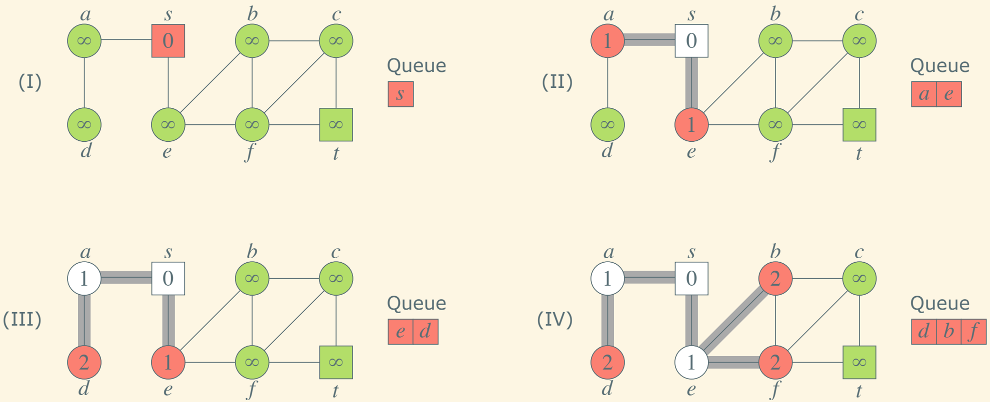
The distance from s to t equals 3.

A shortest path is $s \to c \to d \to t$.

31 / 36

Breadth-first search on an undirected graph

Example

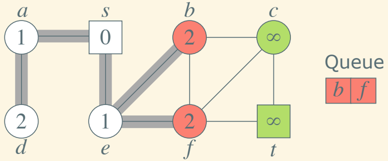


32 / 36

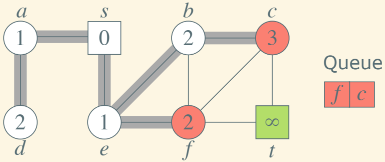
Breadth-first search on an undirected graph

Example

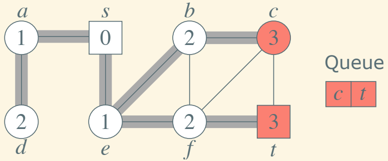
(V)



(VI)



(VII)



Conclusion:
The distance from s to t equals 3.
A shortest path is $s \rightarrow e \rightarrow f \rightarrow t$.

Breadth-first search with queue

Pseudocode

$\text{BFS}(G, s, t)$

for each vertex $u \in G.V - \{s\}$ // Initially all vertices are undiscovered.
    $u.d = \infty$ // Distance.
    $u.\pi = \text{NIL}$ // Parent.
$s.d = 0$ // Discover the source.
$s.\pi = \text{NIL}$
$Q = \emptyset$
$\text{Enqueue}(Q, s)$
while $t.d == \infty$ and $Q \neq \emptyset$
    $u = \text{Dequeue}(Q)$
    for each $v \in G.Adj[u]$ // Iterate over adjacent vertices.
        if $v.d == \infty$ // v has not been discovered yet.
            $v.d = u.d + 1$ // Discover v.
            $v.\pi = u$
            $\text{Enqueue}(Q, v)$

Runtime of breadth-first search

Degree and in-degree of a vertex

Let k_u be the number of vertices pointing away from u.

- In undirected graphs, k_u is called the **degree** of u. Because every edge (u, v) increases the number of elements in the adjacency list by two (once in the sublist for u and once in the sublist for v),

$$\sum_{u \in V} k_u = 2|E|.$$

- In directed graphs, k_u is called the **out-degree** of u. Because every edge (u, v) only increases the number of elements in the adjacency sublist of u,

$$\sum_{u \in V} k_u = |E|.$$

Runtime of Breadth-First Search

$O(V + E)$

The worst-case runtime of breadth-first search is determined by the `while`-loop (lines 8–14).

- During the lifetime of the program, a maximum of $|V|$ vertices are pushed into the queue. Thus, the queue operations need $O(V)$ time.
- The inner `for`-loop is repeated k_u times in one iteration of the `while`-loop. From the previous slide, we know that $\sum_u k_u = O(E)$.

Therefore, breadth-first search runs in $O(V + E)$ time.

35 / 36

Outlook and Conclusion

This lesson: breadth-first search. Next lesson: depth-first search

In this lesson, we learnt that breadth-first search is a graph traversal algorithm that calculates the distances from a source vertex s to a target vertex t. If breadth-first search returns a distance $t.d = \infty$, then t is not reachable from s. Otherwise, breadth-first search also returns a shortest path from s to t.

If we are only interested in reachability and shortest paths (measured in the number of intermediate edges), breadth-first search is a good method. However, for other tasks, other graph traversal algorithms are more suitable. Next time, we learn one such algorithm: **depth-first search**.