

4.1: Undirected Graphs

Exercise 1. What is the maximum number of edges in a graph with V vertices and no parallel edges? What is the minimum number of edges in a graph with V vertices, none of which are isolated (have degree 0)?

Solution. No parallel edges means that at most one edge connects any two given nodes. For any vertex v_i , there are V possible edge candidates, including v_0 itself (because loops are not disallowed, we can assume they are allowed). Then, for v_1 , there are $V - 1$ edges allowed: one for v_1 , and one for each other vertex, except v_0 . Continuing this way, we find that there is a maximum of $V!$ (V factorial) edges.

If V is even, then the minimum is $V/2$, since we can pair all vertices. If V is odd, it is $\lfloor V/2 \rfloor + 1$.

Exercise 2. Draw, in the style of the figure in the text (page 524), the adjacency lists built by `Graph`'s input stream constructor for the file `tinyGex2.txt` depicted at left (input from `tinyGex2.txt`) (see also Figure 1).

```
12
16
 8 4
 2 3
 1 11
 0 6
 3 6
10 3
 7 11
 7 8
11 8
 2 0
 6 2
 5 2
 5 10
 5 0
 8 1
 4 1
```

Solution. See Figure 2.

Exercise 3. Create a copy constructor for `Graph` that takes as input a graph `G` and creates and initializes a new copy of the graph. Any changes a client makes to `G` should not affect the newly created graph.

Solution. See `com.segarciat.algs4.ch4.sec1.ex03`.

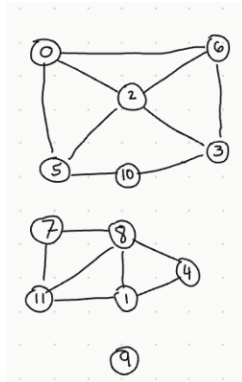


Figure 1: Graph from `tinyGex2.txt`.

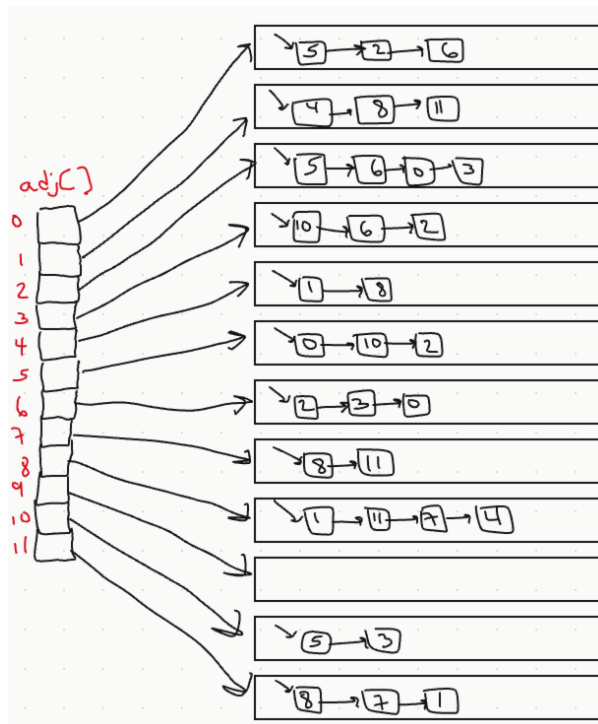


Figure 2: Adjacency list representation for undirected graph from `tinyGex2.txt`

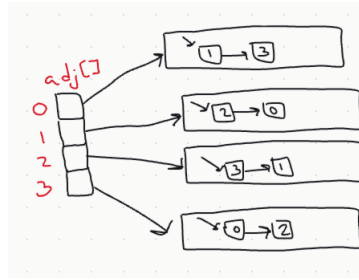


Figure 3: Impossible adjacency-lists for a four-vertex graph with edges 0-1, 1-2, 2-3, and 3-0.

