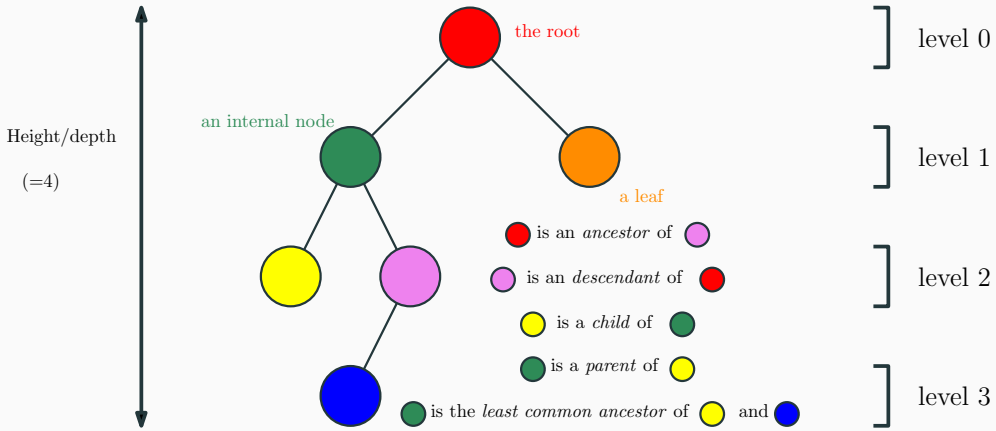
CSCM12: software concepts and efficiency
End of trees & Graphs

Cécilia Pradic

April 18th 2024

slido.com, code #1426271

Last episode: trees



$$\text{depth} \leq \text{size} \leq \max(\mathit{arity})^{\text{depth}}$$

- **breadth-first enumeration:**
- depth-first prefix enumeration:
- depth-first postfix enumeration:
- depth-first infix enumeration:

(← only makes sense for *binary* trees)

Last episode: trees, implementation and an application

- Implementation via recursive classes
- Balanced binary search trees for sets with $\mathcal{O}(\log(n))$ operations

Today

Efficient **priority queues** using trees represented via **arrays**.

Priority queues, heaps and heapsort

Quick note

Motivation

Implement a priority queue with $\mathcal{O}(\log(n))$ operations

$+ \rightarrow$ a new in-place sorting algorithm in $\mathcal{O}(n \log(n))$

The two operations supported by a priority queue

```
void enqueue(T e, int priority);
T dequeue();
```

This material is explained in some details in Lab 5!

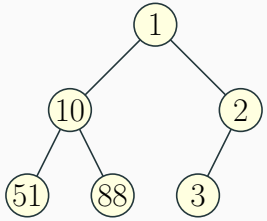
- The challenge task was marked as challenge because it's about trees and we had not covered that back then

What's a heap?

Definition

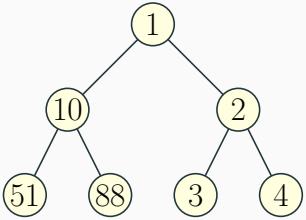
A min-heap is a binary tree such that

- The label of every node is smaller than its children's
- All of its levels are full, except possibly the last
- The last level is completely filled left-to-right

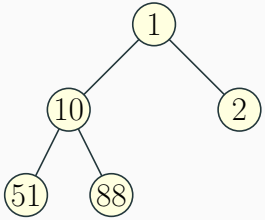


(for priority queues: numbers are priorities + extra label type `T` in nodes)

