CMSC 221 - Fall 2017 - Homework 10

Jory Denny

March 19, 1991

1. Suppose we represent a graph $G$ having $n$ vertices and $m$ edges with the edge list structure. Why, in this case, does the `insertVertex` method run in $O(1)$ time while the `removeVertex` method runs in $O(m)$ time? Statement and proof of complexity required.

**Theorem 1.** When storing a graph as an edge list, the `insertVertex` takes $O(1)$ time.

**Proof.** Inserting a vertex is equivalent to adding an element to the end of a List. This can easily be done in $O(1)$ time with a linked list or an array list structure. \[\square\]

**Theorem 2.** When storing a graph as an edge list, the `removeVertex` takes $O(m)$ time.

**Proof.** In order to remove a vertex from a graph, we also must remove all edges with a source or target as the vertex. This in an edge list requires searching the whole list of edges for incident edges. This operation thus take $O(m)$ time. \[\square\]

2. Suppose we wish to represent an $n$-vertex graph $G$ using the edge list structure, assuming that we identify the vertices with the integers in the set $\{0, 1, \ldots, n-1\}$. Describe how to implement the collection $E$ to support $O(\log n)$-time performance for the `getEdge(u, v)` method. How are you implementing the method in this case?

Store $E$ as a map of map between the source vertex and target vertex using balanced binary search trees. Looking up the source vertex takes $O(\log n^2)$ time, and looking up the destination vertex takes $O(\log n)$ time, as each vertex has at most $n$ edges. So in total, a search would take $O(\log n)$ time.
3. Write and analyze a method, \texttt{components}(G), for an undirected graph \(G\), that returns a dictionary mapping each vertex to an integer that serves as an identifier for its connected component. That is, two vertices should be mapped to the same identifier if and only if they are in the same connected component.

\textbf{Algorithm.}

The algorithm (Algorithm 1) essentially performs a breadth first search over the entire graph. When a new component is encountered a label is given to the first vertex and then the rest of the component is labelled in a breadth-first-search manner (Algorithm 2).

\begin{algorithm}[h]
\caption{\texttt{components}(G)}
\begin{algorithmic}[1]
\State Label \texttt{cc} $\leftarrow 0$
\State Map \texttt{ccs} $\leftarrow \emptyset$
\ForAll {\(v \in V\)}
\If {\texttt{ccs}.get\((v) = \emptyset\)}
\State \texttt{labelComponent}(\(G, v, \texttt{ccs}, \texttt{cc}\))
\State \texttt{cc} $\leftarrow \texttt{cc} + 1$
\EndIf
\EndFor
\State \Return \texttt{ccs}
\end{algorithmic}
\end{algorithm}

\begin{algorithm}[h]
\caption{\texttt{labelComponents}(G, s, \texttt{ccs}, \texttt{cc})}
\begin{algorithmic}[1]
\State Queue \texttt{Q} $\leftarrow \emptyset$
\State \texttt{Q}.enqueue\((s)\)
\While {\neg \texttt{Q}.isEmpty()}
\State \texttt{v} $\leftarrow \texttt{Q}.dequeue()$
\ForAll {e $\in G$.outgoingEdges\((v)\)}
\State \(u \leftarrow G$.opposite\((e, v)\)
\If {\texttt{ccs}.get\((u) = \emptyset\)}
\State \texttt{ccs}.put\((u, \texttt{cc}\))
\State \texttt{q}.enqueue\((u)\)
\EndIf
\EndFor
\EndWhile
\end{algorithmic}
\end{algorithm}

\textbf{Time Complexity.}

\textbf{Theorem 3.} Algorithm \[\texttt{labelComponents}\] \texttt{components} \textit{runs in} \(O(n + m)\) \textit{time on an undirected graph.}

\textit{Proof.} Let the map data structure be implemented as a resizable hash map and the queue data structure to be implemented as a linked list. All labeling and queue operations can be completed in \(O(1)\) time. The overall algorithm will only visit each vertex and label it one time, totalling \(O(n)\). The algorithm must visit each incident edge of all vertices, totalling \(O(\Sigma_{v \in V} deg(v)) = O(m)\). Thus in total the algorithm takes \(O(n + m)\) time. \hfill \(\Box\)

\textbf{Memory Complexity.}

\textbf{Theorem 4.} Algorithm \[\texttt{labelComponents}\] \texttt{components} \textit{uses} \(O(n)\) \textit{additional memory.}

\textit{Proof.} The queue and map data structures can contain a maximum of \(O(n)\) elements in them. There is no other cause for extra memory usage, as the algorithm is non-recursive. \hfill \(\Box\)
Note: Depth-first-search may also be used.
4. A graph \( G \) is bipartite if its vertices can be partitioned into two sets \( X \) and \( Y \) such that every edge in \( G \) has one end vertex in \( X \) and the other in \( Y \). Design and analyze an efficient algorithm for determining if an undirected graph \( G \) is bipartite (without knowing the sets \( X \) and \( Y \) in advance).

**Algorithm.**

The algorithm (Algorithm 3) starts by running a specialized version of breadth first search that colors each vertex upon visiting. Let all vertices be initialized to GRAY. The source, level 0 of the BFS, is labeled WHITE. Then, the BFS alternates coloring vertices WHITE and BLACK such that even numbered levels are WHITE and odd numbered levels are BLACK. This 2-coloring of the graph indicates whether or not the graph is bipartite. After computing the coloring, a loop over the edges checks to see if both endpoints of all edges differ in color. If so, the graph is bipartite, false otherwise.

