Week 6

Trees and BFS

General remarks

- We introduce **trees** and **forests** as important simple graphs.
- We consider the simplest graph-search algorithm, **breadth-first search** (BFS).
- We apply BFS to compute **shortest paths**.

Reading from CLRS for week 4

- Chapter 22, Section 22.2.
- Plus appendices

  B.5.1 "Free trees"

The notion of a tree

You might have seen already "rooted trees" (and you will see more of them in the following weeks) — **trees** are basically just like rooted trees, but without a designated root.

Definition *A* **tree** *is a graph with at least one vertex, which is connected and does not have a cycle.*

- Sometimes these "trees" are called "free trees" ("unrooted trees"), so that "rooted trees" can just become "trees".
- We use here the opposite convention: "trees" are always "free", while for the trees with roots we say "rooted trees".

Cycles

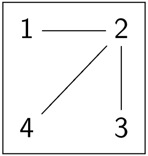
Intuitively, a graph G has a **cycle** if there are two different vertices u, v in G such that there are (at least) two essentially different ways of getting from u to v. And thus going from u to v the one way, and from v to u the other way, we obtain the "round-trip" or "cycle".

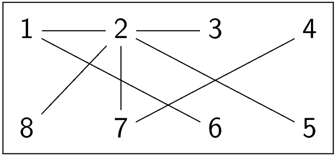
Definition *A* **cycle** *in a graph G is a sequence $v_1, \dots, v_n \in V(G)$, $n \geq 2$, of vertices of G with*

- *$v_1 = v_n$;*
- *the v_i for $1 < i < n$ are pairwise different, and different from $v_1 = v_n$;*
- *for all $1 \leq i < n$ holds $\{ v_i, v_{i+1} \} \in E(G)$ (that is, the vertices v_i, v_{i+1} are adjacent in G).*

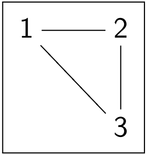
Examples

Here are two trees:





And here is the smallest connected graph with at least one vertex, which is *not* a tree (the "triangle" K_3):



And here the smallest graph with at least one vertex (namely with two vertices), which is *not* a tree:

Examples (cont.)

The complete graphs K_n are trees iff $n = 1$ or $n = 2$:

- For $n = 0$ we are missing a vertex.
- For $n \geq 3$ there are cycles.

Understanding trees:

Trees are the sparsest connected graphs.

The Fundamental Lemma:

- It must become clear, that to connect n things, you need at least $n - 1$ **pairwise** "connections".
- And if you have any more connections, you have a cycle!

The Fundamental Lemma

We have the following fundamental characterisation of trees:

Lemma 1 *A graph G is a tree if and only if*

- *G is connected,*
- *$|V| \geq 1$,*
- *and $|E| = |V| - 1$ holds.*

In other words:

A connected graph with $n \geq 1$ vertices has a cycle
if and only if it has at least n edges
(and it has *always* at least $n - 1$ edges).

So trees realise minimal ways of connecting a set of vertices.

Spanning trees

Definition *Consider a connected graph G with at least one vertex. A* **spanning tree** *of G is a tree T with $V(T) = V(G)$ and $E(T) \subseteq E(G)$.*

So spanning trees just leave out edges which are not needed to connect the whole graph. For example consider the graphs

$$G_1 = \begin{tikzcd} 1 \arrow[r, dash] \arrow[d, dash] & 2 \arrow[d, dash] \\ 3 \arrow[r, dash] & 4 \end{tikzcd}, \quad G_2 = \begin{tikzcd} 1 \arrow[r, dash] \arrow[d, dash] \arrow[dr, dash] & 2 \arrow[d, dash] \\ 3 \arrow[r, dash] & 4 \end{tikzcd}.$$

G_1 has 4 different spanning trees, while G_2 has $4 + 4 = 8$.

Examples

1. The spanning trees of the K_n are precisely all *existing* trees on these n vertices.
   By Cayley's formula the complete graph K_n, $n \geq 1$, has exactly
   $$n^{n-2}$$
   many spanning trees.
2. The spanning trees of an undirected cycle are obtained by removing any single edge.

Spanning trees are good to minimise costs.
But they also minimise safety.