Exercise 4. Add a method `hasEdge()` to `Graph` which takes two `int` arguments `v` and `w` and returns `true` if the graph has an edge `v-w`, `false` otherwise.

Solution. See `com.segarciat.algs4.ch4.sec1.ex04`.

Exercise 5. Modify `Graph` to disallow parallel edges and self-loops.

Solution. See `com.segarciat.algs4.ch4.sec1.ex05`.

Exercise 6. Consider the four-vertex graph with edges 0-1, 1-2, 2-3, and 3-0. Draw an array of adjacency-lists that could *not* have been built calling `addEdge()` for these edges *no matter what order*.

Solution. See Figure 3. The contents suggest that

1. According to 0's adjacency list, 0-3 comes before 0-1.
2. According to 3's adjacency list, 2-3 comes before 0-3.
3. According to 2's adjacency list, 1-2 comes before 2-3.
4. According to 1's adjacency list, 0-1 comes before 1-2.

According to the first three, the implied order is

1-2
2-3
0-3
0-1

but then 1's adjacency list says that 0-1 comes before 1-2, which contradicts that 0-1 comes last in the list above. This can be seen from the adjacency lists because there must be first pair, which means that there is a pair of vertices `v` and `w` that are last in each other's adjacency lists. That would imply that `v-w` (or `w-v`) was the first edge inserted. That doesn't happen in the figure, however.

Exercise 7. Develop a test client for `Graph` that reads a graph from the input stream named as command-line argument and then prints it, relying on `toString()`.

Solution. See `com.segarciat.algs4.ch4.sec1.ex07`.

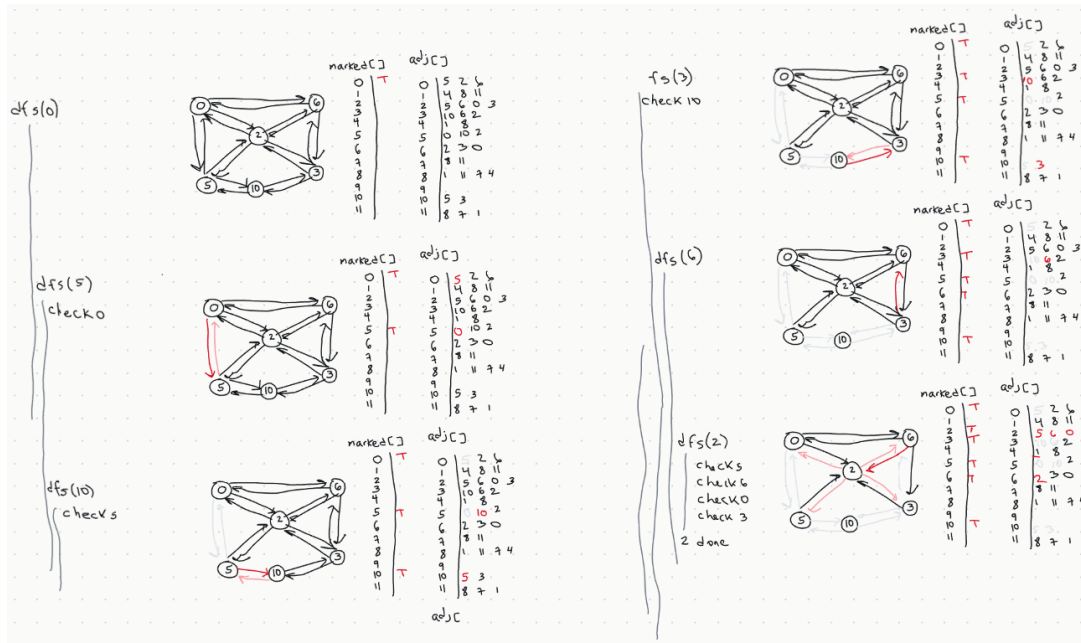


Figure 4: Trace of depth-first search to find all paths from 0 on `tinyGex2.txt`.



