CS624 - Analysis of Algorithms

Fall 2018 – Graph Walks

November 5, 2018
Graphs – Basic Definitions

- Graph $G = (V, E)$
- $V =$ set of vertices, $E =$ set of edges $\subseteq (V \times V)$
- Undirected graph: edge $(u, v) = (v, u)$; for all $v$, $(v, v) \notin E$ (No self loops.)
- Directed graph: $(u, v)$ is edge from $u$ to $v$, denoted as $u \to v$. Self loops are allowed.
- Weighted graph: each edge has an associated weight, given by a weight function $w : E \to \mathbb{R}$.
- Dense graph: $|E| \approx |V|^2$.
- Sparse: $|E| \ll |V|^2$.
- $|E| = O(|V|^2)$
Graphs – Basic Definitions

- If \((u, v) \in E\), then vertex \(v\) is adjacent to vertex \(u\).
- Adjacency relationship is symmetric if \(G\) is undirected, not necessarily so if \(G\) is directed.
- \(G\) is connected if there is a path between every pair of vertices.
- In this case \(|E| \geq |V| - 1\).
- Furthermore, if \(|E| = |V| - 1\), then \(G\) is a tree.
- Other definitions in Appendix B (B.4 and B.5) as needed.
Searching a graph: Systematically follow the edges of a graph to visit the vertices of the graph.

Used to discover the structure of a graph.

Standard graph-searching algorithms:

- Breadth-first Search (BFS).
- Depth-first Search (DFS).
Let $G$ be an undirected graph.

One way to represent a graph is by a set *adjacency lists*, one for each vertex.

For each vertex $v \in V$, we have a list $\text{Adj}[v]$ consisting of those vertices $u$ such that $(v, u) \in E$.

It is actually a set, but usually implemented as a list.

This representation works just as well for directed graphs.

In this case, the edge $(v, u)$ means the edge starting from $v$ and ending at $u$.

BFS scans the graph $G$, starting from some arbitrary node $s$.

The key mechanism in this algorithm is the use of a queue, denoted by $Q$. 
Algorithm 1 \( BFS(G, s) \)

1: for each vertex \( u \in V[G] \setminus s \) do
2: \hspace{1em} Color\([u]\) \( \leftarrow \) White
3: \hspace{1em} \( d[u] \leftarrow \infty \)
4: \hspace{1em} \( \pi[u] \leftarrow \text{nil} \)
5: end for
6: Color\([s]\) \( \leftarrow \) Gray
7: \( d[s] \leftarrow 0 \)
8: \( \pi[s] \leftarrow \text{nil} \)
9: \( Q \leftarrow \emptyset \)
10: Enqueue\((Q, s)\)
11: while \( Q \neq \emptyset \) do
12: \( u \leftarrow \text{Dequeue}(Q) \)
13: \hspace{1em} for each \( v \in \text{Adj}[u] \) do
14: \hspace{2em} if \( \text{Color}[v] == \text{White} \) then
15: \hspace{3em} Color\([v]\) \( \leftarrow \) Gray
16: \hspace{3em} \( d[v] \leftarrow d[u] + 1 \)
17: \hspace{3em} \( \pi[v] \leftarrow u \)
18: \hspace{3em} Mark the edge from \( \pi[v] \) to \( u \) as a “tree edge”.
19: \hspace{3em} Enqueue\((Q, v)\)
20: \hspace{2em} end if
21: end for
22: \( \text{Color}[u] \leftarrow \text{Black} \)
23: end while
The BFS Algorithm

- Expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.
- A vertex is “discovered” the first time it is encountered during the search.
- A vertex is “finished” if all vertices adjacent to it have been discovered.
- Colors the vertices to keep track of progress.
  - White – Undiscovered.
  - Gray – Discovered but not finished.
  - Black – Finished.
- Colors are required only to reason about the algorithm. Can be implemented without colors.
- Note that all nodes are initially colored white.
- A node is colored gray when it is placed on the queue.
A node is colored black when taken off the queue.

Nodes colored white have not yet been visited. The nodes colored black are “finished” and the nodes colored gray are still being processed.
When a node is placed on the queue, the edge from the first node in the queue (which is being taken off the queue) to that node is marked as a *tree edge* in the breadth-first tree.