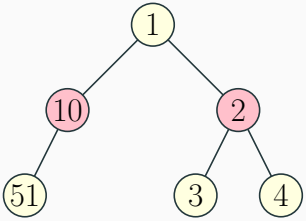
Examples/counter-examples



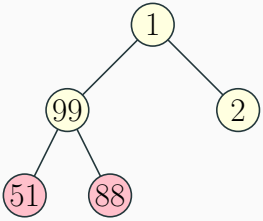
↑ valid heap



↑ valid heap

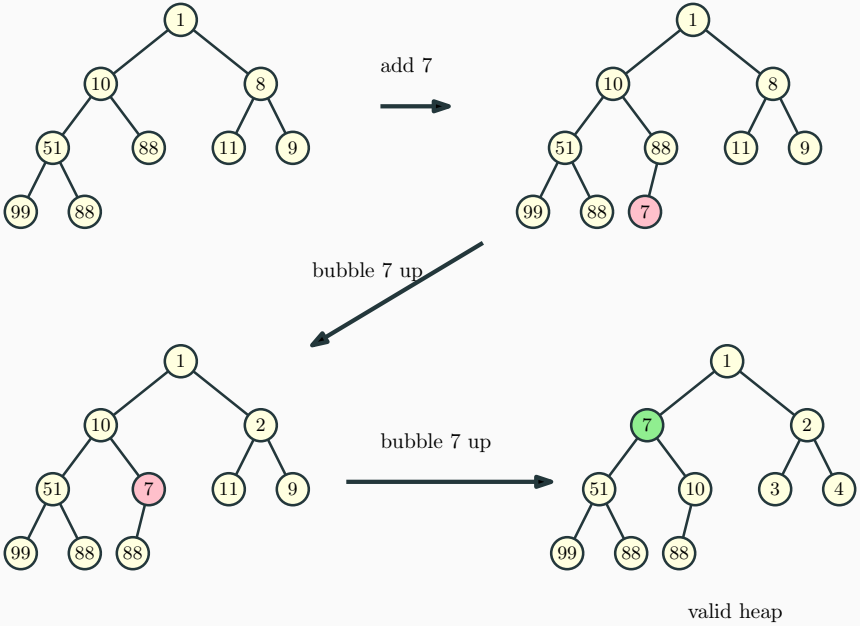


↑ not a heap (unbalanced)

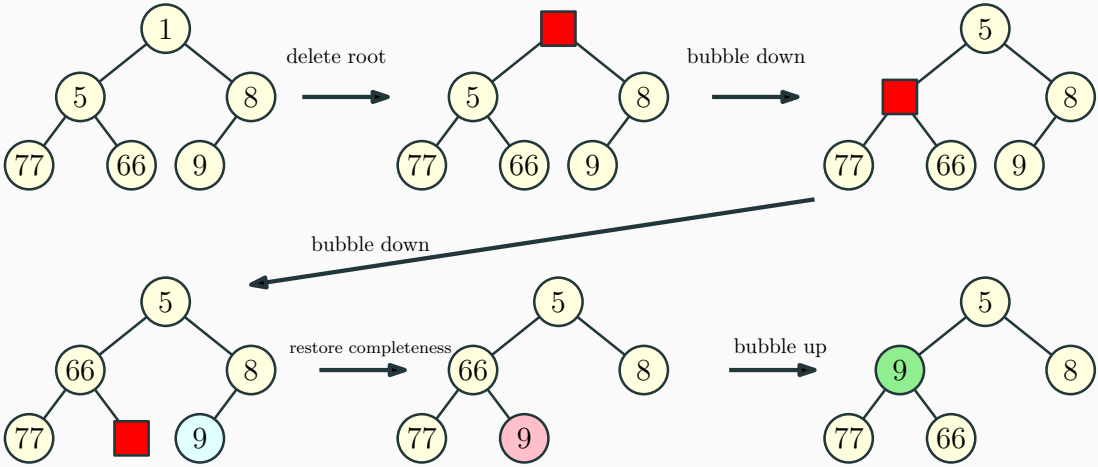


↑ not a heap ($99 > 88$)

Inserting a new element and repairing in $\mathcal{O}(\log(n))$

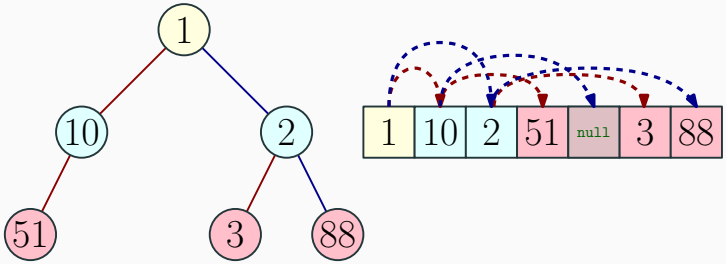


Deleting the root and repairing in $\mathcal{O}(\log(n))$



Representing trees as arrays

While the shape of a tree is good ot keep in mind, when they are of bounded arity and close to complete, it might be better to represent them as arrays



- Fast access due to $\mathcal{O}(1)$ lookup in arrays
- Downsides: *potentially* **wasting memory** and bounding a priori arities

(absent nodes = cells filled with `null`)

For heaps: that's a good representation!

