10.1.2 Application: Counting Word Frequencies

As a case study for using a map, consider the problem of counting the number of occurrences of words in a document. This is a standard task when performing a statistical analysis of a document, for example, when categorizing an email or news article. A map is an ideal data structure to use here, for we can use words as keys and word counts as values. We show such an application in Code Fragment 10.1.

We break apart the original document using a combination of file and string methods that results in a loop over a lowercased version of all whitespace separated pieces of the document. We omit all nonalphabetic characters so that parentheses, apostrophes, and other such punctuation are not considered part of a word.

In terms of map operations, we begin with an empty Python dictionary named freq. During the first phase of the algorithm, we execute the command

```
freq[word] = 1 + freq.get(word, 0)
```

for each word occurrence. We use the get method on the right-hand side because the current word might not exist in the dictionary; the default value of 0 is appropriate in that case.

During the second phase of the algorithm, after the full document has been processed, we examine the contents of the frequency map, looping over freq.items( ) to determine which word has the most occurrences.

```
freq = { }
for piece in open(filename).read( ).lower( ).split( ):
  # only consider alphabetic characters within this piece
  word = ''.join(c for c in piece if c.isalpha( ))
  if word:      # require at least one alphabetic character
    freq[word] = 1 + freq.get(word, 0)

max_word = ''
max_count = 0
for (w,c) in freq.items( ):          # (key, value) tuples represent (word, count)
  if c > max_count:
    max_word = w
    max_count = c
print('The most frequent word is', max_word)
print('Its number of occurrences is', max_count)
```

Code Fragment 10.1: A program for counting word frequencies in a document, and reporting the most frequent word. We use Python's dict class for the map. We convert the input to lowercase and ignore any nonalphabetic characters.

14.2.5 Python Implementation

In this section, we provide an implementation of the Graph ADT. Our implementation will support directed or undirected graphs, but for ease of explanation, we first describe it in the context of an undirected graph.

We use a variant of the ***adjacency map*** representation. For each vertex v, we use a Python dictionary to represent the secondary incidence map $I(v)$. However, we do not explicitly maintain lists V and E, as originally described in the edge list representation. The list V is replaced by a top-level dictionary D that maps each vertex v to its incidence map $I(v)$; note that we can iterate through all vertices by generating the set of keys for dictionary D. By using such a dictionary D to map vertices to the secondary incidence maps, we need not maintain references to those incidence maps as part of the vertex structures. Also, a vertex does not need to explicitly maintain a reference to its position in D, because it can be determined in $O(1)$ expected time. This greatly simplifies our implementation. However, a consequence of our design is that some of the worst-case running time bounds for the graph ADT operations, given in Table 14.1, become ***expected*** bounds. Rather than maintain list E, we are content with taking the union of the edges found in the various incidence maps; technically, this runs in $O(n+m)$ time rather than strictly $O(m)$ time, as the dictionary D has n keys, even if some incidence maps are empty.

Our implementation of the graph ADT is given in Code Fragments 14.1 through 14.3. Classes Vertex and Edge, given in Code Fragment 14.1, are rather simple, and can be nested within the more complex Graph class. Note that we define the __hash__ method for both Vertex and Edge so that those instances can be used as keys in Python's hash-based sets and dictionaries. The rest of the Graph class is given in Code Fragments 14.2 and 14.3. Graphs are undirected by default, but can be declared as directed with an optional parameter to the constructor.

Internally, we manage the directed case by having two different top-level dictionary instances, _outgoing and _incoming, such that _outgoing[v] maps to another dictionary representing $I_{\text{out}}(v)$, and _incoming[v] maps to a representation of $I_{\text{in}}(v)$. In order to unify our treatment of directed and undirected graphs, we continue to use the _outgoing and _incoming identifiers in the undirected case, yet as aliases to the same dictionary. For convenience, we define a utility named is_directed to allow us to distinguish between the two cases.

For methods degree and incident_edges, which each accept an optional parameter to differentiate between the outgoing and incoming orientations, we choose the appropriate map before proceeding. For method insert_vertex, we always initialize _outgoing[v] to an empty dictionary for new vertex v. In the directed case, we independently initialize _incoming[v] as well. For the undirected case, that step is unnecessary as _outgoing and _incoming are aliases. We leave the implementations of methods remove_vertex and remove_edge as exercises (C-14.37 and C-14.38).

