Graph Searching

Learning objectives:

- ☐ write pseudocode for breadth-first search (BFS) and depth-first search (DFS) algorithms,
- ☐ build a spanning tree using a DFS and BFS,
- ☐ build a minimum spanning tree using Prim's algorithm,
- ☐ apply BFS and DFS to some graph problems.

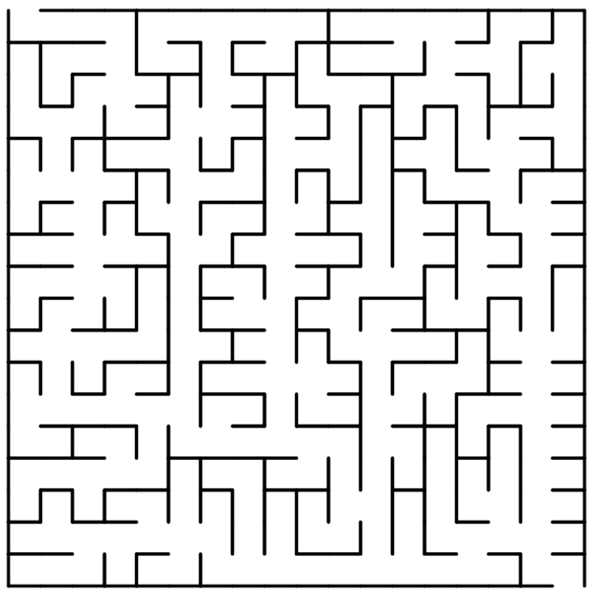
Last lecture we introduced spanning trees and we proved that every connected graph has a spanning tree. A natural question arises: *how do we compute spanning trees?* Today, we'll answer that question using the breadth-first and depth-first search algorithms. We'll also introduce a special type of spanning tree called the minimum spanning tree. After today's lecture, think about how you might apply what you learn to some problems in computer science. Can you apply your knowledge to color a graph? Can you think of a way to create a web crawler?

Did you know?

Some of the algorithms we will see today can be used to get out of a maze. Specifically, Trémaux's algorithm is related to depth-first search and guarantees that you will get out of a maze (though it might not be the shortest path).

Example 1:

Starting at the upper-left corner, find a path out through the maze to exit in the bottom right corner. Describe the approach you used and why you used it. Can you relate it to graph theory?



1 Depth-first search (DFS)

In order to build a spanning tree, we need to make sure of two things:

1. the spanning tree remains connected,
2. the spanning tree has no cycles.

One idea, then, for building a spanning tree, is to pick some arbitrary vertex in a graph (which will be the root) and keep adding edges incident to the last vertex added such that the above properties are still satisfied. We can ensure that no cycles are created by also ensuring that the edge we add is not incident to a vertex already existing in the tree. More formally, Algorithm 1 describes these steps. This algorithm is known as **depth-first search** because we are starting from a vertex and traversing the graph by *depth* to form the spanning tree.

This algorithm is sometimes referred to as **backtracking** because the algorithm returns to previous vertices when traversing the graph.

How many checks are performed in DFS?

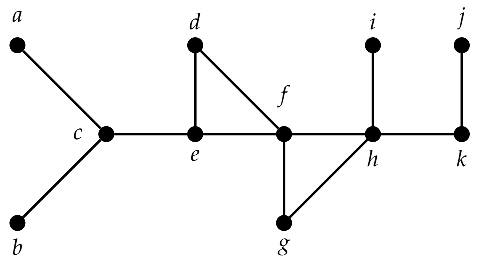
We need to visit every vertex once as well as every edge attached to that vertex. If we are given the **adjacency matrix** for G, then we need about $|V|^2$ checks since we need to traverse every row of the adjacency matrix to determine if a vertex neighbors the one being visited. If we are given the **adjacency list** for G, then we need about $|V|$ checks. However, if the number of edges attached to a vertex is large, then this could be as high as $|V|^2$.

```
depthFirstSearch($G$)

    input: connected graph $G = (V_G, E_G)$
    output: spanning tree $T$
1   $u \leftarrow$ arbitrary vertex in $V_G$
2   $T \leftarrow (\{u\}, \emptyset)$
3   visit($u, G, T$)
```