Heap sort

The algorithm

- start with an empty heap
- insert all the elements in the collection you want sorted

$$\sum_{i=1}^{n} K \log(i) + K' = \mathcal{O}(n \log(n))$$

- insert the value of the root at the back of your output and delete the root

$$\sum_{i=1}^{n} K'' \log(i) + K''' = \mathcal{O}(n \log(n))$$

Heap sort

The algorithm

- start with an empty heap
- insert all the elements in the collection you want sorted

$$\sum_{i=1}^{n} K \log(i) + K' = \mathcal{O}(n \log(n))$$

- insert the value of the root at the back of your output and delete the root

$$\sum_{i=1}^{n} K'' \log(i) + K''' = \mathcal{O}(n \log(n))$$

- Optimal asymptotic complexity for a comparison-based sort!
- Can be done *in-place* in an array wiht minor adjustement

$\mathcal{O}(n)$ space complexity

Quick recap on sorting algorithm over arrays (1/2)

Bubble sort

- $\mathcal{O}(n^2)$
- In-place

Quick sort

- $\mathcal{O}(n^2)$, $\mathcal{O}(n \log(n))$ on average with randomized pivot
- Easily done in-place for arrays
- $\mathcal{O}(n \log(n))$ with a smart pivot, but this breaks the in-place aspect of the algo.

Quick recap on sorting algorithm over arrays (1/2)

Merge sort

- $\mathcal{O}(n \log(n))$, good for parallelization
- Not in-place for arrays
- A *stable* sort (does not disturb elements that are "equal")

Heap sort

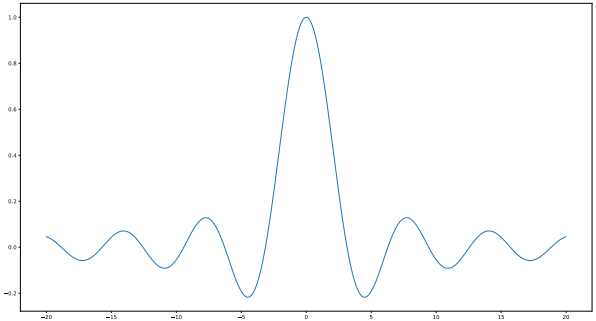
- $\mathcal{O}(n \log(n))$
- In-place!

CountSort

- Not a comparison-based sort, can run in linear time **if working with numbers in a restricted range**.

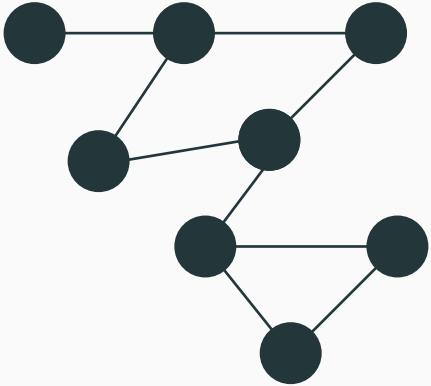
Now for something else: graphs!

Not this kind of graphs



So what is a graph?

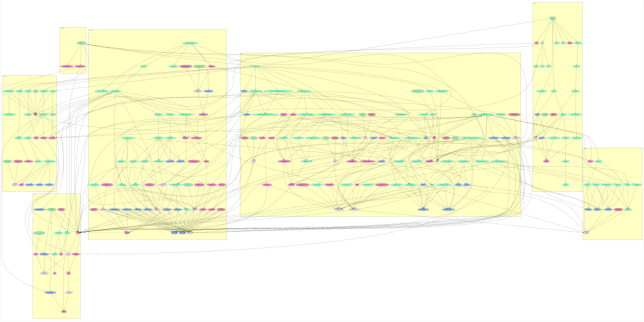
A bunch of **vertices** and **edges** between them.



