Sergio E. Garcia Tapia
Algorithms by Sedgewick and Wayne (4th edition) [SW11]
January 13, 2025

4.2: Directed Graphs

Exercise 1. What is the maximum number of edges in a digraph with V vertices and no parallel edges? What is the minimum number of edges in a digraph with V vertices, none of which are isolated?

Solution. Suppose parallel edges are not allowed, but self-loops are. If there are V vertices, and v_i is a given vertex, then there are $|V|$ possible edges incident from v_i, including the self-loop `v`$_i$`->v`$_i$. If we do this for $i = 1, \ldots, |V|$, then we see that there are $|V|^2$ possible edges.

Now suppose that parallel edges and self-loops are allowed, but we care for the case where no vertices are isolated. This means that the outdegree of the vertex cannot be 0 (what about indegree)? If there are $v_1, \ldots, v_n$ vertices, where $n = |V|$, then we can arrange for exactly one edge to leave each vertex, namely v$_i$-¿v$_{i+1}$, for $1 \leq i < n$. Then, we add one more edge v$_n$-¿v$_i$, for example. Thus we have a minimum of $|V|$ edges.

Exercise 2. Draw, in the style of the figure in the text (page 524), the adjacency lists built by `Digraph`'s input stream constructor for the file `tinyDGex2.txt` (see Figure 1).

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

Solution. See Figure 2.

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

Solution. See `com.segarciat.algs4.ch4.sec2.ex03`.

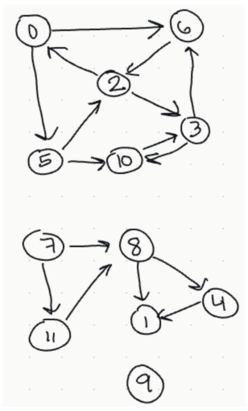

Figure 1: Digraph formed by using the input stream constructor to `Digraph` with `tinyGex2.txt`.

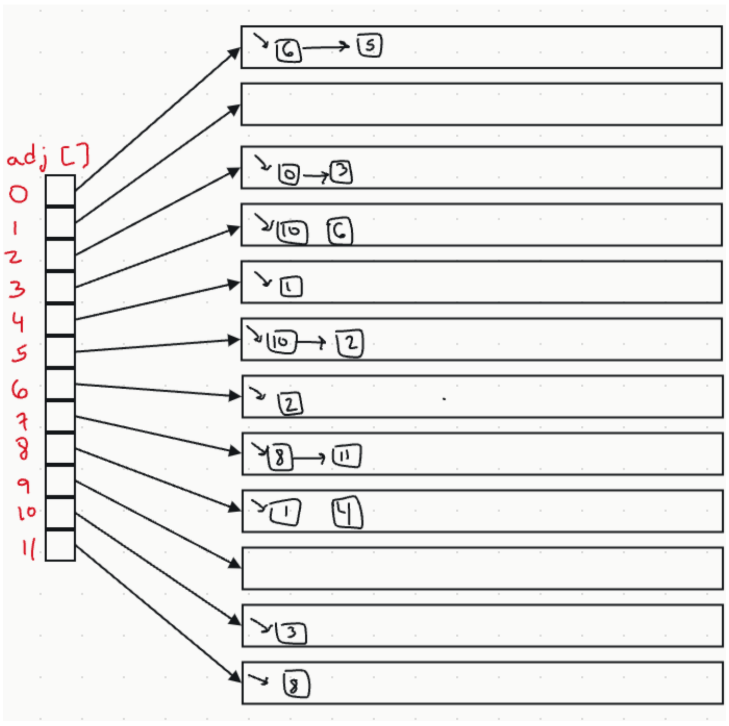

Figure 2: Adjacency list for `Digraph` built from `tinyGex2.txt`.

Exercise 4. Add a method `hasEdge()` to `Digraph` 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.sec2.ex04`.

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

Solution. See `com.segarciat.algs4.ch4.sec2.ex05`.

Exercise 6. Develop a test client for `Digraph`.

Solution. See `com.segarciat.algs4.ch4.sec2.ex06`.

Exercise 7. The *indegree* of a vertex in a digraph is the number of directed edges that point to that vertex. The *outdegree* of vertex in a digraph is the number of directed edges that emanate from that vertex. No vertex is reachable from a vertex of outdegree 0, which is called a *sink*; a vertex of indegree 0, which is called a *source*, is not reachable from any other vertex. A digraph where self-loops are allowed *and* every vertex has outdegree 1 is called a *map* (a function from the set of integers from 0 to $V - 1$ onto itself). Write a program `Degrees.java`. that implements the following API:

```
       public class Degrees
                int Degrees(Digraph G)  // constructor
                int indegree(int v)     // indegree of v
                int outdegree(int v)    // outdegree of v
  Iterable<Integer> sources()           // sources
  Iterable<Integer> sinks()             // sinks
            boolean isMap()             // is G a map?
```

Solution. See `com.segarciat.algs4.ch4.sec2.ex07`.