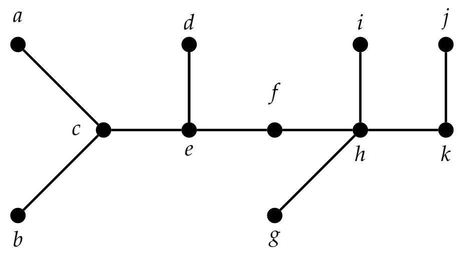
Algorithm 1: Depth-first search.

```
visit($u, G, T$)

    input: starting vertex $u$, connected graph $G = (V_G, E_G)$,
           current spanning tree $T = (V_T, E_T)$
    output: updated spanning tree $T = (V_T, E_T)$
1   for $v \in$ neighbors($u, G$)
2       if $v \in V_T$
3           continue
4       $E_T \leftarrow$ append($\{u, v\}$)
5       $V_T \leftarrow$ append($v$)
6       visit($v, G, T$)
```

Example 2:
Use depth-first search to find the spanning tree of the following graph.

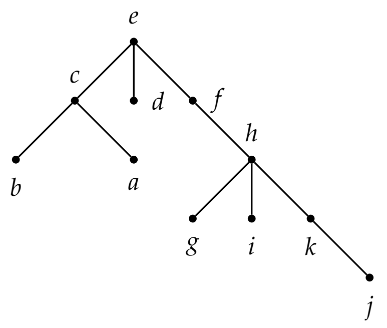


Solution:
The spanning tree is shown below. Starting with e, the sequence of steps taken to draw the spanning tree is shown in the different levels of the figure on the right. Note that the resulting spanning tree is dependent on the starting vertex as well as the order in which the adjacencies are traversed.



Which order should be used to visit adjacent vertices?

In the example on the left, we didn't necessarily go in alphabetical order, but supposed the vertex-vertex adjacencies were stored on some arbitrary order. In general, I'll be explicit about the order in which neighbors should be visited.



2 Breadth-first search (BFS)

Instead of backtracking to previous vertices, we can traverse a graph by looking at every neighbor at the current level before proceeding to the next level. This type of traversal is known as **breadth-first search** (BFS). Similar to DFS, BFS adds edges to the tree as long as we haven't already added the opposite vertex of the edge to the tree. This algorithm is described in Algorithm 2. BFS is useful if you want to find the connected components of a graph or determine if a graph is bipartite.

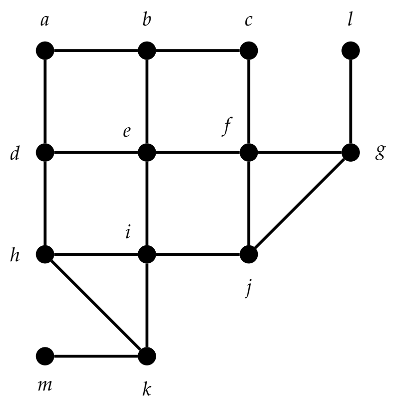
```
breadthFirstSearch($G$)

input: connected graph $G = (V_G, E_G)$
output: spanning tree $T = (V_T, E_T)$
$u \leftarrow$ arbitrary vertex in $G$
$T \leftarrow (\{u\}, \emptyset)$
$L \leftarrow \{u\}$    # unprocessed vertices
while $L \neq \emptyset$
    $v \leftarrow$ pop($L$)    # remove first vertex from $L$
    for $w \in$ neighbor($v, G$)
        if $w \in L \vee w \in V_T$
            continue
        $L \leftarrow L \cup w$
        $V_T \leftarrow$ append($w$)
        $E_T \leftarrow$ append($\{v, w\}$)
```

Algorithm 2: Breadth-first search.