Graphs in the wild

Dependency graphs

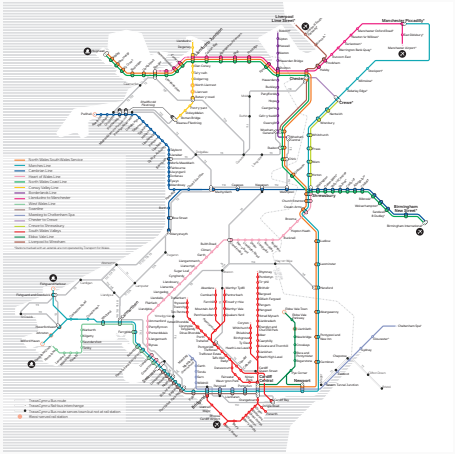
- Nodes are application/libraries
- Edge when a library is required by another



Graphs in the wild

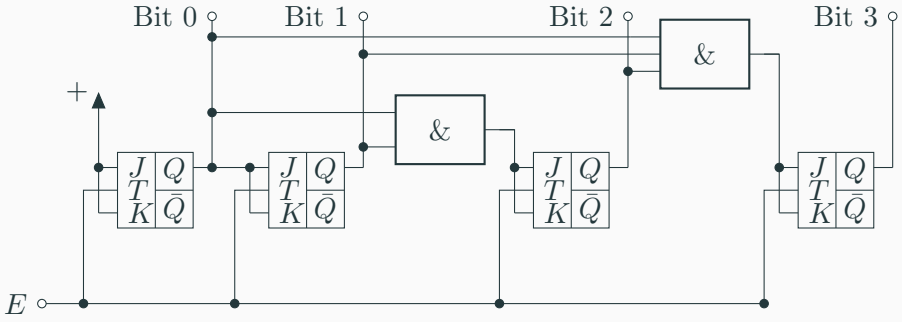
Abstraction of rail maps

- vertices are stations
- edges are routes



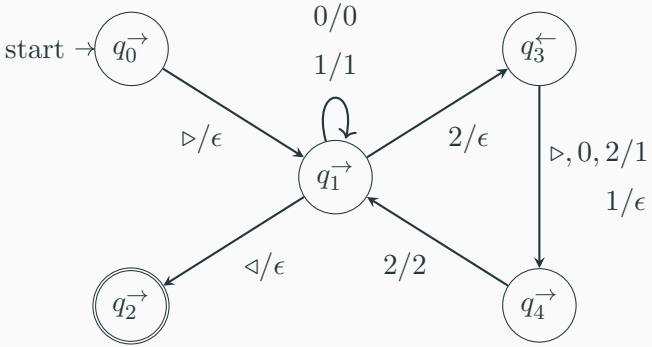
Graphs in the wild

Electrical/electronic circuits



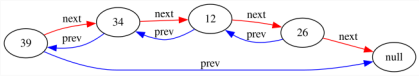
Graphs in the wild

Finite-state machines (useful in CPU design but also **parsing**)



Graphs in the wild

The sort of pictures I used to illustrate datastructures




```
digraph {
  rankdir=LR;
  "39" -> "34" [label = next, color = red];
  "39" -> "null" [label = prev, color = blue];
  "34" -> "12" [label = next, color = red];
  "34" -> "39" [label = prev, color = blue];
  "12" -> "26" [label = next, color = red];
  "12" -> "34" [label = prev, color = blue];
  "26" -> "null" [label = next, color = red];
  "26" -> "12" [label = prev, color = blue];
}
```

(picture generated by the code above using `graphviz`)

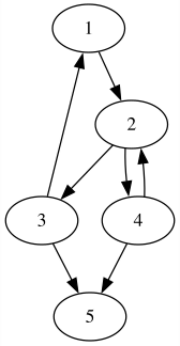
Mathematical definition

Definition

A graph G is given by a pair (V, E) where

- V is a set of **vertices**
- $E \subseteq V^2$ is a set of **edges**

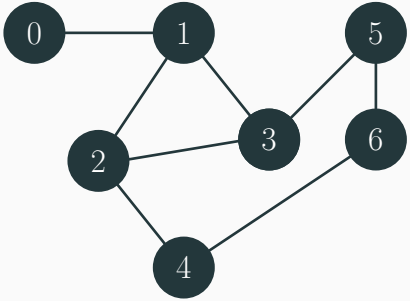
Example: $V = \{1, 2, 3, 4, 5\}, E = \{(1, 2), (2, 3), (2, 4), (3, 1), (3, 5), (4, 2), (4, 5)\}$

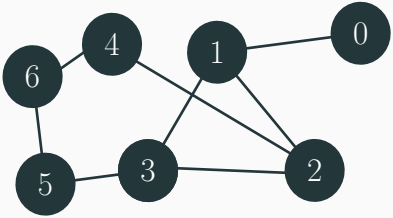


Beware of pictures!