These edges actually do form a tree (called the breadth-first tree) whose root is the start node $s$. 
The BFS Algorithm – Example
Each node is visited once and each edge is examined at most twice.

Therefore the cost is $O(|V| + |E|)$.

Proof of correctness:

**Lemma**

*If $G$ is connected, then the breadth-first tree constructed by this algorithm*

- Really is a tree
- It contains all the nodes in the graph.
Proof.

- A node becomes the target of a tree edge when it is placed on the queue.
- Since that only happens once, no node is the target of two tree edges.
- Next, let us show that every node that is processed by the algorithm is reachable by a chain of tree edges from the root. It is enough to prove the following statement:
  - When a node is placed on the queue, it is reachable by a chain of tree edges from the root.
  - It is clearly true at the beginning: There is only one node in the queue and it is the root. The rest can be shown by induction.
Suppose it is true up to some point.

When the next node $v$ is placed on the queue, $v$ is an endpoint of an edge whose other endpoint is the node at the head of the queue, and that edge is made a tree edge.

By the inductive assumption, the node at the head of the queue is reachable by a path of tree edges from the root.

Appending the new edge to the path gives a path of tree edges from the root to $v$. 
Every node that is processed by the algorithm is reachable by a chain of edges from the root – so the edges form a tree.

Suppose there was one node \( v \) that was not reached by this process.

Since \( G \) is connected, there would have to be a path from the root to \( v \).

On that path there is a *first* node \( (w) \) which was not in the tree.

That node might be \( v \), or it might come earlier in the path.

That means that the edge in the path leading to that node starts from a node in the tree.

At some point, that node in the tree was at the head of the queue.

Therefore, \( w \) would have been placed in the queue by the algorithm, and the edge to \( w \) would have been a tree edge – a contradiction.
Lemma

If at any point in the execution of the BFS algorithm the queue consists of the vertices \( \{v_1, v_2, \ldots, v_n\} \), where \( v_1 \) is at the head of the queue, then \( d[v_i] \leq d[v_{i+1}] \) for \( 1 \leq i \leq n - 1 \), and \( d[v_n] \leq d[v_1] + 1 \).

- In other words, the assigned depth numbers increase as one walks down the queue, and there are at most two different depths in the queue at any one time.
- If there are two, they are consecutive.
The BFS Algorithm – Proof of Correctness

Proof.

- The result is true trivially at the start of the program, since there is only one element in the queue. The rest by induction.

- At any step, a vertex is added to the tail of the queue only when it is reachable from the vertex at the head (which is being taken off).

- The depth assigned to the new vertex at the tail is 1 more than that of the vertex at the head.

- By the inductive hypothesis it is greater than or equal to the depths of any other vertex on the queue, and no more than 1 greater than any of them.
The BFS Algorithm – Proof of Correctness

Lemma

If two nodes in G are joined by an edge in the graph (which might or might not be a tree edge), their d values differ by at most 1.

Proof.

- Let the nodes be v and u. One of them is reached first in the breadth-first walk.
- w.l.o.g, say v is reached first. So v is put on the queue first, and reaches the head of the queue before u does. When v reaches the head of the queue, there are two possibilities:
  - u has not yet been reached. In that case, when we take v off the queue, since there is an edge from v to u, u will be put on the queue and we will have \( d[u] = d[v] + 1 \).
  - u has been reached and therefore is on the queue. In this case, we know from the previous lemma that \( d[v] \leq d[u] \leq d[v] + 1 \).
The BFS Algorithm – Proof of Correctness

**Theorem**

*If G is connected, then the breadth-first search tree gives the shortest path from the root to any node.*

**Proof.**

- We know there is a path in the tree from the root to any node.
- The depth of any node in the tree is the length of the path in the tree from the root to that node.
- So for each node $v$ in the tree, we have

  \[ d[v] = \text{the length of the path in the tree from the root to } v \]

  and let us set

  \[ s[v] = \text{the length of the shortest path in } G \text{ from the root to } v \]
We are trying to prove that $d[v] = s[v]$ for all $v \in G$.

