TDT4121 Introduction to algorithms

Assignment 3

Claudi Lleyda Moltó

September 2024

Directed acyclic graphs

Definition (DAG)

A *DAG* (directed acyclic graph) is a directed graph without cycles.

$$
\begin{tikzcd}
a \arrow[r] & b \arrow[r] \arrow[dr] & c \arrow[r] \arrow[d] \arrow[dr] & d \arrow[r] & g \\
 & & e \arrow[r] & f \arrow[ur] &
\end{tikzcd}
$$

Directed acyclic graphs

Definition (DAG)

A *DAG* (directed acyclic graph) is a directed graph without cycles.

$$
\begin{tikzcd}
a \arrow[r] & b \arrow[r] \arrow[dr] & c \arrow[r] \arrow[d] \arrow[dr] & d \arrow[r] & g \\
 & & e \arrow[r] & f \arrow[ur] &
\end{tikzcd}
$$

DAGs are useful to represent situations such as

- task dependencies
- causal structures
- genealogies

Graph traversal

- Graph traversal refers to the process of visiting each vertex in a graph.
- Such traversals are classified by the order in which the vertices are visited.

Graph traversal

- Graph traversal refers to the process of visiting each vertex in a graph.
- Such traversals are classified by the order in which the vertices are visited.

Breadth first search

Breadth-first search (BFS) visits the sibling vertices before visiting the child vertices, and a queue is used in the search process.

Depth first search

Depth-first search (DFS) visits the child vertices before visiting the sibling vertices, and a stack is used in the search process.

Breadth first search

```
function BFS(Node source)
    queue ← NewQueue()
    Enqueue(queue, source)
    source.parent ← NULL
    source.depth ← 0
    while ¬IsEmpty(queue) do
        node ← Dequeue(queue)
        for adj in node.successors do
            if adj.depth ≠ ∞ then
                Enqueue(queue, adj)
                adj.depth = node.depth + 1
                adj.parent = node
```

Depth first search

function DFS(Node source)
    $stack \leftarrow$ NewStack()
    Push($stack$, $source$)
    $source.parent \leftarrow$ **NULL**
    while $\neg$IsEmpty($stack$) **do**
        $node \leftarrow$ Pop($stack$)
        if $node.visited$ **then continue**
        $node.visited \leftarrow$ **TRUE**
        for adj **in** $node.successors$ **do**
            if $\neg adj.visited$ **then**
                Push($stack$, adj)
                $adj.parent \leftarrow node$

Topological orderings

- It is often useful to “sort” a DAG.
- Particularly for the dependencies example, where it could be interpreted as an order to complete tasks.
- Of course, DAGs will, more often that not, be non-linear.

Topological orderings

- It is often useful to “sort” a DAG.
- Particularly for the dependencies example, where it could be interpreted as an order to complete tasks.
- Of course, DAGs will, more often that not, be non-linear.

Definition (Topological ordering)

A *Topological ordering* of a DAG is a ordering of vertices such that for every directed edge (u, v), vertex u comes before v in the ordering.

Topological orderings

Theorem

A directed graph is a DAG if and only if it has a topological ordering.

- Conveniently enough, all DAGs have a topological ordering.
- And conversely, if we find a topological ordering for a directed graph, we know that it must be a DAG.

Topological orderings

Theorem

A directed graph is a DAG if and only if it has a topological ordering.

- Conveniently enough, all DAGs have a topological ordering.
- And conversely, if we find a topological ordering for a directed graph, we know that it must be a DAG.

How do we find a topological ordering?

- There is a recursive approach

  function FindTopologicalSorting(Graph graph)
  $\quad$ *node* $\leftarrow$ node in *graph* with no parent nodes
  $\quad$ *list* $\leftarrow$ FindTopologicalSorting($graph - node$)
  $\quad$ **return** $node + list$

- You can also modify DFS to find one for you.