Principles of Algorithmic Techniques
CS-E3190

Pekka Orponen

Department of Computer Science
Aalto University

Autumn 2017
Lecture 5: Graph algorithms I

- The depth-first search (DFS) algorithm
- Applications of depth-first search
Graph algorithms

- A very large number of efficient and nontrivial algorithms have been developed for graph problems such as:
  - Finding shortest paths between vertices
  - Constructing minimum-weight spanning trees
  - Determining strongly connected components
  - ... and many others we have not defined here

- We shall present a few of these at this course, as important examples of basic design paradigms:
  - Single-source shortest paths: breadth-first search (lecture 6)
  - All-pairs shortest paths: dynamic programming (lecture 9)
  - Minimum spanning trees: greedy algorithms (lecture 8)
  - Strongly connected components: depth-first search (lecture 5, next)
5.1 The depth-first search algorithm

- A fundamental task: **exploring** a graph, that is visiting all vertices reachable from a given start vertex.
- Two natural exploration orders:
  - **Depth-first search (DFS):** continue forward along edges until hitting a vertex previously visited; then backtrack.
  - **Breadth-first search (BFS):** explore all vertices at a given distance from start vertex; then increase distance.
- Deceptively naïve, but in fact quite subtle procedures.
- DFS discussed today, BFS in next lecture.
Comparison of DFS and BFS search orders

- A DFS exploration order from start vertex A: 
  $A \rightarrow B \rightarrow E \rightarrow I \rightarrow J \rightarrow F \rightarrow C \rightarrow D \rightarrow G \rightarrow H$

- A BFS exploration order from start vertex A: 
  $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \rightarrow G \rightarrow H \rightarrow I \rightarrow J$

- Vertices $K, L$ are unreachable from $A$, so a new pass of the exploration process is needed to cover them in a complete search.
DFS explore: the algorithm (1/2)

- Clearly some stack-like book-keeping is needed to keep track of the backtrack points in a DFS exploration.
- This is implemented conveniently through recursion.

Algorithm 1: The DFS explore algorithm

```plaintext
1 function EXPLORE(G, v);

Input: Graph G = (V, E), start vertex v
Output: visited[u] = true for all vertices u reachable from v

2 visited[v] ← true;
3 previsit(v);
4 for each edge {v, u} ∈ E do
5    if ¬visited[u] then Explore(u);
6 end
7 postvisit(v)
```
In the EXPLORE algorithm, procedures “previsit” and “postvisit” may contain optional operations to be performed when a vertex is first discovered and when it is left for the last time. These will turn out to be very useful.
DFS explore: correctness

Claim. For a given start vertex $v$, \textsc{Explore}(v) visits all vertices reachable from $v$.

Proof. To reach a contradiction, assume that there exists a vertex $u$ that is reachable from $v$ but not visited by \textsc{Explore}(v).

Because $u$ is reachable from $v$, there exists a path $P$ that starts at $v$ and ends at $u$.

Let $z$ be the furthest vertex along $P$ visited by \textsc{Explore}(v), and $w$ the vertex immediately following $z$ on $P$.

But then, by the definition of \textsc{Explore}, also $w$ should have been visited, a contradiction.
DFS explore: tree edges and back edges

- The figure above illustrates one possible DFS exploration of the previous graph from start vertex $A$.
- Solid lines mark edges which the algorithm “traverses”, leading to new vertices and further exploration. These are called tree edges. (Why?)
- Dotted lines mark edges which the algorithm checks only to notice that they lead to previously visited vertices. These are called back edges. (Can back edges cross branches?)
DFS search: complete algorithm

Since a graph $G$ may consist of several distinct components, for complete coverage of $G$ several exploration passes with distinct start vertices need to be run.