Exercise 9. Write a method that checks whether a given permutation of a DAG's vertices is a topological order of that DAG.

Solution. See `com.segarciat.algs4.ch4.sec2.ex09`.

Exercise 10. Given a DAG, does there exist a topological order that cannot result from applying a DFS-based algorithm, no matter in what order the vertices adjacent to each vertex are chosen? Prove your answer.

Solution. No, such a topological order does not exist. It is possible to obtain any topological order by using a DFS-based algorithm.

Proof. Let G be a DAG of V vertices, $n = V$, and σ be a topological order on G. If σ_k is the kth vertex in the order, then σ_n must be a sink. Otherwise, a vertex would follow it, and we would not have a topological order. If we apply DFS to σ_n, it would return immediately. When considering σ_i, where $i < n$, all vertices that come after σ_i have been marked. Once again, either σ_i is a sink (and DFS immediately returns), or it points to a vertex that has already been marked. In either case, the result is that the vertex is done being processed. We continue this way until reaching $i = 1$, at which point our DFS-based algorithm ends. Along the way, vertices were done in reverse order of σ, and hence the algorithm computes the topological order to be the reverse order of the reverse order of σ, which of course is σ itself. $\square$

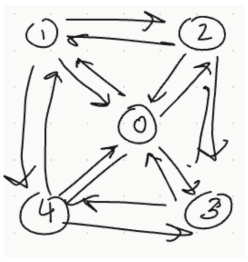

Figure 3: Digraph for Exercise 11.

Exercise 11. Describe a family of sparse digraphs whose number of directed cycles grows exponentially in the number of vertices.

Solution. Digraphs with a center vertex. For example, consider a graph G of $n+1$ vertices, where for each `k` there is an edge `k->0` and an edge `0->k`, where $1 \leq k \leq n$. Also, for k and $k+1$, there is a pair of edges `k->(k+1)` and `(k+1)->k`, for $k > 1$, and for $k = n$, we have `k->1` and `1->k`. Then 0 is the center vertex of such a graph. Such a graph is sparse because there are $6 \cdot n$ edges when there are $n+1$ vertices. See Figure 3.

Such a graph is strongly connected. Given subset of the vertices that contains 0, we can create a directed cycle containing 0. If S_0 is the set of vertices without 0, and $\mathcal{P}(S_0)$ is the power set of S_0, $|\mathcal{P}(S_0)| = 2^n$. By appending 0 to each set, we have at least one unique cycle for each set in $\mathcal{P}(S_0)$, meaning at least 2^n cycles.

Exercise 12. Prove that the strong components in G^R are the same as in G.

Solution.

Proof. Suppose that C is a strong component of G, and let $u, v \in C$. Then there is a path p_{uv} from u to v and there is a path p_{vu} from v to u. In G^R, edges change direction, so the edges in the path p_{uv} and reverse to become path p^R_{uv}, which is now a path from v to u. Similarly, p^R_{vu} is a path from u to v. Hence, u and v belong to the same strong component in G^R. Thus, if C^R is the strong component in G^R that u and v belong to, we see that $C \subseteq C^R$.

Now suppose that $w \notin C$, but w belongs to the same component as u and v in G^R (that is, $u, v, w \in C^R$). Then, without loss of generality, there is a path p^R_{vw} from v to w and a path p^R_{wv} from w to v in G^R. If we reverse the edges of G^R to obtain $(G^R)^R = G$, then the edges in both paths are reversed, and we obtain a path $(p^R_{vw})^R$, from w to v and a path $(p^R_{wv})^R$ from v to w in G. This implies that w and v are in the same strong component, which contradicts the definition of w. Hence, $w \notin C^R$.

We've just argued that if $w \notin C$, then $w \notin C^R$, which implies that $C^R \subseteq C$. We conclude $C = C^R$. $\square$

Exercise 13. Prove that two vertices in a digraph G are in the same strong component if and only if there is a directed cycle (not necessarily simple) containing them both.

Solution.

Proof. Let $v, w \in G$.

Suppose that v and w belong to the same strong component. Then w is reachable from v through a directed path p_{vw} and v is reachable from w through a directed path p_{wv}. By concatenating the paths, we create a cycle that contains both v and w.