Computing spanning trees

We will see two algorithms, BFS and DFS, for computing spanning trees:

- In both cases actually **rooted spanning trees** are computed, that is, additionally a vertex is marked as "root".
- When drawing such rooted spanning trees, one starts with the root (otherwise one could start with an arbitrary vertex!), going from the root to the **leaves**.
- For such rooted spanning trees, one typically speaks of **nodes** instead of vertices.
- Both algorithms compute additional data, besides the rooted spanning trees.
- The DFS version is by default extended to compute a **spanning forest**: It can be applied to non-connected graphs, and computes a (rooted) spanning tree for each connected component.

Representing rooted trees

A **rooted tree**, that is, a tree together with a root T, will be represented by BFS and DFS as follows:

- Now there is a **direction** in the tree, from the root towards the leaves.
- We obtain the usual notion of the **children** of a node (without a root, in a "pure tree", there is no such thing).
- The **leaves** are the nodes without children.
- And we speak of the(!) **parent** of a node (note that every node can have at most one parent).
- The **root** is the only vertex without a parent.

Specifying the parent for *each* non-root vertex
is sufficient to represent the tree.

This is done in BFS and DFS by an array π (like "parent").

Example

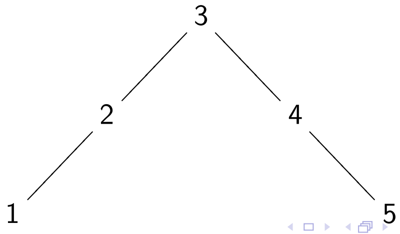
Consider K_5:

- we have 5 vertices
- and $5 \cdot 4/2 = 10$ edges (all possible ones).

For a *spanning tree* we have to remove 6 edges, without breaking the connectivity.

For example the path graph P_5, the path with 5 vertices, will do.

Now let's root it, taking for example 3 as the root, obtaining a *rooted tree* $(P_5, 3)$, which is a spanning tree of K_5:



Example (cont.)

Now the representation is:

1. Node 1 has parent 2: $\pi[1] = 2$.
2. $\pi[2] = 3$.
3. $\pi[4] = 3$.
4. $\pi[5] = 4$.
5. $\pi[3] = \mathrm{NIL}$.

The notions of "forests"

Definition *A* **forest** *is a graph where every connected component is a tree.*

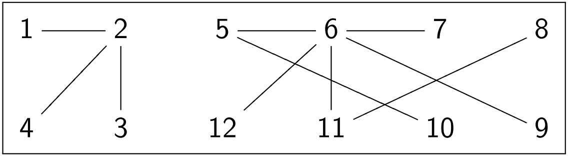
Recall:

- A *connected component* of a graph contains some vertex and precisely all vertices reachable from it.
- So a graph is connected if and only it has at most one connected component.
- Note that the empty graph (with the empty vertex-set) is connected and has zero connected components.

Every tree is a forest (but not the other way around).

Spanning forests

Considering the so-called “disjoint union” of the *two* trees we have seen previously, we get a (proper) forest (now taking this as *one* graph — note the necessity of the renaming):



Lemma 1 *A graph G is a forest if and only if G contains no cycle.*

Definition *Consider any graph G. A* **spanning forest** *of G is a forest F with $V(F) = V(G)$ and $E(F) \subseteq E(G)$.*

Lemma 2 *Every graph G has a spanning forest.*

If we have any spanning forest F of a graph G, then F is a spanning tree of G iff G is connected and has at least one vertex.

Spanning forests

A graph G has a spanning tree if and only if G is not empty and G is connected.

- On the other hand, every graph has a spanning forest.
- These are precisely given by the spanning trees of the connected components (taken together).

If we have a disconnected graph, then we can just break it up into its components, and treat them separately.

In this sense the case of
connected graphs and spanning trees
is the central case.

However for real software systems, we cannot just assume the graphs to be connected, and there we handle forests.

This is just done by restarting BFS (or later DFS)
on the vertices not covered yet.

Breadth-first search

Searching through a graph is one of the most fundamental of all algorithmic tasks.

Breadth-first search (BFS) is a simple but very important technique for searching a connected graph:

- Such a search starts from a given source vertex s and constructs a *rooted* spanning tree for the graph, called the **breadth-first tree** (BFS tree; the root is s).
- It uses a (first-in first-out) **queue** as its main data structure.
- BFS computes the parent $\pi[u]$ of each vertex u in the breadth-first tree (the parent of the source/root is NIL).
- The speciality of BFS is that it computes also the distances $d[u]$ from the source s (initialised to ∞). This is also the length of the path from s to u in the breadth-first tree.
- Thus the BFS tree contains the **shortest paths** from s to any other vertex, and is called an **SPT**.