14.3.2 DFS Implementation and Extensions

We begin by providing a Python implementation of the basic depth-first search algorithm, originally described with pseudo-code in Code Fragment 14.4. Our DFS function is presented in Code Fragment 14.5.

```
def DFS(g, u, discovered):
  """Perform DFS of the undiscovered portion of Graph g starting at Vertex u.

  discovered is a dictionary mapping each vertex to the edge that was used to
  discover it during the DFS. (u should be "discovered" prior to the call.)
  Newly discovered vertices will be added to the dictionary as a result.
  """
  for e in g.incident_edges(u):          # for every outgoing edge from u
    v = e.opposite(u)
    if v not in discovered:              # v is an unvisited vertex
      discovered[v] = e                  # e is the tree edge that discovered v
      DFS(g, v, discovered)              # recursively explore from v
```

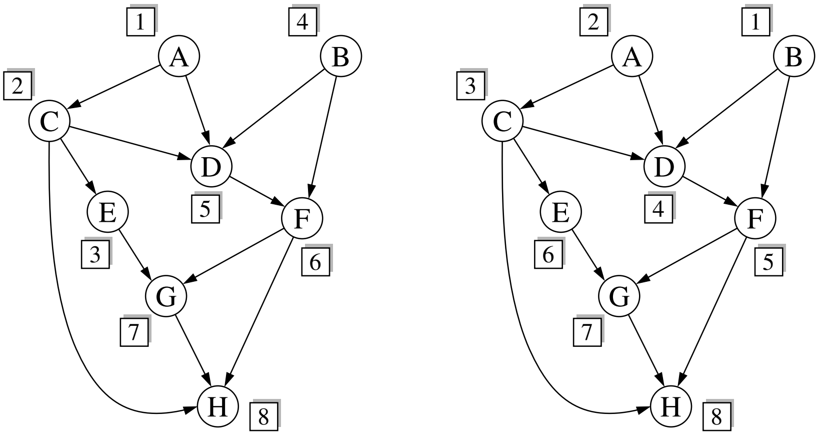
Code Fragment 14.5: Recursive implementation of depth-first search on a graph, starting at a designated vertex *u*.

In order to track which vertices have been visited, and to build a representation of the resulting DFS tree, our implementation introduces a third parameter, named discovered. This parameter should be a Python dictionary that maps a vertex of the graph to the tree edge that was used to discover that vertex. As a technicality, we assume that the source vertex *u* occurs as a key of the dictionary, with None as its value. Thus, a caller might start the traversal as follows:

```
result = {u : None}        # a new dictionary, with u trivially discovered
DFS(g, u, result)
```

The dictionary serves two purposes. Internally, the dictionary provides a mechanism for recognizing visited vertices, as they will appear as keys in the dictionary. Externally, the DFS function augments this dictionary as it proceeds, and thus the values within the dictionary are the DFS tree edges at the conclusion of the process.

Because the dictionary is hash-based, the test, "**if** v **not in** discovered," and the record-keeping step, "discovered[v] = e," run in $O(1)$ ***expected*** time, rather than worst-case time. In practice, this is a compromise we are willing to accept, but it does violate the formal analysis of the algorithm, as given on page 643. If we could assume that vertices could be numbered from 0 to $n-1$, then those numbers could be used as indices into an array-based lookup table rather than a hash-based map. Alternatively, we could store each vertex's discovery status and associated tree edge directly as part of the vertex instance.



(a) (b)

Figure 14.12: Two topological orderings of the same acyclic directed graph.

We now argue the sufficiency of the condition (the "if" part). Suppose $\vec{G}$ is acyclic. We will give an algorithmic description of how to build a topological ordering for $\vec{G}$. Since $\vec{G}$ is acyclic, $\vec{G}$ must have a vertex with no incoming edges (that is, with in-degree 0). Let v_1 be such a vertex. Indeed, if v_1 did not exist, then in tracing a directed path from an arbitrary start vertex, we would eventually encounter a previously visited vertex, thus contradicting the acyclicity of $\vec{G}$. If we remove v_1 from $\vec{G}$, together with its outgoing edges, the resulting directed graph is still acyclic. Hence, the resulting directed graph also has a vertex with no incoming edges, and we let v_2 be such a vertex. By repeating this process until the directed graph becomes empty, we obtain an ordering $v_1,\ldots,v_n$ of the vertices of $\vec{G}$. Because of the construction above, if (v_i, v_j) is an edge of $\vec{G}$, then v_i must be deleted before v_j can be deleted, and thus, $i < j$. Therefore, $v_1,\ldots,v_n$ is a topological ordering. ■

Proposition 14.21's justification suggests an algorithm for computing a topological ordering of a directed graph, which we call ***topological sorting***. We present a Python implementation of the technique in Code Fragment 14.11, and an example execution of the algorithm in Figure 14.13. Our implementation uses a dictionary, named incount, to map each vertex v to a counter that represents the current number of incoming edges to v, excluding those coming from vertices that have previously been added to the topological order. Technically, a Python dictionary provides $O(1)$ expected time access to entries, rather than worst-case time; as was the case with our graph traversals, this could be converted to worst-case time if vertices could be indexed from 0 to $n-1$, or if we store the counter as an element of a vertex.

As a side effect, the topological sorting algorithm of Code Fragment 14.11 also tests whether the given directed graph $\vec{G}$ is acyclic. Indeed, if the algorithm terminates without ordering all the vertices, then the subgraph of the vertices that have not been ordered must contain a directed cycle.