Now suppose that there is a cycle $u_1, u_2, \ldots, u_n, u_1$ containing v an w. Suppose $v = u_i$ and $w = u_j$, where $j > i$. Then there is a path $v_i v_{i+1} \cdots v_{j-1} v_j$ from v_i to v_j and a path $v_j v_{j+1} \cdots v_n v_1 \cdot v_i$ from v_j to v_i. Hence, v and w are strongly connected. □

Exercise 14. Let C be a strong component in a digraph G and let v be any vertex not in C. Prove that if there is an edge e leaving v to any vertex in C, then vertex v appears before *every* vertex in C in the reverse post order of G.

Solution.

Proof. Suppose that u is any vertex in C. If `dfs(u)` is called *before* `dfs(v)`, then every vertex in C will be marked and added to the reverse post order because C is a strong component. However, v will not be added, because if it did, that would imply a path from u to v. Together with each e from v to a component in C, this would imply that u and v are strongly connected, which would contradict the fact that $v \notin C$. The conclusion is that all vertices in C are pushed onto the stack prior to v.

On the other hand, if `dfs(v)` is called *before* `dfs(u)`, then since there is an edge e from v to a component in C, this will result in all components in C being added to the stack before v. Once again, v is added to the stack after all vertices in C, and hence appears *before* every vertex in C in the reverse postorder of G. □

Exercise 15. Let C be a strong component in a digraph G and let `v` be any vertex not in C. Prove that if there is an edge e leaving any vertex in C to v, then vertex `v` appears before *every* vertex in C in the reverse postorder of G^R.

Solution.

Proof. Consider the reverse graph G^R of G. By **Exercise 4.2.12**, C is also a strong component of G^R. Since e is an edge in G leaving C and incident to `v`, its reverse e^R is an edge of G^R leaving `v` incident to a vertex in C. By **Exercise 4.2.14** applied to G^R, we conclude that `v` appears before every vertex in C in the reverse post order of G^R. □

Exercise 16. Given a digraph G, prove that the first vertex in the reverse postorder of G is in a strong component that is a *source* of G's kernel DAG. Then, prove that the first vertex in the reverse postorder of G^R is in a strong component that is a *sink* of G's kernel DAG. *Hint*: Apply **Exercises 4.2.14** and **4.2.15**.

Solution.

Proof. Let v be the first vertex in the reverse postorder of G, and let C be the strong component of G that v belongs to. Suppose that C is not a source of the kernel DAG of G. Then there is a component C' distinct from C and at least one edge from that component to C. In particular, there is a vertex $w \in C'$ that points to a vertex in C. By **Exercise 4.2.14**, w must appears before every vertex in C in the reverse post order of G, including before v, contradicting the definition of v. This implies that C' does not exist, and hence, that C is indeed a source of the kernel DAG of G.

Now let u be the first vertex in the reverse post order of G^R, and let D be the strong component that it belongs to. If D is not a sink of the kernel DAG of G, then there is a strong component D' of G (and G^R) and edge f from D to D'. If z is the tail of f, the $z \in D'$. By **Exercise 4.2.15**, z must appear before every vertex in D in the reverse post order of G^R, contradicting the definition of u. This implies that D' does not exist, and hence, that D is indeed a sink of the kernel DAG of G. $\square$

Exercise 17. How many strong components are there in the digraph on page 591?

Solution. See `com.segarciat.algs4.ch4.sec2.ex17`.

Exercise 18. What are the strong components of a DAG?

Solution. The strong components of a DAG are the singleton vertices of the DAG. Indeed, by definition, a vertex is reachable from itself, so each strong component must have at least 1 vertex. On the other hand, if any component had 2 or more vertices, call them v and w, then there would be a cycle involving v and w. This cannot happen in a DAG, so no such component can exist.

Exercise 19. What happens if you run the Kosaraju-Sharir algorithm on a DAG?

Solution. Assuming a total of V vertices, it will identify a total of V singleton strong components.

Exercise 20. True or false: The reverse postorder of a digraph's reverse is the same as the postorder of the digraph.

Solution. False. Consider digraph G with vertices `1`, `2`, and `3`, and the single edge `1->2`. Its reverse G^R has the single edge `2->1`.

The algorithm for the reverse postorder (applied to G^R) starts with `dfs(1)`, which is immediately added to the stack, followed by `2`, and followed by `3`. Hence, the algorithm computes the reverse postorder `3`, `2`, `1` for G.

