22.1 Breadth-first Search

A search is a systematic way of searching through the nodes for a specific node. The two standard searches are breadth-first search and depth-first search, which both run in linear time.

The idea behind breadth-first search is to:

Visit the source; then all its neighbors; then all their neighbors; and so on.

If the graph is a tree and one starts at the root, then one visits the root, then the root’s children, then the nodes at depth 2, and so on. That is, one level at a time. This is sometimes called level ordering.

BFS uses a queue: each time a node is visited, one adds its (not yet visited) out-neighbors to the queue of nodes to be visited. The next node to be visited is extracted from the front of the queue.

Algorithm: BFS (start):

enqueue start
while queue not empty {
    v = dequeue
    for all out-neighbors w of v
        if ( w not visited ) {
            visit w
            enqueue w
        }
}

22.2 Depth-First Search

The idea for depth-first search (DFS) is "labyrinth wandering":

keep exploring new vertex from current vertex; when get stuck, backtrack to most recent vertex with unexplored neighbors
In DFS, the search continues going deeper into the graph whenever possible. When the search reaches a dead end, it backtracks to the last (visited) node that has unvisited neighbors, and continues searching from there. A DFS uses a **stack**: each time a node is visited, its unvisited neighbors are pushed onto the stack for later use, while one of its children is explored next. When one reaches a dead end, one pops off the stack. The edges/arcs used to discover new vertices form a tree.

**Example.** Here is a graph and a DFS-tree from vertex A:

If the graph is itself a tree, we can still use DFS. Here is an example:

**Algorithm:** DFS(v):

for all edges e outgoing from v

w = other end of e

if w unvisited then {

label e as tree-edge

recursively call DFS(w)

}

Note:

- DFS visits all vertices that are reachable
- DFS is fastest if the graph uses adjacency list
- to keep track of whether visited a vertex, one must add field to vertex (the decorator pattern)

**22.3 Test for Strong Connectivity**

Recall that a directed graph is strongly connected if one can get from every vertex to every other vertex. Here is an algorithm to test whether a directed graph is strongly connected or not:
Algorithm: 1. Do a DFS from arbitrary vertex $v$ & check that all vertices are reached
2. Reverse all arcs and repeat

Why does this work? Think of vertex $v$ as the hub...

### 22.4 Distance

The *distance* between two vertices is the minimum number of arcs/edges on path between them. In a weighted graph, the *weight* of a path is the sum of weights of arcs/edges. The distance between two vertices is the minimum weight of a path between them. For example, in a BFS in an unweighted graph, vertices are visited in order of their distance from the start.

**Example.** In the example graph below, the distance from $A$ to $E$ is 7 (via vertices $B$ and $D$):

![Graph with distances](image)

### 22.5 Dijkstra’s Algorithm

Dijkstra’s algorithm determines the distance from a start vertex to all other vertices. The idea is to

*Determine distances in increasing distance from the start.*

For each vertex, maintain $\text{dist}$ giving minimum weight of path to it found so far. Each iteration, choose a vertex of minimum $\text{dist}$, finalize it and update all $\text{dist}$ values.

Algorithm: Dijkstra (start):

1. initialise $\text{dist}$ for each vertex
2. while some vertex un-finalized {
   1. $v =$ un-finalized with minimum $\text{dist}$
   2. finalize $v$
   3. for all out-neighbors $w$ of $v$
      1. $\text{dist}(w) = \min(\text{dist}(w), \text{dist}(v)+\text{cost } v-w)$
}

70
If doing this by hand, one can set it out in a table. Each round, one circles the smallest value in an unfinalized column, and then updates the values in all other unfinalized columns.

**Example.** Here are the steps of Dijkstra’s algorithm on the graph of the previous page, starting at A.

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>13</td>
<td>12</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>9</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
</tbody>
</table>

Comments:

- Why Dijkstra works? Exercise.

- Implementation: store boolean array *known*. To get the actual shortest path, store Vertex array *prev*.

- The running time: simplest implementation gives a running time of $O(n^2)$. To speed up, use a priority queue that supports `decreaseKey`.