The way the pictures are drawn do not typically matter from a formal standpoint.

I.e., the two examples below picture **the same** graph.

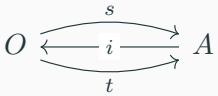




There are many variations (depending on applications)

- Do we allow **multiple** edges between two vertices?

  (people *sometimes* say **multigraphs** in this case)



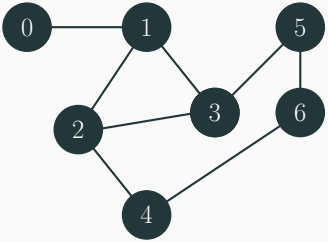
- Do the edge carry a direction information? (**directed** vs **undirected** graphs)

$$v \longrightarrow v' \qquad \text{vs} \qquad v \longrightarrow v'$$

- Do we allow **self-loops**? (rather rare)

- Are the edges/nodes labelled by data? (integer-labelled edges = **weighted** graphs)

Terminology



- A **path** is a sequence of **compatible edges**
  - in non-multigraphs: $\cong$ a sequence of **linked nodes**
  - example: $[(1,3),(3,5)]$, which can be written 135 since we have a simple graph, is a path, but 023 isn't
- A **cycle** is a path with the same source and target
  - example: 1231 and 1231231 are cycles
  - a cycle is **simple** if there is no repeating node
- The number of neighbours of a vertex is its **degree**
  - example: the node 2 has degree 3
- Nodes are **connected** if there is a path between them
  - example: the whole graph is **connected**

Typical algorithmic problem over graphs

Given an input (weighted) (multi)graph, compute:

- whether there are cycles
- whether the graph is connected
- the minimal length of a path connecting two nodes
- the maximal flow one may push through between two nodes

We want to do so **efficiently**.

Typical algorithmic problem over graphs

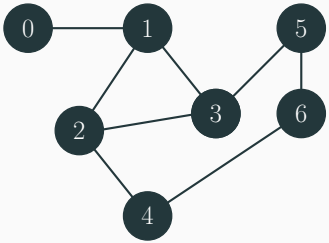
Given an input (weighted) (multi)graph, compute:

- whether there are cycles
- whether the graph is connected
- the minimal length of a path connecting two nodes
- the maximal flow one may push through between two nodes

We want to do so **efficiently**.

But wait, what is the *size* of a graph?

Size of a graph



The size of a graph is the sum $|V| + |E|$ of

- the number of vertices $|V|$
- the number of edges $|E|$

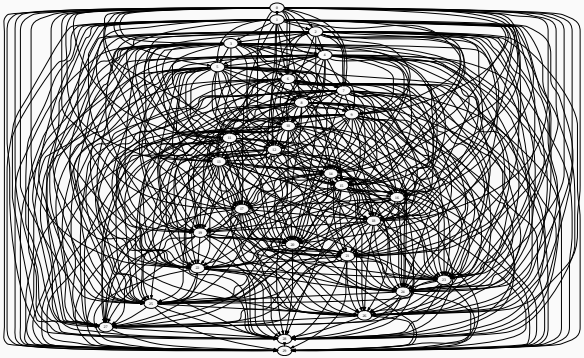
$|E|$ in function of $|V|$

If ≤ 1 edge between two vertices, $|E| = \mathcal{O}(|V|^2)$.

- this bound is tight
- many edges ($\Theta(|V|^2)$) = **dense** graph
- few edges = **sparse** graph
- if talking about classes of graphs, dense = $\Theta(|V|^2)$ edges and sparse = $o(|V|^2)$ edges

Just to get a sense of scale

Define the graph K_n on $\{0, ..., n-1\}$ by setting $E = \{(i, j) \mid i < j < n\}$.