Applying the algorithm to G, we starts with `dfs(1)`, which causes a call to `dfs(2)`, which returns and is added to the queue, followed by `1`. Since `2` is marked, it is skipped, and then `dfs(3)` is called, causing `3` to be added to the queue. Now the queue contains the postorder `2`, `1`, `3` of G, which is distinct from the reverse postorder of G^R.

Exercise 21. True or false: If we consider the vertices of a digraph G (or its reverse) in postorder, then vertices in the same strong component will be consecutive in that order.

Solution. False. See Figure 4. The call `dfs(0)` will result in the postorder `1`, `3`, `2`, and `0` of G. However, `1` and `0` are in the same component, and they do not appear consecutively.

Exercise 22. True or false: If we modify the Kosaraju-Sharir algorithm to run the depth-first search in the digraph G (instead of the reverse digraph G^R) and the second depth-first search in G^R (instead of G), then it will still find the strong components.

Solution. True.

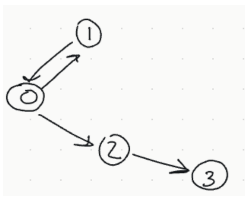

Figure 4: Digraph for Exercise 21. The postorder of G is not the same as the reverse postorder of G^R.

Proof. Recall that the first vertex in the reverse postorder of G is in a source of the kernel DAG of G. However, it is in a sink component of the kernel DAG of G^R. If we let $H = G^R$, and $H^R = G$, then the algorithm computes the reverse postorder of H^R, and then uses DFS to visit the vertices of H in that order, which is precisely applying the Kosaraju-Sharir algorithm to H. By **Proposition H**, the algorithm identifies the strong components of $H = G^R$. By **Exercise 4.2.12**, the strong components of G^R are precisely the same as the strong components of G. $\square$

Exercise 23. True or false: If we modify the Kosaraju-Sharir algorithm to replace the second depth-first search with breadth-first search, then it will still the strong components.

Solution. True. Recall that the first vertex in the reverse postorder of G^R appears in a sink of the kernel DAG of G (see **Exercise 4.2.16**). When BFS is used with that first vertex as the source, all vertices in that component are visited (by definition of strong connectivity), and no other vertices. Suppose that a vertex v that did not belong to the sink is visited. This would imply that an edge from a sink of the kernel DAG to the component containing v, contradicting the definition of a sink. Now that all vertices in that sink component are visited, they are effectively removed, and the algorithm continues in the same way in the remainder of the digraph.

The only thing that changes is the order in which the vertices on a given component are visited.

Exercise 24. Compute the memory usage of a `Digraph` with V vertices and E edges, under the memory cost model of Section 1.4.

Solution. First, there are 16 bytes of object overhead for `Digraph`. The class needs 4 bytes for its `int E` instance variable, and 4 bytes for its `int V` instance variable. There is also 8 bytes for its reference to the `adj` array that contains the adjacency lists of the vertices of the digraph. The array itself takes up 24 bytes. So far, that's a flat cost of 56 bytes.

The `adj` array it has V references to `Bag` objects, and each reference is 8 bytes. Each `Bag` contains the adjacency list of a given vertex. A `Bag` object uses 16 bytes of object overhead, `8` bytes for its reference to the `first` node, `4` bytes for its to the `int size` instance variable, and 4 bytes of padding for a total of 32 bytes. Thus, that's $40V$ bytes for the `Bag` objects themselves.

Now, there are E edges, and each is represented by a single `Node` object. A `Node` uses 40 bytes, and since it points to `Integer` objects, which take up 24 bytes, the overall cost is `64` bytes for an edge. Hence, we require $64E$ bytes for all edges.

The overall cost of `Digraph` is $56 + 40V + 64E$ bytes.

Exercise 25. How many edges are there in the transitive closure of a digraph that is a simple directed path with V vertices and $V - 1$ edges?

Solution. A simple directed path has no repeated edges and no repeated vertices. Let $n = V$, and suppose the simple path is $\mathtt{v}_1\mathtt{v}_2 \cdots \mathtt{v}_n$. Note that $\mathtt{v}_n$ is reachable from all previous vertices, so the transitive closure contains an edge for each of them, a total of $n - 1$ edges. Similarly, the transitive closure has $n - 2$ edges incident to $\mathtt{v}_{n-1}$, and more generally, the transitive closure has $i - 1$ edges incident to v_i, where $1 \leq i \leq n$. Thus, we sum:

$$\sum_{i=1}^{n}(i-1) = \sum_{1\leq i\leq n}(i-1) = \sum_{0\leq i-1\leq n}(i-1) = \frac{(n-1)n}{2}$$

Hence, the transitive closure has $(V - 1)V/2$ edges

References

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