algorithms

MDPI

Article

Adjacency Maps and Efficient Graph Algorithms

Gabriel Valiente

Department of Computer Science, Technical University of Catalonia, E-08034 Barcelona, Spain; gabriel.valiente@upc.edu

Abstract: Graph algorithms that test adjacencies are usually implemented with an adjacency-matrix representation because the adjacency test takes constant time with adjacency matrices, but it takes linear time in the degree of the vertices with adjacency lists. In this article, we review the adjacency-map representation, which supports adjacency tests in constant expected time, and we show that graph algorithms run faster with adjacency maps than with adjacency lists by a small constant factor if they do not test adjacencies and by one or two orders of magnitude if they perform adjacency tests.

Keywords: discrete mathematics; graph theory; performance and testing of algorithms; adjacency matrices; adjacency lists; adjacency maps

1. Introduction

Adjacency lists have been the preferred graph representation for over five decades now because a large number of graph algorithms can be implemented to run in linear time in the number of vertices and edges in the graph using an adjacency-list representation, while no graph algorithm can be implemented to run in linear time using an adjacency-matrix representation. The only exception to the latter is the sparse representation of static directed graphs of [1], which uses (allocated, but uninitialized) quadratic space in the number of vertices in the graph and allows for implementing graph algorithms that test edge existence, such as finding a universal sink (a vertex of in-degree equal to the number of vertices minus one and out-degree zero) in a directed graph ([2] [Ex. 22.1-6]), to run in linear time in the number of vertices and edges in the graph.

Graph algorithms can be described using a small collection of abstract operations on graphs, which can be implemented using appropriate data structures such as adjacency matrices, adjacency lists, and adjacency maps. For example, the representation of graphs in the LEDA library of efficient data structures and algorithms [3] supports about 120 abstract operations, and the representation of graphs in the BGL library of graph algorithms [4] supports about 50 abstract operations.

A smaller collection of 32 abstract operations is described in [5], which allows for describing most graph algorithms. Actually, the following collection of only 11 abstract operations suffices for describing most of the fundamental graph algorithms, where lists of vertices and edges are arranged in the order fixed by the representation of the graph. Much of the following is adapted from ([5] [Section 1.3]).

- $G.vertices()$ gives a list of the vertices of graph G.
- $G.edges()$ gives a list of the edges of graph G.
- $G.incoming(v)$ gives a list of the edges of graph G coming into vertex v.
- $G.outgoing(v)$ gives a list of the edges of graph G going out of vertex v.
- $G.adjacent(v, w)$ is true if there is an edge in graph G going out of vertex v and coming into vertex w, and false otherwise.
- $G.source(e)$ gives the source vertex of edge e in graph G.
- $G.target(e)$ gives the target vertex of edge e in graph G.
- $G.new\_vertex()$ inserts a new vertex in graph G.

check for updates

Citation: Valiente, G. Adjacency Maps and Efficient Graph Algorithms. *Algorithms* **2022**, *15*, 67. https://doi.org/10.3390/a15020067

Academic Editors: Frank Werner and Francesc Pozo

Received: 16 January 2022
Accepted: 18 February 2022
Published: 20 February 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

- $G.new\_edge(v, w)$ inserts a new edge in graph G going out of vertex v and coming into vertex w.
- $G.del\_vertex(v)$ deletes vertex v from graph G, together with all those edges going out of or coming into vertex v.
- $G.del\_edge(e)$ deletes edge e from graph G.

These abstract operations apply to both undirected and directed graphs. An undirected graph is the particular case of a directed graph in which for every edge (v, w) of the graph, the reversed edge (w, v) also belongs to the graph. For example, a simple traversal of an undirected graph G, in which vertices and edges are visited in the order fixed by the representation of the graph, can be described using these abstract operations as shown in Algorithm 1.

Algorithm 1 Simple traversal of an undirected graph G.

```
for all v ∈ G.vertices() do
    for all e ∈ G.outgoing(v) do
        w = G.target(e)
        ...
```

Essentially, the adjacency list representation of a graph is an array of lists, one for each vertex in the graph, where the list corresponding to a given vertex contains the target vertices of the edges coming out of the given vertex. However, this is often extended by making edges explicit, as follows:

Definition 1. *Let $G = (V, E)$ be a graph with n vertices and m edges. The adjacency list representation of G consists of a list of n elements (the vertices of the graph), a list of m elements (the edges of the graph), and two lists of n lists of a total of m elements (the edges of the graph). The incoming list corresponding to vertex v contains all edges $(u, v) \in E$ coming into vertex v, for all vertices $v \in V$. The outgoing list corresponding to vertex v contains all edges $(v, w) \in E$ going out of vertex v, for all vertices $v \in V$. The source vertex v and the target vertex w are associated with each edge $(v, w) \in E$.*

