Depth-first search, the problem

We want to visit all the graph nodes using a recursive process \texttt{dfs}(u) that has the following property:

during \texttt{dfs}(u) visit every node such that there exists a path from \texttt{u} to \texttt{v} on which no node was visited before \texttt{dfs}(u) started.

We will see that this property has useful implications for various problems defined in terms of relation \texttt{u} \rightarrow \texttt{v}.

We have to mark somehow the fact that a node was visited. We will have arrays for visiting numbers and finishing numbers, \texttt{VNum[n]}, \texttt{FNum}

\begin{itemize}
  \item \texttt{VNum[u]} == \texttt{n} if \texttt{dfs}(u) hasn’t started yet,
  \item \texttt{FNum[u]} == \texttt{n} if \texttt{dfs}(u) hasn’t finish yet.
\end{itemize}

// top-level process
\begin{verbatim}
int vcounter = 0;
int fcounter = 0;
int Num[n], VNum[n];
for (every node \texttt{u})
  VNum[u] = FNum[u] = n;
for (every node \texttt{u})
  if (VNum[u] == n)
    dfs(u);
\end{verbatim}

// recursive process
\begin{verbatim}
void dfs(int u)
{
  int v;
  VNum[u] = vcounter++;
  for (every \texttt{v} adjacent to \texttt{u})
    if (VNum[v] == n)
      dfs(v); // (\texttt{u},\texttt{v}) is a tree edge
  FNum[u] = fcounter++;}
\end{verbatim}
Topological numbering problem

A directed graph is acyclic if $u \rightarrow v$ implies that no $v \sim u$.

A topological numbering $T\text{Num}$ has the following property:

$$u \rightarrow v \text{ implies } T\text{Num}[u] < T\text{Num}[v]$$

We will show that $n - F\text{Num}[u]$ is a good topological numbering, unless the graph has cycles. Consider an edge $u \rightarrow v$. We have the following cases:

**Bef**  $dfs(v)$ started before $dfs(u)$. During $dfs(v)$ we visit only such nodes that $v \sim w$, hence $dfs(v)$ has to finish before $dfs(u)$ starts. Thus $F\text{Num}[v] < F\text{Num}[u]$.

**Dur**  $dfs(v)$ started during $dfs(u)$: then $dfs(v)$ must finish before, and thus $F\text{Num}[v] < F\text{Num}[u]$.

In the for loop of $dfs(u)$ we inspect $v$. If the call $dfs(v)$ did start before that time, we have case **Bef** or **Dur**. Otherwise we make $dfs(v)$ at that time, so we have case **Dur**.

Thus in every case $F\text{Num}[v] < F\text{Num}[u]$. 
Application: strongly connected components

We define relation $u < \sim \sim v$ as $u \sim \sim v \sim \sim v \sim \sim u$. One can see that this is an equivalence relation.

The equivalence classes of $\sim \sim$ are called **strongly connected components**.

We can simplify some graph problems if we consider nodes grouped in strongly connected components, in order indicated by their topological numbering.

Now we will describe how to correctly computed one of the strongly connected components: take $v_{\text{last}}$ with the maximum value of $\text{FNum}[u]$ and run $\text{dfsRev}(v_{\text{last}})$, which is depth first search in the reversed graph.

In reversed graph we have $v \rightarrow u$ for every $u \rightarrow v$ in the original graph. It is easy to see that the reversed graph has the same strongly connected components. **WHY?**

Let $C$ be the strongly connected component of $v_{\text{last}}$. It is easy to see that $\text{dfsRev}(v_{\text{last}})$ will visit all nodes of $C$. Suppose that it also visits another node, say, $w$. Then we have $w \sim \sim v_{\text{last}}$ (WHY?) but not $v_{\text{last}} \sim \sim w$. We will get a contradiction.