Figure 5: Tree built from Figure 4.

Exercise 8. Develop an implementation of the `Search` API on page 528 that uses UF, as described in the text.

Solution. See `com.segarciat.algs4.ch4.sec1.ex08`.

Exercise 9. Show, in the style of page 533, a detailed trace of the call `dfs(0)` for the graph built by `Graph`'s input stream constructor for the file `tinyGex2.txt` (see Exercise 4.1.2 and Figure 1). Also, draw the tree represented by `edgeTo[]`.

Solution. From Figure 1, we see that G is not connected, so we can focus on the connected component of G that contains 0. See Figure 4 for the trace, and Figure 5 for the tree.

Exercise 10. Prove that every connected graph has a vertex whose removal (including all incident edges) will not disconnect the graph, and write a DFS method that finds such a vertex. *Hint:* Consider a vertex whose adjacent vertices are all marked.

Solution.

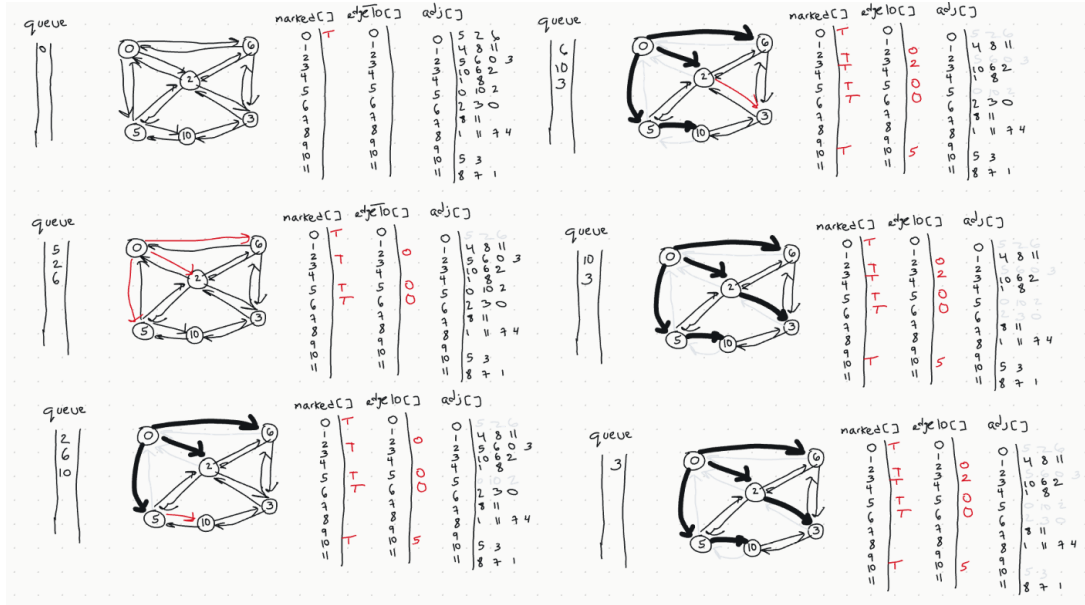


Figure 6: Trace of breadth-first search (BFS) on `tinyGex2.txt`.

Proof. Suppose G is connected. If we choose any vertex s , **Proposition A** in Section 4.1 of [SW11] implies that the depth-first search algorithm marks all vertices in the graph, since G is connected. Eventually, the algorithm encounters a vertex whose adjacent vertices are all marked. Otherwise, the algorithm always finds a new unmarked vertex, but this must stop after at most $|G|$ steps because at that point all vertices in the graph have been marked.