The adjacency list representation of a directed graph is illustrated in Figure 1. The small collection of 11 abstract operations can be implemented using the adjacency list representation to take $O(1)$ time, with the exception of $G.adjacent(v, w)$, which takes $O(\min(\text{outdeg}(v), \text{indeg}(w)))$ time, and $G.del\_node(v)$, which takes $O(\deg(v))$ time, as follows:

- $G.vertices()$ and $G.edges()$ are respectively the list of vertices and the list of edges of graph G.
- $G.incoming(v)$ and $G.outgoing(v)$ are respectively the list of edges coming into vertex v and the list of edges going out of vertex v.
- $G.adjacent(v, w)$ is implemented by scanning the list of edges going out of vertex v, or the list of edges coming into vertex w.
- $G.source(e)$ and $G.target(e)$ are respectively the source and the target vertex associated with edge e.
- $G.new\_vertex()$ is implemented by appending a new vertex v to the list of vertices of graph G, and returning vertex v.
- $G.new\_edge(v, w)$ is implemented by appending a new edge e to the list of edges of graph G, setting to v the source vertex associated with edge e, setting to w the target vertex associated with edge e, appending e to the list of edges going out of vertex v and to the list of edges coming into vertex w, and returning edge e.
- $G.del\_vertex(v)$ is implemented by performing $G.del\_edge(e)$ for each edge e in the list of edges coming into vertex v and for each edge e in the list of edges going out of vertex v, and then deleting vertex v from the list of vertices of graph G.

- $G.del\_edge(e)$ is implemented by deleting edge e from the list of edges of graph G, from the list of edges coming into vertex $G.target(e)$, and from the list of edges going out of vertex $G.source(e)$.

vertex	incoming edges	outgoing edges
v_1	$[e_4]$	$[e_1, e_2]$
v_2	$[e_1, e_5]$	$[e_3]$
v_3	$[e_6, e_{11}]$	$[e_4]$
v_4	$[e_2, e_9]$	$[e_5, e_6, e_7, e_8]$
v_5	$[e_3]$	$[e_9, e_{10}]$
v_6	$[e_7, e_{12}]$	$[e_{11}]$
v_7	$[e_8, e_{10}]$	$[e_{12}]$

vertex	incoming edges	outgoing edges
v_1	$[v_3 \to e_4]$	$[v_2 \to e_1, v_4 \to e_2]$
v_2	$[v_1 \to e_1, v_4 \to e_5]$	$[v_5 \to e_3]$
v_3	$[v_4 \to e_6, v_6 \to e_{11}]$	$[v_1 \to e_4]$
v_4	$[v_1 \to e_2, v_5 \to e_9]$	$[v_2 \to e_5, v_3 \to e_6, v_6 \to e_7, v_7 \to e_8]$
v_5	$[v_2 \to e_3]$	$[v_4 \to e_9, v_7 \to e_{10}]$
v_6	$[v_4 \to e_7, v_7 \to e_{12}]$	$[v_3 \to e_{11}]$
v_7	$[v_4 \to e_8, v_5 \to e_{10}]$	$[v_6 \to e_{12}]$

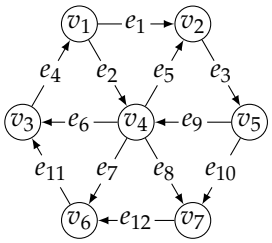

Figure 1. Adjacency list (**top left**) and adjacency map (**bottom**) representation of a directed graph (**top right**). Source and target vertices of each edge not shown.

The adjacency list representation of a graph $G = (V, E)$ with n vertices and m edges takes $O(n + m)$ space, and it allows for implementing graph algorithms such as depth-first search, biconnectivity, acyclicity, planarity testing, topological sorting, and many others to take $O(n + m)$ time [6,7].

In the adjacency list representation of a graph, edges can also be made explicit by replacing the lists of incoming and outgoing edges with dictionaries of source vertices to incoming edges and target vertices to outgoing edges. This allows for a more efficient adjacency test, although adding a logarithmic factor to the cost to all of the operations (when dictionaries are implemented using balanced trees) or turning the worst-case cost for all of the operations into expected cost (when dictionaries are implemented using hashing).

Such a representation was advocated in [5,8], and adopted as the default graph representation in the NetworkX package for network analysis in Python [9]. Essentially, the adjacency map representation of a graph consists of a dictionary D of vertices to a pair of dictionaries of vertices to edges: a first dictionary I of source vertices to incoming edges, and a second dictionary O of target vertices to outgoing edges.