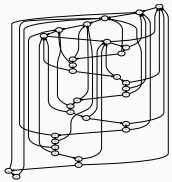


What is $|E|$ for K_{30}? Would you call K_n dense?

In practice: a lot of graphs are sparse

In pratice, graphs may be rather sparse

- if given by e.g. a rail map: the degree of each node will tend to be low
- typical if we have geometric constraints



To make a quick estimates: do the nodes have high degree?

For simple graphs $G = (V, E)$ (undirected, no self-loops, no parallel edges)

$$|E| = \frac{\sum_{v \in V} \text{degree(v)}}{2} \leq \frac{|V| \cdot \max_{v \in V} \text{degree(v)}}{2}$$

OK now that we have a sensible notion of size...

Question

How do we represent graphs in the computer?

- Index vertices by number from 0 to $|V| - 1$
- Two strategies: **adjacency matrices** or **adjacency lists**

Adjacency matrices

Store whether an edge is there in a 2D array

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

```
int[][] adjMat = new int[7][7];
adjMat[0][1] = adjMat[1][0] = 1;
adjMat[1][2] = 1;
...
```

Adjacency lists

Have an array of lists of successors for each node

$$
\begin{aligned}
0 &\to 1 \\
1 &\to 0, 2, 3 \\
2 &\to 1, 3, 4 \\
3 &\to 1, 2, 5 \\
4 &\to 2, 6 \\
5 &\to 3, 6 \\
6 &\to 4, 5
\end{aligned}
$$

```
LinkedList<Integer>[] adjList =
  new LinkedList<Integer>[7];
adjList[0].add(1);
adjList[1].add(0);
...
```

Pros and cons

Adjacency matrices

- very easy to implement
- very fast access to edge information
- $\mathcal{O}(|V|^2)$ space used $\rightarrow$ not good for sparse graphs

Adjacency lists

- efficient operations
- might need a predecessor table as well for efficient info
- $\mathcal{O}(|E| + |V|)$ space used $\rightarrow$ good for all graphs

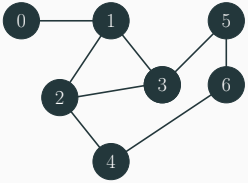
Now let us traverse graphs!

Goal

Given a graph $G = (V, E)$ and $v \in G$, enumerate all the vertices connected to v.

Two typical strategies: **breadth-first** and **depth-first**

breadth-first search (**BFS**):



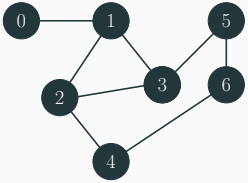
Now let us traverse graphs!

Goal

Given a graph $G = (V, E)$ and $v \in G$, enumerate all the vertices connected to v.

Two typical strategies: **breadth-first** and **depth-first**

breadth-first search (**BFS**): $0, 1, 2, 3, 4, 5, 6$



Now let us traverse graphs!

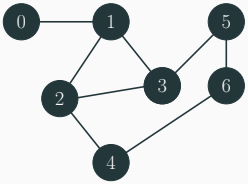
Goal

Given a graph $G = (V, E)$ and $v \in G$, enumerate all the vertices connected to v.

Two typical strategies: **breadth-first** and **depth-first**

breadth-first search (**BFS**): $0, 1, 2, 3, 4, 5, 6$

depth-first search (**DFS**):

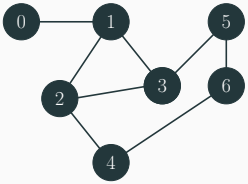


Now let us traverse graphs!

Goal

Given a graph $G = (V, E)$ and $v \in G$, enumerate all the vertices connected to v.

Two typical strategies: **breadth-first** and **depth-first**



breadth-first search (**BFS**): $0, 1, 2, 3, 4, 5, 6$

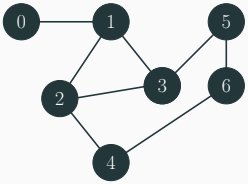
depth-first search (**DFS**): $0, 1, 2, 3, 5, 6, 4$

Now let us traverse graphs!

Goal

Given a graph $G = (V, E)$ and $v \in G$, enumerate all the vertices connected to v.

