Math 428, Breadth-First and Depth-First Search, Outline

We start with a remark about Dijkstra’s algorithm. Let $G$ be a connected graph of order $n$ with non-negative edges weights and let $u$ be a starting vertex. Let $u, v_1, v_2, \ldots, v_{n-1}$ list the vertices of $G$ in the order in which Dijkstra’s algorithm adds them to the growing, shortest routes tree for $u$. A simple induction argument shows that the distances from $u$ along this sequence are non-decreasing: that is,

$$0 = d(u, u) \leq d(u, v_1) \leq d(u, v_2) \leq \cdots \leq d(u, v_{n-1}).$$

Suppose we are given a complicated, unweighted, connected graph $G$ and a vertex $u$ of $G$, and we want to compute distances of the other vertices in the graph from $u$ and geodesic paths from $u$ to the other vertices. To do this, we can assign weight 1 to every edge of $G$ and use Dijkstra’s algorithm, which will produce a shortest routes spanning tree for $u$. A method like this which visits all the vertices of a graph in some organized fashion is called a graph search. We know from the initial remark that the search using Dijkstra’s algorithm will add vertices in order of increasing distance from $u$; thus it will first add all vertices a distance one from $u$, then all vertices a distance two from $u$, and so on. For this reason, it is called a Breadth-First Search (BFS) of $G$.

BFS is important in theoretical computer science and algorithmic graph theory. This lecture will present a formulation of BFS more streamlined than a direct application of Dijkstra’s algorithm. The discussion is based on material in D. West, An Introduction to Graph Theory, Prentice Hall, 1996, Chapter 2.

Throughout, $G$ will denote a connected graph and $u$ a starting vertex. The algorithm will build the BFS tree starting from $T = \{u\}$, by adding new edges, and the vertices incident to the new edges, one-by-one. We shall define the operation of adding edges incident to a vertex $v$ in $T$ as a “search” of $v$. Specifically, a search of $v$ in $T$ consists in the addition to $T$ of every edge in $G$ of the form $vx$ where $x$ is not already in $T$. So once a search of $v$ is completed, $v$ will have no neighbors outside of $T$. We shall imagine that vertices are assigned a priority in some way and that in the search of $v$ they are added in order of priority; for example they may be numbered and prioritized by numerical order. Once a vertex is added to $T$, it awaits
its turn to be searched. All important to the algorithm is the order in which vertices are searched. This is controlled by maintaining an ordered list \( R \) of all the vertices which are in \( T \) but have not yet been searched; one says that vertices in \( R \) have been “reached” by \( T \), but not searched. BFS works by using a first-in/first-out protocol. Given a current tree \( T \) and a current list \( R \), the algorithm chooses the first vertex in \( R \) to search next. As it searches, it places each new vertex that has been added to the tree at the end of the list \( R \). It continues this procedure until \( T \) is a spanning tree. Here is a formal statement with the minor details clarified and a distance computation added.

**Breadth-First Search**

**Input:** A connected graph \( G \) and a starting (root) vertex \( u \).

**Output:** A shortest routes spanning tree and distances \( d(u, v) \) to all other vertices of \( G \).

**Algorithm:**

1. Initialize with \( T = \{u\} \), \( R = \{u\} \).

2. As long as \( R \) is nonempty, do the following. Search the first vertex \( v \) in \( R \). As the search of \( v \) proceeds add each new vertex \( x \) to the end of \( R \) and set \( d(u, x) = d(u, v) + 1 \). At the end of the search, remove \( v \) from \( R \).

To get some feel for BFS suppose we apply it to a tree. Of course the spanning tree produce by BFS will merely replicate the tree, but it is of interest to observe the order in which the vertices are searched. Consider then the very simple example below, in which the starting vertex is the leftmost vertex labelled 1. Suppose that the vertices are prioritized so that \( x \) has higher priority than \( y \) if \( x \) is lower down in the drawing than \( y \). Then the vertices are added to the tree in the order shown by the numbers. The algorithm starts with \( T = \{1\} \) and \( R = \{1\} \) and a search of vertex 1. This adds the vertices 2 and 3 and they are entered in that order in the list \( R \), which after the search of 1 and removal of vertex 1 from \( R \) is \( R = \{2, 3\} \). Then vertex 2 is searched and after this step \( R = \{3, 4, 5\} \), and so on. It is easy to see in this example that the algorithm searches all vertices distance \( k \) from \( u \) before moving on to vertices at distance \( k + 1 \).
It is interesting to explore the relationship of the edges of $G$ not in a breadth-first search spanning tree. For this we make a definition.

**Definition:** Let $T$ be a tree and let $u$ be a vertex of $T$ designated as its “root.” We say that vertex $v$ is an *ancestor* of vertex $w$, or equivalently that $w$ is a descendant of $v$ if $v$ is lies on the unique path in the tree connecting $u$ and $w$. The root $u$ is the ancestor of all other vertices in the tree. Two vertices are said to be *related* if one is the ancestor of the other.