Let u be the vertex whose adjacent vertices have all been marked. If we remove u and all edges connected to it, then the graph remains connected. To see this, suppose v and w are two distinct vertices, neither of which are u . If v or w were adjacent to u , then the fact that its adjacent vertices have been marked means that a path to either v or w from s was already found. Otherwise, if neither v nor w were adjacent to u , then the fact that depth-first search marks all vertices means that there is a path from s to both v and w . Thus, if p_{vs} is a path from v to s , and p_{sw} is a path from s to w , then concatenating p_{vs} and p_{sw} creates a path p_{vw} from v to w . Hence, after removing u and the edges containing u from G , the graph remains connected. \square

Exercise 11. Draw the tree represented by `edgeTo[]` after the call `bfs(G, 0)` in **Algorithm 4.2** for the graph built by `Graph`'s input stream constructor for the file `tinyGex2.txt` (see **Exercise 4.1.2** and Figure 1).

Solution.

Solution.

Exercise 12. What does the BFS tree tell us about the distance from v to w when neither is at the root?

Solution. If the path from the root s to v has length k and the path from s to w has length m , then it tells us that the distance from v to w is bounded by $k + m$, or:

$$\text{dist}(v, w) \leq \text{dist}(v, s) + \text{dist}(s, w)$$

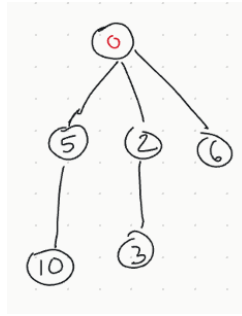


Figure 7: Tree representation of `edgeTo[]` array corresponding to BFS in Figure 6.

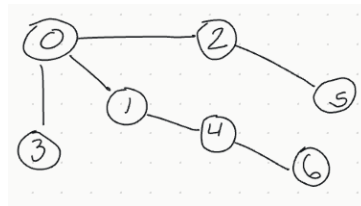


Figure 8: Exercise 14

However, it does not tell us what the distance is, since we only know information about vertices relative to the root `s`.

Exercise 13. Add a `distTo()` method to the `BreadthFirstPaths` API and implementation, which returns the number of edges on the shortest path from the source to a given vertex. A `distTo()` query should run in constant time.

Solution. See `com.segarciat.algs4.ch4.sec1.ex13`.

Exercise 14. Suppose you use a stack instead of a queue when running breadth-first search. Does it still compute shortest paths?

Solution. No. In Figure 8, if 0 were the source vertex, and 2 was the last note in the adjacency list for node 0, then it would be the first removed from the stack. Processing it would lead to a path of length 4, namely 0-2-5-6-4, but the shortest path from 0 to 4 is 0-1-4, of length 2.

Exercise 15. Modify the input stream constructor for `Graph` to also allow adjacency lists from standard input (in a manner similar to `SymbolGraph`), as in the example `tinyGadj.txt` shown at right. After the number of vertices and edges, each line contains a vertex and its list of adjacent vertices.

Solution. See `com.segarciat.algs4.ch4.sec1.ex15`.

Exercise 16. The *eccentricity* of a vertex `v` is the length of the shortest path from that vertex to the furthest vertex from `v`. The *diameter* of a graph is the maximum eccentricity of any vertex. The *radius* of a graph is the smallest eccentricity of any vertex. A *center* is a vertex whose eccentricity is the radius. Implement the following API:

```

public class GraphProperties
    GraphProperties(Graph G) // constructor (exception if G not connected)

```

```
int diameter()           // diameter of G
int radius()             // radius of G
int center()             // a center of G
```

Solution. See `com.segarciat.algs4.ch4.sec1.ex16`.

Exercise 17. The *Wiener index* of a graph is the sum of the lengths of the shortest paths between all pairs of vertices. Mathematical chemists use this quantity to analyze *molecular graphs*, where vertices correspond to atoms and edges correspond to chemical bonds. Add a method `wiener()` to `GraphProperties` that returns the Wiener index of a graph.

Solution. See `com.segarciat.algs4.ch4.sec1.ex17`.