We know just by the definition of $s[v]$ that $s[v] \leq d[v]$ for all $v$.

Suppose there is at least one node for which the theorem is not true.

All the nodes $w$ for which the statement of the theorem is not true satisfy $s[w] < d[w]$.

Among all those nodes, pick one – call it $v$ – for which $s[v]$ is smallest.
Cont.

- Let $u$ be the node preceding $v$ on a shortest path from the root to $v$.
- We have

\[
\begin{align*}
d[v] &> s[v] \\
s[v] &= s[u] + 1 \\
s[u] &= d[u]
\end{align*}
\]

- But by former lemma, this is impossible.
We assume that $BFS(G, s)$ has already been run, so that each node $x$ has been assigned its depth $d[x]$.

**Algorithm 2** \textit{PrintPath}(G, s, v)

1: \textbf{if} $v = s$ \textbf{then}
2: \hspace{1em} \textit{PRINT} $s$
3: \textbf{else}
4: \hspace{1em} \textbf{if} $\pi[v] = \text{nil}$ \textbf{then}
5: \hspace{2em} \textit{PRINT} “no path from” $s$ “to” $v$ “exists”
6: \hspace{1em} \textbf{else}
7: \hspace{2em} \textit{PrintPath}(G, s, \pi[v])
8: \hspace{1em} \text{PRINT} $v$
9: \hspace{1em} \textbf{end if}
10: \textbf{end if}

The cost of this algorithm is proportional to the number of vertices in the path, so it is $O(d[v])$. 
Depth-First Search (DFS)

- **Input:** $G = (V, E)$, directed or undirected. No source vertex given!
- **Output:** 2 timestamps on each vertex. Integers between 1 and $2|V|$.
  - $d[v] =$ discovery time ($v$ turns from white to gray)
  - $f[v] = $ finishing time ($v$ turns from gray to black)
  - $\pi[v] = $ predecessor of $v$. A vertex $u$ such that $v$ was discovered during the scan of $u$ adjacency list.
- Uses the same coloring scheme for vertices as BFS.
Algorithm 3 $DFS(G)$

1: for each $u \in V[G]$ do
2: \hspace{1em} $color[u] \leftarrow \text{white}$
3: \hspace{1em} $\pi[u] \leftarrow \text{NIL}$
4: \hspace{1em} end for
5: $time \leftarrow 0$
6: for each $u \in V[G]$ do
7: \hspace{1em} if $color[u] == \text{white}$ then
8: \hspace{2em} $DFS - \text{Visit}(u)$
9: \hspace{1em} end if
10: end for

Algorithm 4 $DFS - \text{Visit}(u)$

1: $color[u] \leftarrow \text{GRAY}$
2: $time \leftarrow time + 1$
3: $d[u] \leftarrow time$
4: for each $v \in Adj[u]$ do
5: \hspace{1em} if $color[v] == \text{WHITE}$ then
6: \hspace{2em} $\pi[v] \leftarrow u$
7: \hspace{1em} $DFS - \text{Visit}(v)$
8: \hspace{1em} end if
9: end for
10: $color[u] \leftarrow \text{BLACK}$
11: $f[u].time \leftarrow time + + + 1$
The DFS Algorithm – Example

Nurit Haspel
CS624 - Analysis of Algorithms
The DFS Algorithm – Example
The DFS Algorithm – Example
The loops on lines 1-2 & 5-7 take $\Theta(V)$ time, excluding time to execute DFS-Visit.

DFS-Visit is called once for each white vertex $v \in V$ when its painted gray the first time.

Lines 3-6 of DFS-Visit is executed —$\text{Adj}[v]$— times. The total cost of executing DFS-Visit is $\sum_{v \in V} |\text{Adj}[v]| = \Theta(E)$.

Total running time of DFS is $\Theta(V + E)$. 
The Parenthesis Theorem

Theorem

For all $u$, $v$, exactly one of the following holds:

2. $d[u] < d[v] < f[v] < f[u]$ and $v$ is a descendant of $u$.


OK: ( ) [ ] ( [ ] ) [ ( ) ]
Not OK: ( [ ) ] [ ( ) ]

Corollary

$v$ is a proper descendant of $u$ iff $d[u] < d[v] < f[v] < f[u]$. 
The Parenthesis Theorem

Proof.