Algorithm 2: The depth-first search algorithm

1. **function DFS** ($G$);

   **Input:** Graph $G = (V, E)$
   **Output:** visited[$v$] = true for all vertices $v \in V$

2. **for all** $v \in V$ **do** visited[$v$] $\leftarrow$ false;

3. **for all** $v \in V$ **do**
   
   4. **if** $\neg$visited[$v$] **then** Explore($v$);

5. **end**
DFS search: complexity

During a DFS search process of a graph $G = (V, E)$:

- For each vertex $v \in V$, \texttt{EXPLORE}(v) is called exactly \textit{once}. (Because of the visited[$v$] marks.)
- Each edge $\{v, u\} \in E$ is examined exactly \textit{twice}: once during \texttt{EXPLORE}(v) and once during \texttt{EXPLORE}(u).

Hence, assuming the previsit and postvisit operations are constant-time, a full DFS search of $G$ requires time $O(|V| + |E|)$, i.e. \textit{linear} in the size of the adjacency list representation of $G$. 
DFS search: example

In general, a DFS search of a graph induces a forest of DFS search trees, one per each connected component of the graph.
Previsit and postvisit timestamps

- In the previous figure of a search forest, each vertex $v$ was labelled according to the “event times” of when it was first entered and last exited in the search: $(\text{pre}[v], \text{post}[v])$.
- These important timestamps can be generated using the previsit and postvisit code blocks.
- Assign to variable “clock” value 1 at the beginning of a DFS search, and define:
  - $\text{previsit}(v) = \{\text{pre}[v] \leftarrow \text{clock}; \text{clock} \leftarrow \text{clock} + 1\}$
  - $\text{postvisit}(v) = \{\text{post}[v] \leftarrow \text{clock}; \text{clock} \leftarrow \text{clock} + 1\}$

Claim. For any vertices $u$ and $v$, the intervals $(\text{pre}[u], \text{post}[u])$ and $(\text{pre}[v], \text{post}[v])$ are either disjoint or one is contained within the other.
DFS in directed graphs

DFS search works exactly the same way also in directed graphs, but since the edge orientation needs to be followed, the resulting search trees\(^1\) have a somewhat richer structure.

\(^1\)A directed graph whose underlying undirected graph is a tree and where every vertex is reachable from a unique root vertex is called an arborescence.
DFS in digraphs: edge types

- **Tree edges** are the ones the search actually traverses. They comprise the search forest proper.
- **Forward edges** lead from a vertex to one of its *nonchild* descendants.
- **Back edges** lead from a vertex to one of its ancestors.
- **Cross edges** lead from a vertex to another vertex which is neither its descendant nor ancestor.

Think how the different types of edges emerge in the search process. Do the cross edges always point “backwards”? 
DFS in digraphs: edge types and pre/post timestamps

- In a DFS search forest, vertex $u$ is an ancestor of vertex $v$ if and only if $u$ is discovered first in the search process, and $v$ during the execution of $\text{EXPLORE}(u)$, i.e. if $\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u)$
- This can be represented in terms of “matching brackets” of the sequence of events as: $[u \ [v \ ]_v \ ]_u$
- One can then characterise all the edge types as follows:
5.2 Applications of depth-first search 1: Linearisation of DAGs (1/3)

- DAG = directed acyclic graph
- Very common structure in applications: sequential dependencies, causalities, temporal orderings, . . .

**Claim.** A directed graph $G$ contains a cycle if and only if its DFS search forest contains a back edge.

**Proof.** The “if” direction is clear, so let us focus on the “only if” direction.

Assume $G$ contains a cycle $C : v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_k \rightarrow v_0$. Let $v_i$ be the first vertex on $C$ to be discovered in the search. Then all the other vertices on $C$ are descendants of $v_i$ in the search tree, and $v_{i-1} \rightarrow v_i$ is a back edge. (Or $v_k \rightarrow v_0$ if $i = 0$.)
Linearisation of DAGs (2/3)

- A linearisation or topological sort of a DAG is an ordering of its vertices so that all the edges are oriented forwards w.r.t. the ordering.
- Can all DAGs be linearised? What would be a good algorithm?
Claim. In a DFS timestamp labelling for a DAG, 
\[ \text{post}(u) > \text{post}(v) \] 
for every edge \((u, v)\).