Two typical strategies: **breadth-first** and **depth-first**



breadth-first search (**BFS**): $0, 1, 2, 3, 4, 5, 6$

depth-first search (**DFS**): $0, 1, 2, 3, 5, 6, 4$

applications:

- Topological sort ($\cong$ figure out an order for dependencies)
- Checking connectedness
  (BFS/DFS, then check that all vertices were reached)
- Computing minimal paths/distance
  (BFS, keeping track of paths/distance)

How to write a BFS

Breadth-first search

Given an input graph (V, E) with $V = \{0, \ldots, n-1\}$ and $v \in V$

1. Allocate an array to keep track of visited vertices; initially regard all vertices as non-visited
2. Create an empty **queue**. Enqueue v.
3. While the queue is non-empty:
   - 3.1 Dequeue a vertex u and enumerate it; consider it visited.
   - 3.2 For each neighbour of u, if it was not visited before, enqueue it

How to write a BFS

Breadth-first search

Given an input graph (V, E) with $V = \{0, \ldots, n-1\}$ and $v \in V$

1. Allocate an array to keep track of visited vertices; initially regard all vertices as non-visited
2. Create an empty **queue**. Enqueue v.
3. While the queue is non-empty:
   - 3.1 Dequeue a vertex u and enumerate it; consider it visited.
   - 3.2 For each neighbour of u, if it was not visited before, enqueue it

Time complexity (with adjacency lists):

How to write a BFS

Breadth-first search

Given an input graph (V, E) with $V = \{0, \ldots, n-1\}$ and $v \in V$

1. Allocate an array to keep track of visited vertices; initially regard all vertices as non-visited
2. Create an empty **queue**. Enqueue v.
3. While the queue is non-empty:
   - 3.1 Dequeue a vertex u and enumerate it; consider it visited.
   - 3.2 For each neighbour of u, if it was not visited before, enqueue it

Time complexity (with adjacency lists): $\mathcal{O}(|V| + |E|)$

How to write a BFS

Breadth-first search

Given an input graph (V, E) with $V = \{0, \ldots, n-1\}$ and $v \in V$

1. Allocate an array to keep track of visited vertices; initially regard all vertices as non-visited
2. Create an empty **queue**. Enqueue v.
3. While the queue is non-empty:
   - 3.1 Dequeue a vertex u and enumerate it; consider it visited.
   - 3.2 For each neighbour of u, if it was not visited before, enqueue it

Time complexity (with adjacency lists): $\mathcal{O}(|V| + |E|)$

- Each node is dequeued at most once; 3. runs at most $|V|$ times

How to write a BFS

Breadth-first search

Given an input graph (V, E) with $V = \{0, \ldots, n-1\}$ and $v \in V$

1. Allocate an array to keep track of visited vertices; initially regard all vertices as non-visited
2. Create an empty **queue**. Enqueue v.
3. While the queue is non-empty:
   - 3.1 Dequeue a vertex u and enumerate it; consider it visited.
   - 3.2 For each neighbour of u, if it was not visited before, enqueue it

Time complexity (with adjacency lists): $\mathcal{O}(|V| + |E|)$

- Each node is dequeued at most once; 3. runs at most $|V|$ times
- 3. runs in $\mathcal{O}(\text{degree}(v))$ where v is the dequeued vertex, so

$$\text{time complexity} \leq \sum_{v \in V} K(1 + \text{degree}(v)) = K(2|E| + |V|)$$

What about writing a DFS

- The difference between a DFS and BFS is *the order in which we explore new nodes*
- Using a queue we explore first the nodes closest to the origin

→ Using a **stack** instead, we get a DFS!

What about writing a DFS

- The difference between a DFS and BFS is *the order in which we explore new nodes*
- Using a queue we explore first the nodes closest to the origin

→ Using a **stack** instead, we get a DFS!

Depth-first search

Given an input graph (V, E) with $V = \{0, \ldots, n-1\}$ and $v \in V$

1. Allocate an array to keep track of visited vertices; initially regard all vertices as non-visited
2. Create an empty **stack**. **Push** v.
3. While the **stack** is non-empty:
   - 3.1 **Pop** a vertex u and enumerate it; consider it visited.
   - 3.2 For each neighbour of u, if it was not visited before, enqueue it

What about writing a DFS

- The difference between a DFS and BFS is *the order in which we explore new nodes*
- Using a queue we explore first the nodes closest to the origin

→ Using a **stack** instead, we get a DFS!