Exercise 18. The *girth* of a graph is the length of its shortest cycle. If a graph is acyclic, then its girth is infinite. Add a method `girth()` to `GraphProperties` that returns the girth of the graph. *Hint:* Run BFS from each vertex. The shortest cycle containing `s` is an edge between `s` and some vertex `v` concatenated with a shortest path between `s` and `v` (that doesn't use edge `s-v`).

Solution. See `com.segarciat.algs4.ch4.sec1.ex18`.

Exercise 19. Show, in the style of the figure on page 545, a detailed trace of `CC` for finding the connected components in the graph built by `Graph`'s input stream constructor for the file `tinyGex2.txt` (see **Exercise 4.1.2** and Figure 1).

Solution. See Figure 9.

Exercise 20. Show, in the style of the figures in this section, a detailed trace of `Cycle` for finding a cycle in the graph built by `Graph`'s input stream constructor for the file `tinyGex2.txt` (see **Exercise 4.1.2** and Figure 1). What is the order of growth of the running time of the `Cycle` constructor, in the worst case?

Solution. See Figure 10. The constructor of `Cycle` uses depth-first search on all vertices, and its implementation is nearly equivalent to `CC`. Similar to **Proposition C** in [SW11], each adjacency-list entry is examined once, and all $2E$ entries are examined. Then there is the cost for initializing the `marked[]` array of size V . Thus the order of growth of the running time of the `Cycle` constructor is $E + V$ in the worst case.

Exercise 21. Show, in the style of the figures in this section, a detailed trace of `TwoColor` for finding a two-coloring of the graph built by `Graph`'s input stream constructor for the file `tinyGex2.txt` (see **Exercise 4.1.2** and Figure 1). What is the order of growth of the running time of the `TwoColor` constructor, in the worst case?

Solution. Once again, the code is similar to `CC`, applying depth-first search on all vertices in the graph. Thus, the order of growth of the running time is about $E + V$ in the worst case, according to **Proposition C** in [SW11]. I am omitting the detailed trace, but upon reaching vertex 2, and examining its adjacency list, the algorithm would encounter vertex 5 with the same color. Thus the graph is not bipartite.