Proof. The only edges \((u, v)\) with \(\text{post}(u) < \text{post}(v)\) in a DFS search tree are the back edges. A DAG has no back edges.

Consequently, we obtain a simple DAG linearisation algorithm:

- Perform a DFS search and obtain the timestamp labels.
- List vertices in order of decreasing \(\text{post}(v)\) timestamp.
5.4 Applications of depth-first search 2: Strongly connected components (1/5)

- In a directed graph $G$, two vertices $u$ and $v$ are strongly connected if there is a path that starts at $u$ and ends at $v$ and a path that starts at $v$ and ends at $u$.
- This is an equivalence relation that partitions the vertices of $G$ into equivalence classes, its strongly connected components (SCCs).
- The digraph $G$ can be reduced to a DAG of its SCCs, $\tilde{G}$. 
Determining the SCCs of a directed graph seems like quite a formidable problem. Nevertheless, it can be solved in linear time by a clever application of depth-first search!

Let us develop a few basic properties of DFS that we need.

**Property 1.** An execution of `EXPLORE(u)` will visit exactly all the vertices reachable from `u`.

Thus, if `EXPLORE` is initiated at a vertex belonging to a sink (no outgoing edges) SCC of a digraph `G`, it will map out exactly that component. This can then be removed from `G` and the process continued with the remaining graph `G'`.

But how to find a sink SCC in the first place?
Strongly connected components (3/5)

- Finding sink SCCs may be difficult, but finding source SCCs (no incoming edges) is easier.

- **Property 2.** The vertex $v$ that obtains the highest $\text{post}(v)$ timestamp in a DFS search of digraph $G$ is located in a source SCC of $G$.

- Property 2 follows from a more general fact:

- **Property 3.** If $C$ and $C'$ are SCCs of digraph $G$, and there is an edge from a vertex of $C$ to a vertex of $C'$, then the highest $\text{post}(v)$ timestamp in $C$ is bigger than the highest $\text{post}(v')$ timestamp in $C'$.

- Property 3 implies also that the DAG $\tilde{G}$ can be linearised in the order of decreasing $\text{post}(v)$ timestamps.
Strongly connected components (4/5)

- So now we know how to hit a source SCC in $G$, and how to order SCC’s from sources to sinks.
- But mapping out the vertices in SCCs requires that we start from sinks, not sources. What to do?
- Clever idea: reverse the graph! (i.e. change the directions of all the edges.)\(^2\) The new digraph is denoted $G^R$.
- Then all the SCCs stay the same (check this!), but sinks become sources and vice versa: more generally, the DAG $\tilde{G}$ gets reversed too.
- The decreasing-post($v$) linearisation of the DAG $\tilde{G}^R$ provides a sinks-to-sources ordering of the DAG $\tilde{G}$.

\(^2\)Think about how to do this in linear time in the adjacency list representation.
Digraphs $G$ and $\tilde{G}$, $G^R$ and $\tilde{G}^R$
Strongly connected components (5/5)

So here is a linear-time (!) algorithm for determining the strongly connected components of a digraph $G$:

1. Reverse all the edges in $G$, yielding the digraph $G^R$.
2. Run DFS on $G^R$, obtaining the post($v$) timestamps for all vertices $v$.
3. Set $k \leftarrow 1$.
4. Run EXPLORE($v$) in $G$ from the vertex $v$ that has the highest post($v$) timestamp in $G^R$, and has not yet been assigned to any SCC. Assign all vertices mapped out by the exploration into SCC $k$. Remove these vertices from $G$.
5. Set $k \leftarrow k + 1$ and repeat from step 4, until all vertices have been assigned to SCCs.