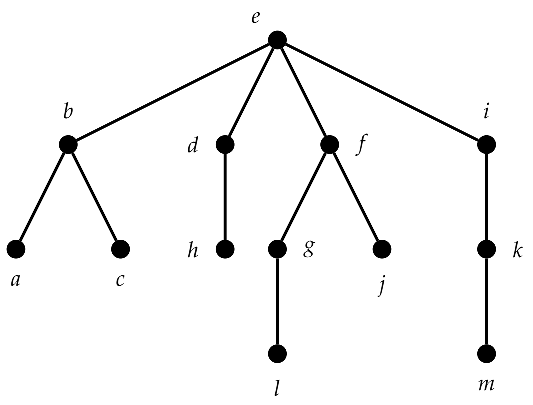
How many checks are performed in BFS?

BFS performs about $|V|$ checks since every vertex is visited. It also explores the adjacencies of every vertex. Depending on the degree of the graph, this could be as high as $|V|^2$. However, this is generally improved to about $|V|$ checks for low degrees.

Example 3:
Use breadth-first search to find the spanning tree of the following graph starting at vertex e (see Rosen Chapter 11.4 Example 5).



Solution:
The steps in forming the tree are shown in the four levels below. Note that each leave is completed before moving onto the next level.



3 Minimum Spanning Trees (MST)

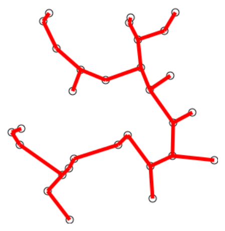


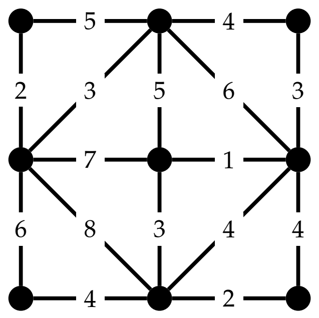
Figure 1: Minimum spanning tree of random points in the plane. Here's a really nice demo.

We have seen a few methods for building spanning trees, but if the graph has *weights* on its edges, we may want to compute a *minimum spanning tree.*

Definition 1. *A minimum spanning tree of an edge-weighted graph G is the spanning tree of G with the* smallest *possible sum of edge weights.*

We can apply our traversal algorithms to account for these edge weights. The following algorithm (Algorithm 3) is known as Prim's algorithm for building a minimum spanning tree. The check on Line 6 (checking if a cycle is formed) can be done by checking if both vertices on the edge e have already been added to the tree.

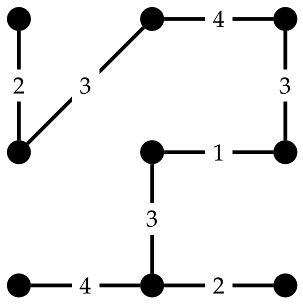
Example 4:
Build the minimum spanning tree of the following graph using Prim's algorithm.



```
minimumSpanningTree$(G, W)$

    input: graph $G = (V_G, E_G)$ with weights $W$ on the edges $E_G$
    output: minimum spanning tree $T = (V_T, E_T)$
1   $E_G \leftarrow$ sort$(E_G, W_G)$   # sort edges by increasing weight
2   while $|E_T| < |V_G| - 1$   # property of trees
3       for $e \in E_G$
4           if $e \in E_T \vee$ (vertices of $e$) $\notin V_T$   # maintain connectivity
5               continue
6           if adding $e$ to $T$ forms a cycle
7               continue
8           $E_T \leftarrow$ append$(e)$
9           $V_T \leftarrow$ append(vertices of $e$)
10          break   # restart loop to find edge with min weight
```

Algorithm 3: Prim's algorithm for finding a minimum spanning tree.

Solution:
We can use Prim's algorithm to generate the minimum spanning tree. The resulting weight is: $1 + 3 + 2 + 4 + 3 + 4 + 3 + 2$.



How many checks are performed in Prim's algorithm?

If you use an adjacency matrix or adjacency list to represent the graph, then the algorithm requires $|E|^2$ or $|V|^2$ checks since we are checking every possible edge that we can add next in the list (in the worst, case $|E|$ times).

There are other data structures, specifically heaps, that you can use to improve performance.