**Algorithm 3** \texttt{isBipartite}(\( G \))

\begin{algorithm}
\textbf{Input}: Undirected graph \( G = (V, E) \)
\textbf{Output}: \texttt{true} if \( G \) is bipartite, \texttt{false} otherwise
\begin{algorithmic}
1: \texttt{2ColorWithBFS}(\( G \))
2: for all \( e \in E \) do
3: \hspace{1em} if Colors of \( e \)'s endpoints differ then
4: \hspace{2em} return \texttt{false}
5: \hspace{1em} return \texttt{true}
\end{algorithmic}
\end{algorithm}

**Time Complexity.**

**Theorem 5.** Algorithm 3 runs in \( O(n + m) \) time on an undirected graph.

*Proof.* Let the coloring data structure be implemented as a resizable hash map. The time complexity of BFS to color the graph is the same as a typical BFS, i.e., \( O(n + m) \). The final loop to see if any colors differ will take \( O(m) \) time. In total, the algorithm takes \( O(n + m) \) time. \( \square \)

**Memory Complexity.**

**Theorem 6.** Algorithm 3 uses \( O(n) \) additional memory.

*Proof.* The coloring data structure to store the colors will take \( O(n) \) additional memory. The BFS also might have as many as \( O(n) \) nodes in a single level. In total, \( O(n) \) additional memory. \( \square \)
5. An Euler tour of a directed graph $G$ with $n$ vertices and $m$ edges is a cycle that traverses each edge of $G$ exactly once according to its direction. Such a tour always exists if $G$ is connected and the in-degree equals the out-degree of each vertex in $G$. Describe and analyze an $O(n+m)$-time algorithm for finding an Euler tour of such a directed graph $G$.

**Algorithm.**

The algorithm is described in Algorithm 4. Its basis is a depth first search. Because of the input, the depth first search can start from any vertex and be completed in one execution of the main loop. The modification is tracking a stack of a current cycle and adding it to the overall tour at the appropriate points (when a cycle is finished). The input is assumed to be a graph where in-degree is the same as out-degree for all nodes.

**Algorithm 4 Euler Tour**

**Input:** Directed Graph $G = (V, E)$

**Output:** Euler tour $T$ of vertices in cycle

1. $T ← ∅$
2. $G' = (V', E') ← G$
3. $v ← G'.\text{anyVertex}()$
4. Stack $S ← ∅$
5. repeat
6. if $G'.\text{outDegree}(v) = 0$ then
7. $T ← T ∪ \{v\}$
8. $v ← S.\text{pop}()$
9. else
10. $S.\text{push}(v)$
11. $e ← G'.\text{anyOutgoingEdge}(v)$
12. $v ← G'.\text{opposite}(e)$
13. $E' ← E' \setminus \{e\}$
14. until $G'.\text{outDegree}(v) = 0 ∧ S.\text{isEmpty}()$
15. return $T$

**Time Complexity.**

**Theorem 7.** Algorithm 4 runs in $O(n+m)$ time on a directed graph.

*Proof.* The overall algorithm will only visit each edges once. The removal of edges can actually be implemented as labeling or in a hash set. Each vertex will be visited at least once because the graph is connected – $O(n)$. The algorithm must visit each incident edge of all vertices, totalling $O(\sum_{v \in V} \text{deg}(v)) = O(m)$. Thus in total the algorithm takes $O(n+m)$ time. □

**Memory Complexity.**

**Theorem 8.** Algorithm 4 uses $O(n+m)$ additional memory.

*Proof.* The stack will contain at most $O(n)$ nodes and the tour will contain $O(m)$ vertices because of the reasoning in the above proof. Thus in total there is $O(n+m)$ memory usage. □

**Note:** There are many ways to define and write this algorithm.
6. **Bonus.** An independent set of an undirected graph \( G = (V, E) \) is a subset \( I \) of \( V \) such that no two vertices in \( I \) are adjacent. That is, if \( u \) and \( v \) are in \( I \), then \( (u, v) \) is not in \( E \). A maximal independent set \( M \) is an independent set such that, if we were to add any additional vertex to \( M \), then it would not be independent any more. Every graph has a maximal independent set. (Can you see this? This question is not part of the exercise, but it is worth thinking about.) Give and analyze an efficient algorithm that computes a maximal independent set for a graph \( G \).

**Algorithm.**

The algorithm is quite simplistic and is described in Algorithm 5. Essentially, pick a vertex and remove all adjacent vertices. Repeat this until there are no vertices left. Every time a vertex is chosen add it to the independent set.

---

**Algorithm 5 Maximal Independent Set**

**Input:** Graph \( G = (V, E) \)

**Output:** Maximal Independent Set \( I \subseteq V \)

1. \( I \leftarrow \emptyset \)
2. \( V' \leftarrow V \)
3. while \( \neg V'.\text{isEmpty()} \) do
4. \( v \leftarrow \text{anyVertex}(V') \)
5. \( I \leftarrow I \cup \{v\} \)
6. \( V' \leftarrow V' \setminus \{v\} \)
7. for all \( u \in G.\text{outgoingEdges}(v) \) do
8. \( V' \leftarrow V' \setminus \{u\} \)
9. return \( I \)

---

**Time Complexity.**

**Theorem 9.** Algorithm \( \Box \) runs in \( O(n) \) time, where \( n \) is the number of vertices of \( G \).

*Proof.** All vertices will be visited once. Let the removal procedure actually be implemented by labeling or a hash set mechanism for constant time operations. Total is \( O(n) \).

---

**Memory Complexity.**

**Theorem 10.** Algorithm \( \Box \) uses \( O(n) \) additional memory.

*Proof.** At most every node would contain a label and possibly be placed in \( I \). No other additional memory is used. Total is thus \( O(n) \).