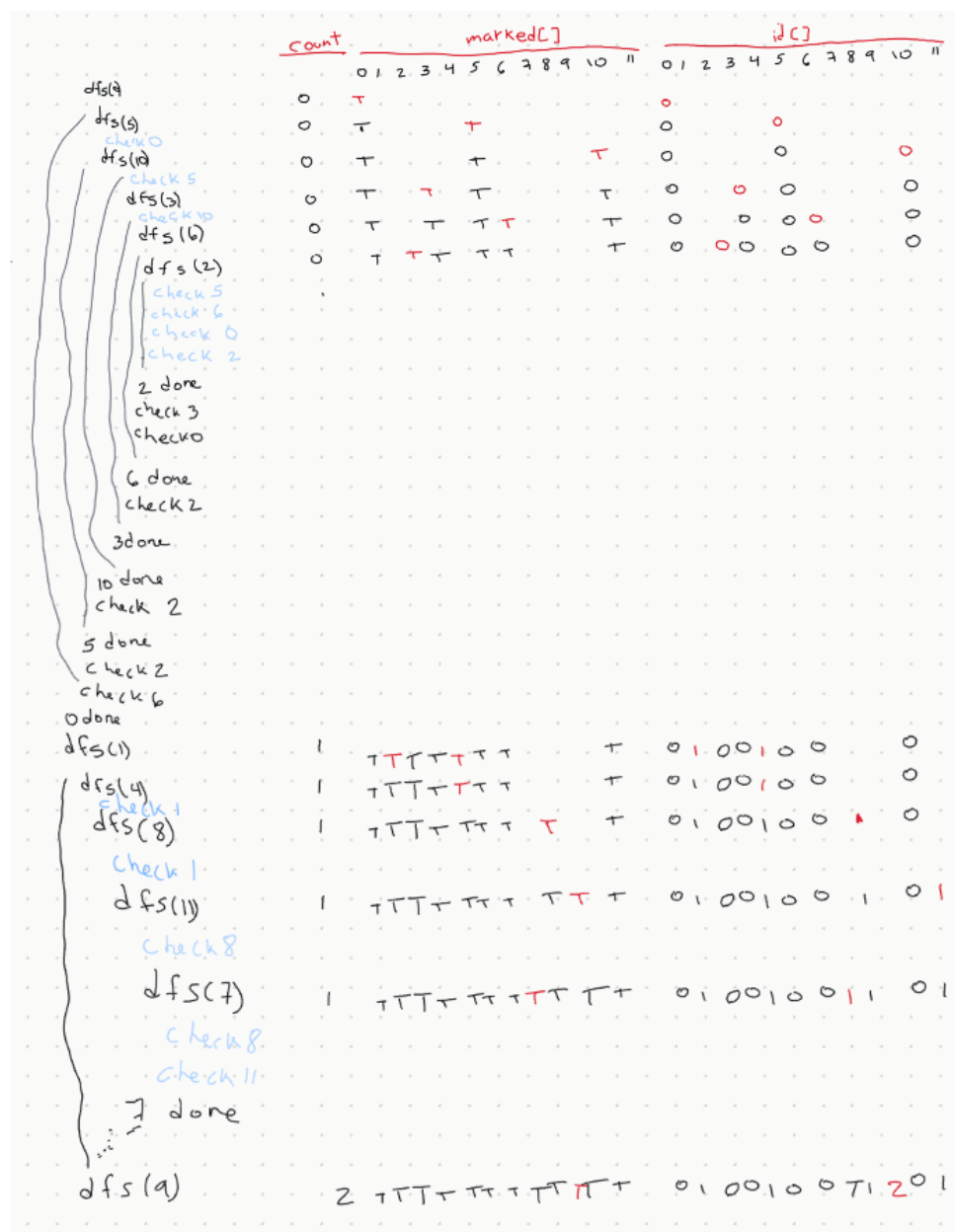


Figure 9: Trace of `CC` to find the connected components corresponding to `tinyGex2.txt` in **Exercise 4.1.2**.