Depth-first search

Given an input graph (V, E) with $V = \{0, \ldots, n-1\}$ and $v \in V$

1. Allocate an array to keep track of visited vertices; initially regard all vertices as non-visited
2. Create an empty **stack**. **Push** v.
3. While the **stack** is non-empty:
   - 3.1 **Pop** a vertex u and enumerate it; consider it visited.
   - 3.2 For each neighbour of u, if it was not visited before, enqueue it

Same space/time complexity.

Application: using a BFS to compute distances

For now let us assume all edges denote a distance of 1 between two nodes

(no edges mean an ∞ distance)

Distance using a kind of BFS

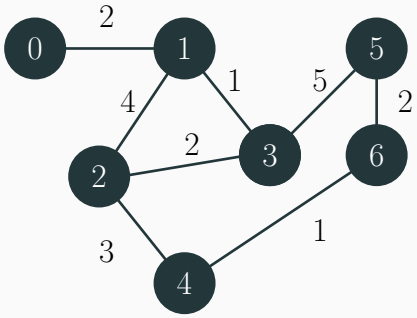
Given an input graph (V, E) with $V = \{0, \ldots, n-1\}$ two vertices s and t:

1. Allocate an array A of integers; set $A(s) = 0$ and $A(v) = \infty$ for $v \neq s$
2. Create an empty **queue**. Enqueue s.
3. While the queue is non-empty and $A(t) = \infty$:
   - 3.1 Dequeue a vertex u.
   - 3.2 For each neighbour v of u, if $A(v) = \infty$, set $A(v) = A(u) + 1$ and enqueue v

One can also check that this is in $\mathcal{O}(|E| + |V|)$.

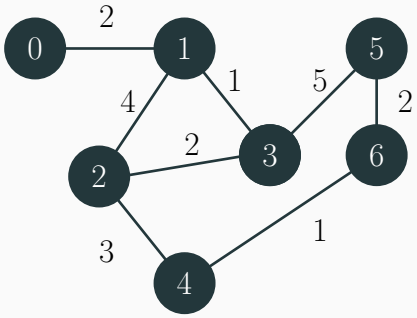
Distances in weighted graphs

If length of a path = sum of the weights of its edges, this won't do! ($2 \to 5$ below?)



Distances in weighted graphs

If length of a path = sum of the weights of its edges, this won't do! ($2 \to 5$ below?)



Dijkstra's algorithm

Use a priority queue instead of a queue.

Dijkstra's algorithm

`Djikstra(`G`,source)`

- $Q \leftarrow$ an empty priority queue
- Enqueue `source` with priority 0 in Q
- **while** Q *is not empty* **do**
  - Dequeue the element v with minimal priority d from Q
  - **if** v *was not visited before* **then**
    - Set the distance between `source` and v to be d
    - **for** *all edges* $v \xrightarrow{w} v'$ **do**
      - Enqueue v' with priority $d + w$ in Q
- **return** *the computed distances*

Running time $\mathcal{O}((|E| + |V|) \log(|V|))$

What if we want all distances between nodes?

- Run Dijkstra for every node $\mathcal{O}(|V|(|E| + |V|)\log(|V|))$
- We can do a bit better and simpler for dense graphs: $\mathcal{O}(|V|^3)$

The Floyd-Warshall algorithm

```
FloydWarshall(M)
    D ← a copy of M
    n ← dimension of M
    for k from 0 to n − 1 do
        for i from 0 to n − 1 do
            for j from 0 to n − 1 do
                D[i][j] ← min(D[i][j], D[i][k] + D[k][j])
    return D
```

To conclude

- Graphs = powerful abstraction/modelling tools
- We have seen how to traverse graphs, compute distance
  - Many problems that you can tackle using BFS/DFS
    - Compute connected components
    - Detect cycles
    - ...
- We have just scratched the surface though!
  - Computing **minimum spanning trees**
  - Computing **maximal flows**
  - Computing Eulerian cycles
  - Many classical NP-complete problems:
    - e.g., vertex cover, colorability and finding hamiltonian cycles

    → in real life: do check if you are trying to do something that is known to be hard!

In short: I can't do the topic justice in a single lecture :)