Algorithms

P. Danziger

1 Algorithms on Graphs

We will see a number of algorithms relating to graphs, generally speaking these fall into two broad categories, Depth First Search (DFS) or Breadth First Search (BFS).
Depth First Search and Breadth First Search are essentially methods of ordering the vertices of a graph. In a real algorithm the vertices would be processed in the corresponding order, with a call to the processing routine before moving to the next vertex. In both cases we have the notion of a *live* vertex as one which has not been processed yet. A vertex which is not live is called *used*, or *visited*.
In a Breadth First Search, we label all live nieghbours of a vertex before moving on to a new vertex. In a Depth First Search we move from vertex to live vertex. Depth First Search is a recursive algorithm, Breadth First is iterative using a stack.
In both cases we start from an *initial vertex*, which may be chosen randomly. Though the order in which the graph is traversed will depend on the choice of the initial vertex, the algorithm does not. Both algorithms work on connected components of the graph, the algorithm must be restarted with a new initial vertex for each connected component. We often assume that the graph is connected, with the understanding that the procedure is to be run on each connected component.
In giving an algorithm give all the initial inputs and the action of the algorithm. Most algorithms consist of an *Initialisation* phase in which variables are set up, and a *Recursive* or *Iterative* stage which recurses through the graph, usually either as a BFS or a DFS.
Another idea in algorithms is that of a Greedy Algorithm. In a Greedy Algorithm we make the locally optimal choice in the hope that this will lead to a globally oprtimal solution. In most cases this approach does not work, but it is always worth considering.
In another category entirely we have *Random Algorithms*. These algorithms have a random element at each step. The benefit of random algorithms is that they can solve much harder problems than deterministic algorithms. The downside is that either there is a chance that an incorrect solution will be given or a chance that the algorithm never returns. Examples of random algorithms are Hill climbs, Simulated Annealing and Genetic algorithms.

2 Depth First Search

2.1 Depth First Search

In this example of DFS, each vertex v is assigned and index $D(v)$. D is the order in which each vertex was visited during the algorithm and is called the depth first index of the vertex.
Input is a graph G.

Initialization:
for all $v \in V(G)$, $D(v) = 0$.

$F = \emptyset$
$i = 1$

Recursion:
DFS(v)
$D(v) = i$
$i = i + i$
For all $u \in N(v)$
if $D(u) == 0$ then
$F = F \cup \{v, u\}$
DFS(u)
end if
end for

Call: while there is $v \in V$ for which $D(v) = 0$
DFS(v)

F contains a spanning subtree of the graph. The algorithm visits each edge of the graph once, however the initialisation take n steps, so the running time is $O(\max(n, m))$.

2.2 Dijkstra's Algotrithm

A practical application of DFS is found in Dijksatra's algorithm for finding paths from a given vertex in a wieghted graph G. The idea is that we are given an initial vertex v and we wish to find the length of the shortest $uv-$path for every other vertex u in the graph. Note that the algorithm finds the length of the path, but does not actually give you a path.
The input is a weighted graph G, and a specified vertex $u \in V$. We define the function $w : E \longrightarrow \mathbb{Z}$ as follows:

$$w(u, v) = \begin{cases} \text{The weight of the edge } u\,v & \text{if } u\,v \in E \\ 0 & \text{if } u = v \\ \infty & \text{if } u\,v \notin E \end{cases}$$

Each vertex is given a label $L(v)$, initially $L(v) = w(u, v)$, by the end of the algorithm $L(v)$ is the length of the shortest path from u to v. T is a list of used vertices.

Initialization:
for all $v \in V(G)$ $L(v) = w(u, v)$.
$T = \{u\}$

Iteration:
while $T \neq V$
find $v' \in T$ such that for all $v \in T$, $L(v') \leq L(v)$
$T = T \cup \{v'\}$
for all $v \notin T$
if $L(v) > L(v') + w(v, v')$ then
$L(v') = L(v') + w(v, v')$

```
            end if
        end for
    end while
```

The idea is that at each stage $L(v)$ represents the length of the shortest $uv-$path, passing only through verticies of T. Dijkstra's algorithm has up to n comparisons for each of the n iterations of the loop, givinga running time of $O(n^2)$.

3 Breadth First Search

3.1 Breadth First Search

As with Depth First Search, Breadth First Search assigns an index $B(v)$ to each vertex, representing the order in which the vertex v was visited. BFS makes use of a queue Q (FIFO buffer) to keep track of vertices waiting to be processed.

Initialization:
for all $v \in V(G)$, $B(v) = 0$.
$i = 1$
Pick a starting vertex u
$B(u) = i$
push u onto Q

Iteration:

```
while Q ≠ ∅
    pop w from Q
    for every v ∈ N(w)
        if B(v) == 0 then
            i = i + 1
            B(v) = i
            push v onto Q
        endif
    end for
end while
```

As with DFS the running time of BFS is $O(\max(n, m))$.

3.2 Bipartition Algorithm

An example of BFS is provided by the bipartition algorithm. I this case an index $A(v)$ determines which of the two parts the vertex v will be in. Input is a (di)graph G

Initialization:
for all $v \in V(G)$, $A(v) = -1$.
$i = 0$

Pick a starting vertex u
$A(u) = i$
push u onto Q

Iteration:
    while $Q \neq \emptyset$
        pop w from Q
        $i = (A(w) + 1) \bmod 2$
        for every $v \in N(w)$
            if $A(v) == A(w)$ then
                return FAIL - Not Bipartite
            else if $A(v) == -1$ then
                $A(v) = i$
                push v onto Q
            endif
        end for
    end while

Upon termination $C = \{v \in V \mid A(v) = 0\}$ and $D = \{v \in V \mid A(v) = 1\}$ form a bipartition of V. If G is not bipartite, the algorithm reports this fact.

3.3 Shortest Path

This BFS finds the shortest path between two specified vertices.
Input is a digraph G, and two vertices x and y.
The routine finds the shortest path from x to y in G.
The algorithm conducts a BFS of G using a FIFO queue Q.
For each vertex v in the search
    It assigns each vertex a distance from x $L(v)$.
    It also builds up the pre-adjacencey list $B(v) = \{u \in V(G) \mid (u\,v) \in E(G)\}$
Once y has been reached the BFS terminates and the path P is constructed backwards starting from y.
If there is no such path it returns CUT. In this case if $S = \{v \in V(G) \mid L(v) \geq 0\}$ then $[S, \overline{S}]$ is an x, y cut.

Shortest_Path(x, y, G) {
Initialization:
    $\forall v \in V(G)$ do $L(v) = -1,\ B(v) = \phi$ done
    $i = 0,\ L(x) = 0, Path =$ False
    $Q = (x)$ (Add x to Q)

Recursion (BFS):
    While Q is not empty {
        Remove u from the top of Q
        If $u = y$ break fi (is this ok?)

$i = L(u) + 1$
For each $v \in N(u)$ do {
Add u to the list $B(v)$
if $L(v) = -1$ then $L(v) = i$ and add v to Q
}
}
If $L(y) = -1$ then return CUT

Create Path P
Initialization:
$P = y$, $u = y$
Recursion:
While $u \neq x$ {
Find $v \in B(u)$ with $L(v) = L(u) - 1$
$P = vP$ (Add v to the head of P)
$u = v$
}
}

The running time is again $O(\max(n, m))$ and so this is better than Dijkstra's algorithm for sparse graphs.