The algorithm

Input: A graph G with vertex set $V(G)$ and edges represented by adjacency lists *Adj*.

Queue A queue is a “first-in-first-out data structure”.

$\mathrm{BFS}(G, s)$

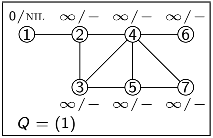
for each $u \in V(G)$
    $d[u] = \infty$
$\pi[s] = \mathrm{NIL}$
$d[s] = 0$
$Q = (s)$
while $Q \neq ()$
    $u = \mathrm{DEQUEUE}[Q]$
    for each $v \in Adj[u]$
        if $d[v] = \infty$
            $d[v] = d[u] + 1$
            $\pi[v] = u$
            $\mathrm{ENQUEUE}(Q, v)$

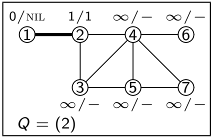
BFS illustrated

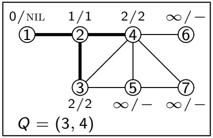
$d[u]\,/\,\pi[u]$

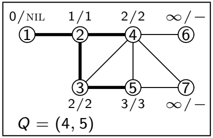
u

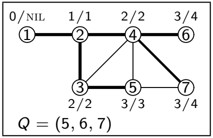
(labelling)

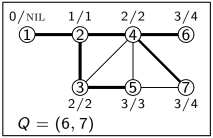


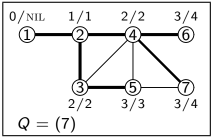


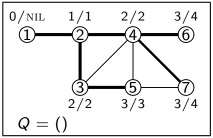












Analysis of BFS

Correctness Analysis: At termination of $\mathrm{BFS}(G, s)$, for every vertex v reachable from s we (should) have:

- v has been encountered;
- $d[v]$ holds the length of the shortest path from s to v (a natural number ≥ 0);
- $\pi[v]$ is the penultimate node on a shortest path from s to v for $v \neq s$ (while $\pi[s] = \mathrm{NIL}$).

For every vertex v NOT reachable from s we have $d[v] = \infty$.

Definition *Consider a graph G, and two vertices $u, v \in V(G)$. The* **distance** *between u, v in G is the minimum number of edges in a path in G connecting u, v, if there is a path, and otherwise it is ∞.*

- So the distance of a vertex to itself is 0.
- And the distance between vertices is ∞ iff these vertices are in different connected components.

Analysis of BFS (cont.)

Time Analysis:

- The initialisation takes time $\Theta(V)$.
- Each vertex is ENQUEUEd once and DEQUEUEd once; these queueing operations each take constant time, so the queue manipulation takes time $\Theta(V)$ (altogether).
- The Adjacency list of each vertex is scanned only when the vertex is DEQUEUEd, so scanning adjacency lists takes time $\Theta(E)$ (altogether).

The overall time of BFS is thus $\Theta(V + E)$.

In other words, the runtime of BFS is

linear in the size of the graph.

Background: Why do we get shortest paths?

Is it really true that we get always shortest paths (that is, using the minimum number of edges)?

1. s gets the correct distance 0, and is the only vertex with that distance.
2. Exactly the neighbours of s are the vertices with distance 1 (from s).
3. We process these vertices first and completely, assigning distance 1 to them. Essential that we use a *queue* here.
4. All the unexplored neighbours of these vertices are exactly all the vertices with distances 2. Again, due to using a queue, we process these vertices first and completely.
5. In general, we can assume that we found exactly all vertices with distance $\leq k$, and the unexplored neighbours of vertices at distance $= k$ are then precisely the vertices at distance $k + 1$. and indeed they get this distance assigned.

Running BFS on a disconnected graph

A graph G with at least one vertex s is connected if and only if $\mathrm{BFS}(G, s)$ yields no $d[v] = \infty$.

The vertices v with $d[v] = \infty$ are precisely
the vertices not reachable from s.

So for a disconnected G, running $\mathrm{BFS}(G, s)$ does not yield a spanning tree for G,

but a spanning tree for the component of s.