Definition 2. *Let $G = (V, E)$ be a graph with n vertices and m edges. The adjacency map representation of G consists of a dictionary of n elements (the vertices of the graph) to a pair of dictionaries of m elements (the source and target vertices for the edges of the graph, respectively). The incoming dictionary corresponding to vertex v contains the mappings $(u, (u, v))$ for all edges $(u, v) \in E$ coming into vertex v, for all vertices $v \in V$. The outgoing dictionary corresponding to vertex v contains the mappings $(v, (v, w))$ for all edges $(v, w) \in E$ going out of vertex v, for all vertices $v \in V$.*

The adjacency map representation of a directed graph is also illustrated in Figure 1. The small collection of 11 abstract operations can also be implemented using the adjacency map representation to take $O(1)$ expected time, with the exception of $G.del\_vertex(v)$, which takes $O(\deg(v))$ expected time, as follows:

- $G.vertices()$ are the keys in dictionary D.
- $G.edges()$ are the values $D[v].O[w]$ for all keys v in dictionary D and for all keys w in dictionary $D[v].O$.

- $G.incoming(v)$ are the values $D[v].I[u]$ for all keys u in dictionary $D[v].I$.
- $G.outgoing(v)$ are the values $D[v].O[w]$ for all keys w in dictionary $D[v].O$.
- $G.adjacent(v, w)$ is true if $(w, e) \in D[v].O$, where $e = D[v].O[w]$, and false otherwise.
- $G.source(e)$ is the source vertex associated with edge e.
- $G.target(e)$ is the target vertex associated with edge e.
- $G.new\_vertex()$ is implemented by inserting an entry in dictionary D, with the key as a new vertex v and value a pair of empty dictionaries $D[v].I$ and $D[v].O$, and returning vertex v.
- $G.new\_edge(v, w)$ is implemented by setting to v the source vertex associated with a new edge e, setting to w the target vertex associated with edge e, inserting an entry in dictionary $D[v].O$ with key w and value e, inserting an entry in dictionary $D[w].I$ with key v and value e, and returning edge e.
- $G.del\_vertex(v)$ is implemented by performing $G.del\_edge(e)$ for each entry with key u and value e in dictionary $D[v].I$ and for each entry with key w and value e in dictionary $D[v].O$, and then deleting the entry with key v from dictionary D.
- $G.del\_edge(e)$ is implemented by deleting the entry with key w from dictionary $D[v].O$ and deleting the entry with key v from dictionary $D[w].I$, where $v = G.source(e)$ and $w = G.target(e)$.

Similar to the adjacency list representation, the adjacency map representation of a graph $G = (V, E)$ with n vertices and m edges also takes $O(n + m)$ space. In addition to the low space requirement, the main advantage of the adjacency map representation is the support of the adjacency test in $O(1)$ expected time, when dictionaries are implemented using hashing.

In this article, we compare the performance of three graph algorithms on a large benchmark dataset of random directed graphs, when implemented with an adjacency-list and an adjacency-map representation, and we show that they run faster on the average with adjacency maps than with adjacency lists.

2. Materials and Methods

We have implemented 9 of the 11 abstract operations on graphs in Python, namely

- $G.vertices()$
- $G.edges()$
- $G.incoming(v)$
- $G.outgoing(v)$
- $G.adjacent(v, w)$
- $G.source(e)$
- $G.target(e)$
- $G.new\_vertex()$
- $G.new\_edge(v, w)$

for both the adjacency list and the adjacency map representation, and for labeled vertices and edges. Figure 2 shows the corresponding classes in detail.

For the benchmark dataset, we have used random directed graphs with $n = 8, 16, 32, 64, 128, 256$ vertices and $m = 1, \ldots, n(n-1)$ directed edges. These 86,856 directed graphs were generated using the Erdős–Rényi model, by which all (directed) graphs with n vertices and m (directed) edges have the same probability [10,11], as implemented in the NetworkX package for network analysis in Python [9].

```
class Graph:
class Vertex:
def __init__(self,x=None):
self._lbl = x
self._I = list()
self._O = list()
class Edge:
def __init__(self,v,w,x=None):
self._src = v
self._tgt = w
self._lbl = x



def __init__(self):
self._V = list()
self._E = list()
def vertices(self):
return self._V
def edges(self):
return self._E


def incoming(self,v):
return iter(v._I)
def outgoing(self,v):
return iter(v._O)
def adjacent(self,v,w):
if len(v._O) < len(w._I):
return w in
[e._tgt for e in v._O]
else:
return v in
[e._src for e in w._I]
def source(self,e):
return e._src
def target(self,e):
return e._tgt
def new_vertex(self,x=None):
v = self.Vertex(x)
self._V.append(v)
return v


def new_edge(self,v,w,x=None):
e = self.Edge(v, w, x)
self._E.append(e)
v._O.append(e)
w._I.append(e)
return e
```

```
class Graph:
class Vertex:
def __init__(self,x=None):
self._lbl = x
def __hash__(self):
return hash(id(self))
class Edge:
def __init__(self,v,w,x=None):
self._src = v
self._tgt = w
self._lbl = x
def __hash__(self):
return hash((id(self._src),
id(self._tgt)))
def __init__(self):
self._D = dict()

def vertices(self):
return iter(self._D.keys())
def edges(self):
return iter([self._D[v]._O[w]
for v in self._D
for w in self._D[v]._O])
def incoming(self,v):
return iter(self._D[v]._I.values())
def outgoing(self,v):
return iter(self._D[v]._O.values())
def adjacent(self,v,w):
if len(self._D[v]._O)
< len(self._D[w]._I):
return w in self._D[v]._O
else:
return v in self._D[w]._I

def source(self,e):
return e._src
def target(self,e):
return e._tgt
def new_vertex(self,x=None):
v = self.Vertex(x)
self._D[v] = type('', (), {})
self._D[v]._I = dict()
self._D[v]._O = dict()
return v
def new_edge(self,v,w,x=None):
e = self.Edge(v, w, x)
self._D[v]._O[w] = e
self._D[w]._I[v] = e
return e
```