- If $\text{start}[x] < \text{start}[y] < \text{finish}[x]$ then $x$ is on the stack when $y$ is first reached.
- Therefore the processing of $y$ starts while $x$ is on the stack, and so it also must finish while $x$ is on the stack:
- we have $\text{start}[x] < \text{start}[y] < \text{finish}[y] < \text{finish}[x]$.
- The case when $\text{start}[y] < \text{start}[x] < \text{finish}[y]$ is handled in the same way.
- Another way to state the parenthesis nesting property is that given any two nodes $x$ and $y$, the intervals $[\text{start}[x], \text{finish}[x]]$ and $[\text{start}[y], \text{finish}[y]]$ must be either nested or disjoint.
The Parenthesis Theorem – Example

\[(s (z (y (x x) y) (w w) z) s) (t (v v) (u u) t))\]
Depth First Trees

- Predecessor subgraph defined slightly different from that of BFS.
- The predecessor subgraph of DFS is $G_\pi = (V, E_\pi)$ where $E_\pi = \{(\pi[v], v) : v \in V$ and $\pi[v] \neq NIL\}$.
- How does it differ from that of BFS?
- The predecessor subgraph G forms a depth-first forest composed of several depth-first trees.
- The edges in E are called tree edges.

Definition (Forest)

An acyclic graph G that may be disconnected.
**Theorem**

\( v \) is a tree descendant of \( u \) if and only if at time \( d[u] \), there is a path \( u \leadsto v \) consisting of only white vertices (Except for \( u \), which was just colored gray.)

**Proof.**

**One direction:** (if \( v \) is a tree descendant of \( u \) then there is a white path \( u \leadsto v \) at time \( d[u] \)) is obvious from the definition of a tree descendant (see the parenthesis theorem).
Is it possible that \( v \) is not a descendant of \( u \) in the DFS forest?

By induction on all the vertices along the path: Of course \( u \) is a descendant of itself.

Let us pick any vertex \( p \) on the path other than the first vertex \( u \), and let \( q \) be the previous vertex on the path [so it can be that \( q \) is \( u \)].

We assume that all vertices along the path from \( u \) to \( q \) inclusive are descendants of \( u \) (inductive hypothesis).

We will argue that \( p \) is also a descendant of \( u \).
White Path Theorem

Cont. – Reverse Direction.

- At time $d[u]$ vertex $p$ is white [by assumption about the white path], So $d[u] < d[p]$.
- But there is an edge from $q$ to $p$, so $q$ must explore this edge before finishing.
- At the time when the edge is explored, $p$ can be:
  - **WHITE**, then $p$ becomes a descendant of $q$, and so of $u$.
  - **BLACK**, then $f[p] < f[q]$ [because $f[p]$ must have already been assigned by that time, while $f[q]$ will get assigned later].
  - But since $q$ is a descendant of $u$ [not necessarily proper], $f[q] \leq f[u]$, we have $d[u] < d[p] < f[p] < f[q] \leq f[u]$, and we can use the Parenthesis theorem to conclude that $p$ is a descendant of $u$. 
Cont. – Reverse Direction.

- **GRAY**, then $p$ is already discovered, while $q$ is not yet finished, so $d[p] < f[q]$.

- Since $q$ is a descendant of $u$ [not necessarily proper], by the Parenthesis theorem, $f[q] \leq f[u]$.

- Hence $d[u] < d[p] < f[q] \leq f[u]$. So $d[p]$ belongs to the set \{d[u], \ldots, f[u]\}, and so we can use the the Parenthesis theorem again to conclude that $p$ must be a descendant of $u$.

- The conclusion thus far is that $p$ is a descendant of $u$. Now, as long as there is a vertex on the remainder of the path from $p$ to $v$, we can repeatedly apply the inductive argument, and finally conclude that the vertex $v$ is a descendant of $u$, too.
Classiﬁcation of Edges

- **Tree edge:** in the depth-first forest. Found by exploring \((u, v)\).
- **Back edge:** \((u, v)\), where \(u\) is a descendant of \(v\) (in the depth-first tree).
- **Forward edge:** \((u, v)\), where \(v\) is a descendant of \(u\), but not a tree edge.
- **Cross edge:** any other edge. Can go between vertices in same depth-first tree or in different depth-first trees.
- Edge type for edge \((u, v)\) can be identiﬁed when it is ﬁrst explored by DFS based on the color of \(v\).
The edge $x \rightarrow z$ will be discovered when exploring $x$, hence it’s a back edge.
In DFS of an undirected graph, we get only tree and back edges. No forward or cross edges.