$\text{dfs}(w)$ could not start before $\text{dfs}(v_{\text{last}})$, otherwise $\text{dfs}(v_{\text{last}})$ would start during $\text{dfs}(w)$ and $\text{FNum}[w] > \text{FNum}[v_{\text{last}}]$;

$\text{dfs}(w)$ could not start during $\text{dfs}(v_{\text{last}})$ because $v_{\text{last}} \sim \sim w$ is not true;

$\text{dfs}(w)$ could not start after $\text{dfs}(v_{\text{last}})$ because then we would have $\text{FNum}[w] > \text{FNum}[v_{\text{last}}]$.

Conclusion: a node belongs to $C$ if and only if it is visited during $\text{dfsRev}(v_{\text{last}})$.

How to find the remaining strongly connected components? We can apply the same reasoning, so now we find the largest $\text{FNum}[u]$ among the nodes that do not belong to $C$, say, $\text{FNum}[u_{\text{next}}]$ and we run $\text{dfsRev}(u_{\text{next}})$, except that we have to ignore nodes of $C$, and we can continue in this manner.

This is achieved if we order nodes “topologically”, with decreasing $\text{FNum}[u]$, and we run the top-level process of $\text{dfsRev}$ in that order.
Application: biconnected components and articulation points

In an undirected graph, an articulation point is a node that is necessary to connect some other two nodes (u that is on every path from v to w).

An articulation point belongs to edges from different biconnected components, which is basically a maximal union of overlapping edge sets of simple cycles (equivalence classes of relation edge e and edge f belong to some simple cycle).

We can recognize the articulation points by computing a depth first search tree, number the nodes in pre-order fashion (those are visit numbers) and then computing for every subtree, what is the lowest number that can be “seen” from that subtree.

For a non-root node, we are checking if an edge is in the same biconnected component as the edge from the parent.

An edge to an already visited node has to lead to an ancestor, so-called back edge, and thus it is surely on the same cycle as the edge from the parent.

An edge to a child (a node not visited before) is has that property if and only if the subtree of that child can “see” an ancestor, i.e. a node with a lower number than the parent.

For a root node r, there is no edge from the parent so the test is different, and actually, simpler. If u is a child of r, then every edge from the subtree of u (inspected during dfs(u)) is on the same cycle as (r, u), so these edges are in the same biconnected component. Thus r is not an articulation point if it has but one child.

But if r has two children, u and v, then dfs(u) did not find v, so r is necessary on a path from u to v. Thus r is an articulation point.

We can implement these ideas by adding few lines to our depth first search.
// top-level process
int vcounter = 0;
int Children[n], Articulation[n], VNum[n], Low[n];
for (every node u)
    Children[u] = Articulation[u] = 0;
    VNum[u] = n;
for (every node u)
    if (VNum[u] == n) {
        dfs(u);
        Articulation[u] = Children[u] < 2;
    }

// recursive process
void dfs(int u)
{
    int v;
    Low[u] = VNum[u] = vcounter++;
    for (every v adjacent to u)
        if (VNum[v] == n) {
            dfs(v); // (u,v) is a tree edge
            if (Low[v] < Low[u])
                Low[u] = Low[v];
            if (Low[v] >= VNum[u])
                Articulation[u] = 1;
                Children[u]++;
        }
    else if (VNum[v] < Low[u])
        Low[u] = VNum[v];
}
**Reading before the midterm exam**

Chapters 2, 3, 4, the part of Chapter 5 that describes the depth first search and applications, posted lecture notes.

**Types of questions for the midterm exam**

1. Converting functions to a “nice form” and sorting by order of growth.
2. Applying Master theorem.
3. Showing on a supplied example that you understand an algorithm: mergesort, quicksort, finding strongly connected components, finding biconnected components, finding a maximum independent set, finding a convex hull. Examples will have less than 20 numbers or points.
4. Design and analyze a simple algorithm, e.g. find a maximal independent set (an independent set that provides a neighbor for every other node), find a median element in an array that consists of two sorted fragments etc.