For example, running BFS on
$G := (\{1, 2, 3, 4\}, \{\{1, 2\}, \{3, 4\}\})$ with $s = 1$ we get

- $\pi[1] = \mathrm{NIL}$, $\pi[2] = 1$, $\pi[3], \pi[4]$ not set.
- $d[1] = 0$, $d[2] = 1$, $d[3] = d[4] = \infty$.

The number of vertices and edges

What is the number of vertices and edges in the following examples?

1. $(\emptyset, \emptyset)$:

The number of vertices and edges

What is the number of vertices and edges in the following examples?

1. $(\emptyset, \emptyset)$: $0, 0$
2. $(\{1, 4, 6\}, \{\{4, 1\}, \{4, 6\}\})$:

The number of vertices and edges

What is the number of vertices and edges in the following examples?

1. $(\emptyset, \emptyset)$: $0, 0$
2. $(\{1,4,6\}, \{\{4,1\},\{4,6\}\})$: $3, 2$
3. K_n:

The number of vertices and edges

What is the number of vertices and edges in the following examples?

1. $(\emptyset, \emptyset)$: $0, 0$
2. $(\{1, 4, 6\}, \{\{4, 1\}, \{4, 6\}\})$: $3, 2$
3. K_n: $n, \frac{1}{2}n(n-1)$
4. C_n:

The number of vertices and edges

What is the number of vertices and edges in the following examples?

1. $(\emptyset, \emptyset)$: $0, 0$
2. $(\{1, 4, 6\}, \{\{4, 1\}, \{4, 6\}\})$: $3, 2$
3. K_n: $n, \frac{1}{2}n(n-1)$
4. C_n: n, n.
5. A spanning tree of K_n:

The number of vertices and edges

What is the number of vertices and edges in the following examples?

1. $(\emptyset, \emptyset)$: $0, 0$
2. $(\{1, 4, 6\}, \{\{4, 1\}, \{4, 6\}\})$: $3, 2$
3. K_n: $n, \frac{1}{2}n(n-1)$
4. C_n: n, n.
5. A spanning tree of K_n: $n, n-1$ (for $n \geq 1$)
6. A spanning tree of a graph with n vertices:

The number of vertices and edges

What is the number of vertices and edges in the following examples?

1. $(\emptyset, \emptyset)$: $0, 0$
2. $(\{1, 4, 6\}, \{\{4, 1\}, \{4, 6\}\})$: $3, 2$
3. K_n: $n, \frac{1}{2}n(n-1)$
4. C_n: n, n.
5. A spanning tree of K_n: $n, n-1$ (for $n \geq 1$)
6. A spanning tree of a graph with n vertices: $n, n-1$ (for $n \geq 1$)
7. A spanning forest of a graph with n vertices and m connected components:

The number of vertices and edges

What is the number of vertices and edges in the following examples?

1. $(\emptyset, \emptyset)$: $0, 0$
2. $(\{1, 4, 6\}, \{\{4, 1\}, \{4, 6\}\})$: $3, 2$
3. K_n: $n, \frac{1}{2}n(n-1)$
4. C_n: n, n.
5. A spanning tree of K_n: $n, n-1$ (for $n \geq 1$)
6. A spanning tree of a graph with n vertices: $n, n-1$ (for $n \geq 1$)
7. A spanning forest of a graph with n vertices and m connected components: $n, n-m$.

Spanning trees

List all spanning trees of

$$
\begin{tikzcd}
1 \arrow[r, dash] \arrow[d, dash] \arrow[dr, dash] & 2 \arrow[d, dash] \\
3 \arrow[r, dash] & 4
\end{tikzcd}
.
$$

If you already did so, then you might attempt to create all 16 spanning trees of the complete graph with 4 vertices:

$$
K_4 = \begin{tikzcd}
1 \arrow[r, dash] \arrow[d, dash] \arrow[dr, dash] & 2 \arrow[d, dash] \arrow[dl, dash] \\
3 \arrow[r, dash] & 4
\end{tikzcd}
.
$$

Solution

The first task is to delete 2 edges such that the graph stays connected (and to find all such possibilities):

- If we delete the diagonal, then we can delete any other edge, yielding 4 spanning trees.
- If we do not delete the diagonal, then:
  - We can delete either the two other edges incident with vertices 1 or 4.
  - Or either delete the two horizontal or the two vertical edges.

  That makes 4 more spanning trees.

You might have a look at Wikipedia for further information (there you find the 16 spanning trees of K_4).