Starting from 1, either 2 discovers 3 or vice versa, therefore one of them is the other's descendant, Hence no cross edges.
Directed Acyclic Graph (DAG)

- DAG – Directed graph with no cycles.
- Good for modeling processes and structures that have a partial order:
  - \(a > b\) and \(b > c\) \(\Rightarrow\) \(a > c\).
- But may have a and b such that neither \(a > b\) nor \(b > a\).
- Can always make a total order (either \(a > b\) or \(b > a\) for all \(a \neq b\)) from a partial order.
Lemma

A directed graph $G$ is acyclic iff a DFS of $G$ yields no back edges.

Proof.

⇒ Show that back edge → cycle:
Suppose there is a back edge $(u, v)$. Then $v$ is ancestor of $u$ in depth-first forest. Therefore, there is a path $v \leadsto u$, so $v \leadsto u \leadsto v$ is a cycle.
Proof.

⇒: Show that a cycle implies a back edge.

- $c$: cycle in $G$, $u$: first vertex discovered in $c$, $(v, u)$: preceding edge in $c$.
- At time $d[v]$, vertices of $c$ form a white path $u \leadsto v$. Why?
- By white-path theorem, $v$ is a descendent of $u$ in depth-first forest.
- Therefore, $(v, u)$ is a back edge.

Diagram:

```
       T
      /    \
   T     T
  /      \
 u ------ T ------ v
     \    /           \    /
      B
```
Topological Sorting

- We want to “sort” a DAG.
- Think of original DAG as a partial order.
- We want a total order that extends this partial order.
Performed on a DAG.
Linear ordering of the vertices of $G$ such that if $(u, v) \in E$, then $u$ appears somewhere before $v$.

**TopologicalSort(G)**

1. call DFS($G$) to compute finishing times $f[v]$ for all $v \in V$
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices

Runtime – $\Theta(V + E)$
Topological Sorting – Example

Linked list:

Linked list:
Topological Sorting – Example

Linked list:

2/3
E

Linked list:

1/4 → 2/3
D → E
Topological Sorting – Example

Graph:

A → B
C → B

Linked list:
1/4 → 2/3

A → B
D → E

Linked list:
1/4 → 2/3

A → B
D → E

Linked list:
Topological Sorting – Example

Linked list:

\[
\begin{align*}
6/7 & \longrightarrow 1/4 & \longrightarrow 2/3 \\
C & \longrightarrow D & \longrightarrow E
\end{align*}
\]

Linked list:

\[
\begin{align*}
5/8 & \longrightarrow 6/7 & \longrightarrow 1/4 & \longrightarrow 2/3 \\
B & \longrightarrow C & \longrightarrow D & \longrightarrow E
\end{align*}
\]
Topological Sorting – Example

Linked list:

5/8 → 6/7 → 1/4 → 2/3

Linked list:

9/10 → 5/8 → 6/7 → 1/4 → 2/3
Topological Sorting – Proof of Correctness

- Just need to show if \((u, v) \in E\), then \(f[v] < f[u]\).
- When we explore \((u, v)\) then \(u\) is gray. What is the color of \(v\)?
  - Is \(v\) **gray**?
  - No, because then \(v\) would be ancestor of \(u\). \(\Rightarrow (u, v)\) is a back edge, which contradicts the fact that A DAG has no back edges.
- Is \(v\) **white**?
  - Then becomes descendant of \(u\).
  - By parenthesis theorem, \(d[u] < d[v] < f[v] < f[u]\).
- Is \(v\) **black**?
  - Then \(v\) is already finished.
  - Since were exploring \((u, v)\), we have not yet finished \(u\).
  - Therefore, \(f[v] < f[u]\).