There is an interesting fact concerning the relation of the edges of $G$ which are not included in a breadth-first spanning tree to the vertices of the tree itself.

**Theorem 1** Let $G$ be a connected graph, let $u$ be a vertex of $G$, and let $T$ be the spanning tree obtained by doing a breadth-first search starting at $u$. Then if $e = xy$ is an edge of $G$ not in the spanning tree $T$, the vertices $x$ and $y$ are not related, (such edges are called cross-edges since they connect different branches of the tree).

It is not to hard to see why this is true. If vertex $y$ is a descendent of vertex $x$ and if $yx$ is not an edge of the tree, then the path in $T$ from $u$ to $y$ passes through $x$ and at least one intermediate vertex before arriving at $y$. Since $T$ is a shortest routes tree for the vertex $u$, this means that
Thus \( d(u, y) \geq d(u, x) + 2 \). Thus \( xy \) cannot be an edge in \( G \), for this would imply \( d(u, y) \leq d(u, x) + 1 \).

The figure below shows the breadth-first search tree of a graph \( G \). The edges of \( G \) that are not in \( T \) are shown with lighter lines. Notice that the vertices are arranged in columns with each successive column containing all the vertices of the same distance from \( u \). The cross edges either connect vertices in the same column or adjacent columns. This reflects another property of the edges of \( G \) not in \( T \); because \( T \) is a shortest routes tree, an edge not in \( T \) must connect vertices whose distances from \( u \) are either the same or differ by 1.

![Figure 2.](image-url)

**Depth-First Search**

Breadth-First Search operates using what we consider a fair service protocol: the vertices are put into a queue, namely the list \( R \), and searched in order of arrival, that is, first-in/first-served. Depth-First Search operates on completely the opposite principle, using the most unfair service protocol—first-in/last served. Like Breadth-First Search, Depth-First Search builds a spanning tree by adding edges and vertices one at a time. But, roughly, Depth-First Search works by always adding an edge and vertex to the last vertex added to the tree, if possible; if not possible, it moves back vertex by vertex in the current tree until it does find a vertex to which it can add an
As with BFS, Depth-First Search can be organized formally, by means of an ordered list $R$ of certain of the vertices of $T$. Given a current tree $T$ and list $R$, the next iteration of the DFS algorithm looks at the first vertex $v$ in $R$. If $v$ has no neighbors not already in $T$, it is removed from the list and when this happens, $v$ is said to be “finished.” If $v$ has any neighbors not already in $T$, one of these neighbors, say $x$, is added to the tree along with the edge $vx$, and $x$ is put at the front of $R$, without removal of $v$. The list $R$ contains all the current, unfinished vertices of $T$, and the algorithm stops when $R$ becomes empty. The procedure is initialized by setting $T = \{u\}$ and $R = \{u\}$, where $u$ is a starting vertex specified as an input to the algorithm. The output of DFS is a spanning tree, but it is not a shortest routes tree. What we have just said is a full and precise definition of the algorithm, so we won’t bother with stating it in the usual summarized fashion.

As an example, let us first do a DFS of the tree of Figure 1. Of course, DFS will reproduce this tree, but of more interest to us is the order in which vertices are “found” by the algorithm and this we indicate by integer labels. Again, whenever there is a choice, we add the bottom-most vertex possible. Notice how different the numbering is from Figure 1.

![Figure 3.](image-url)
As a second example, consider a DFS of the graph in Figure 2. It leads to the very different spanning tree shown in Figure 4. Again, when there is a choice, we take the bottom-most vertex available. For added emphasis, we have numbered the vertices in the order in which they were found.

![Figure 4](image)

By examining Figure 4, you will notice that every edge of $G$ not in $T$ connects related vertices, that is, connects a vertex and a descendent of that vertex. In contrast to a Breadth-First Search tree, there are no cross edges. This illustrates an important general property of DFS spanning trees.

**Theorem 2** Let $T$ be a spanning tree obtained by Depth-First Search starting from some vertex $u$ of a connected graph $G$. Then every edge of $G$ that is not in $T$ connects vertices that are related relative to the root $u$.

To see why this theorem is true, suppose that vertex $y$ is found, that is, is added to the tree and put in the list $R$, before vertex $x$, and suppose that $yx$ is an edge of $G$. Now once $y$ is added to $R$, all the vertices that are added to the tree while $y$ is still in $R$, and hence unfinished, are descendents of $y$. Vertex $y$ will not be finished as long as there is an edge from $y$ to a vertex not in $T$, so certainly $y$ will not be finished before $x$ is added to the tree. Thus $x$ must be a descendent of $y$. 