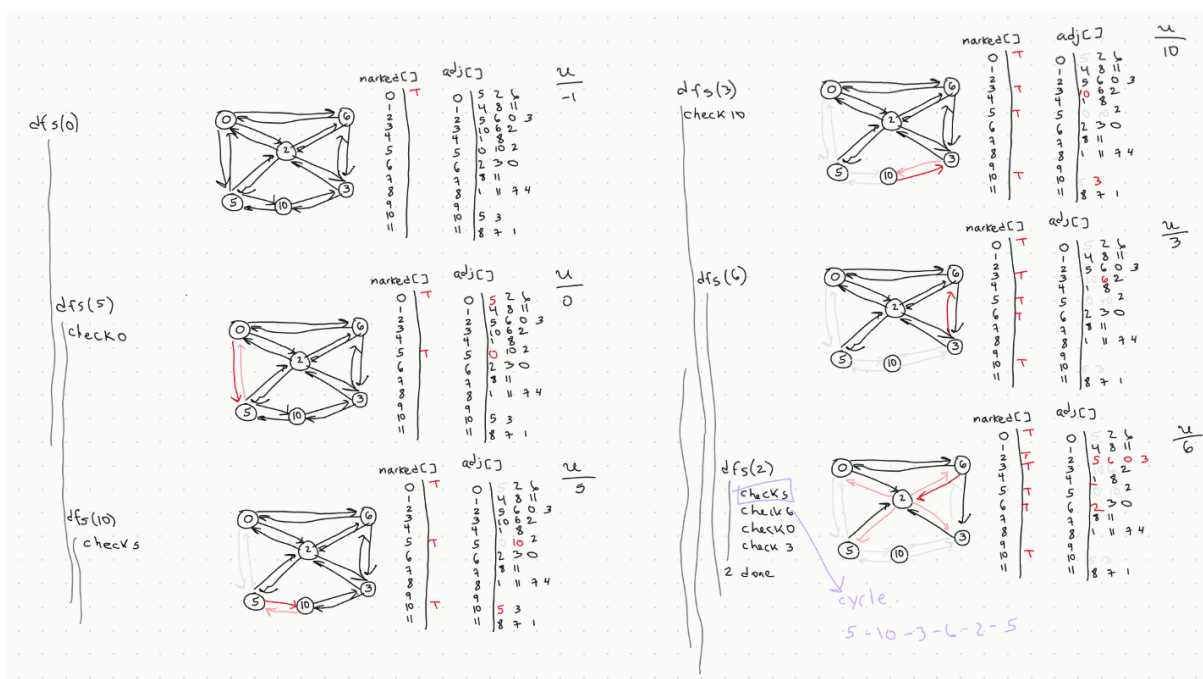


Figure 10: Trace of `Cycle` to detect a cycle in the graph determined by `tinyGex2.txt` in **Exercise 4.1.2**.

Exercise 23. Write a program `BaconHistogram` that prints a histogram of Kevin Bacon numbers, indicating how many performers from `movies.txt` have a Bacon number of 0, 1, 2, 3, ... Include a category for those who have an infinite number (not connected to Kevin Bacon).

Solution. See `com.segarciat.algs4.ch4.sec1.ex23`.

Exercise 24. Compute the number of connected components in `movies.txt`, the size of the largest component, and the number of components of size less than 10. Find the eccentricity, diameter, radius, a center, and the girth of the largest component in the graph. Does it contain Kevin Bacon?

Solution. See `com.segarciat.algs4.ch4.sec1.ex24`.

Exercise 26. Write a `SymbolGraph` client like `DegreesOfSeparation` that uses *depth-first* search instead of breadth-first search to find paths connecting two performers.

Solution. See `com.segarciat.algs4.ch4.sec1.ex26`.

Exercise 27. Determine the amount of memory used by `Graph` to represent a graph with V vertices and E edges, using the memory-cost model of Section 1.4

Solution. The breakdown is: 16 bytes of object overhead, 4 bytes for the `int` instance variable V , 4 bytes for the `int` instance variable E , 8 bytes for the reference to `adj`, 24 bytes for the array `adj`, itself, making up the flat cost part of 56 bytes. Next, there are $8V$ bytes of references to `Bag` objects. For each `Bag` object, there's 16 bytes of overhead, 8 bytes for the reference to the `first` node, 4 bytes for a reference to an `int` for the size,

and 4 bytes of padding. That means a $40V$ cost for the **Bag** objects. Lastly, the cost of each **Node** is about 40 bytes, of which there are $2E$. 16 bytes of object overhead, 8 bytes for a reference to the **item** field, 8 bytes for a reference to the **next** field, 8 bytes for a reference to the enclosing class, which is 40 bytes. The referenced object is an **Integer**, which takes up about 24 bytes. Since there are $2E$ values among all adjacency lists (since edges are double counted), and each takes about 64 bytes, that comes out to $64 \cdot 2E = 128E$ bytes. The total cost is $56 + 40V + 128E$ bytes.

Exercise 29. Modify **Cycle** so that it works even if the graph contains self-loops and parallel edges.

Solution. See `com.segarciat.algs4.ch4.sec1.ex29`.

Exercise 38. *Nonrecursive depth-first search.* Implement depth-first search using an explicit stack instead of recursion. *Warning:* Replacing the queue in **BreadthFirstPaths** with a stack yields some graph searching algorithm but not depth-first search.

Solution. See `com.segarciat.algs4.ch4.sec1.ex38`.

References

- [SW11] Robert Sedgewick and Kevin Wayne. *Algorithms*. 4th ed. Addison-Wesley, 2011.
ISBN: 9780321573513.