Figure 2. Python implementation of the adjacency list (**left**) and adjacency map (**right**) representation of a directed graph.

In order to compare the performance of the adjacency-list and the adjacency-map representation, we have chosen three graph algorithms:

- A simple algorithm for constructing a graph from a list of edges, thereby testing the performance of the abstract operations for adding new vertices and edges to a graph.
- An iterative algorithm for the breadth-first traversal of a graph ([5] [Section 5.2]), thereby testing the performance of the abstract operations for iterating over the vertices and edges of the graph.
- An algorithm for finding a universal sink of a directed graph ([2] [Ex. 22.1-6]), thereby testing the performance of the adjacency-test abstract operation.

The universal sink algorithm, adapted from [1], is shown in Algorithm 2. The first loop, which breaks at the first iteration, is used to set an initial candidate for the universal sink to the first vertex in the order fixed by the representation of the graph (actually, any vertex of the graph would suffice). The second loop is used to discard all but one of the vertices in the graph as candidates for universal sink. The third loop is used to check if the remaining candidate vertex indeed has in-degree equal to the number of vertices of the graph minus one and out-degree zero.

Algorithm 2 Finding a universal sink in a directed graph $G = (V, E)$ with $|V| \geqslant 2$.

```
function universal_sink(G)
    for all v ∈ vertices(G) do
        break
    for all w ∈ vertices(G) do
        if v ≠ w and G.adjacent(v, w) then
            v = w
    for all w ∈ vertices(G) do
        if G.adjacent(v, w) or (v ≠ w and not G.adjacent(w, v)) then
            return false
    return true
```

Assuming the adjacency test takes $O(1)$ time, the graph construction algorithm, the breadth-first graph traversal algorithm, and the universal sink algorithm all take $O(n + m)$ time, on a graph with n vertices and m directed edges.

3. Results

We have implemented the simple graph construction algorithm and the universal sink algorithm in Python, taken the Python implementation of the breadth-first graph traversal algorithm from ([5] [Appendix A]), and run the algorithms for graph construction, breadth-first traversal, and universal sink on the 86,856 random directed graphs in the benchmark dataset. Table 1 shows the average running time of each of the three algorithms with the adjacency-list and the adjacency-map representation, over $n(n-1)$ = 56, 240, 992, 4032, 16,256, 65,280 random directed graphs with $n = 8, 16, 32, 64, 128, 256$ vertices, respectively, on a computer with a 12-core Intel Xeon processor and 64 GB of memory.

Table 1. Number of vertices (n) for the benchmark graphs, and average running times for graph construction (Construct), breadth-first traversal (Traversal), and universal sink (Sink) for both the adjacency-list (List) and the adjacency-map (Map) representation. All running times are in seconds.

	Construct		Traversal		Sink	
n	List	Map	List	Map	List	Map
8	0.000098	0.000411	0.000049	0.000038	0.000014	0.000027
16	0.000282	0.001071	0.000114	0.000086	0.000020	0.000031
32	0.000822	0.002202	0.000321	0.000239	0.000052	0.000059
64	0.003640	0.008158	0.001079	0.000788	0.000161	0.000110
128	0.015521	0.031588	0.004194	0.003021	0.000659	0.000220
256	0.085243	0.144632	0.016632	0.012799	0.004290	0.000450

The ratio of these average running times (adjacency maps over adjacency lists) for the three graph algorithms are plotted in Figure 3. Graph construction is about 4 times slower with adjacency maps for graphs with at most 16 vertices, but only about 2 times slower for graphs with at least 128 vertices. On the other hand, breadth-first graph traversal and the universal sink algorithm run faster with adjacency maps for all graph sizes in the case of graph traversal and for graphs with at least 64 vertices in the case of universal sink on the average.

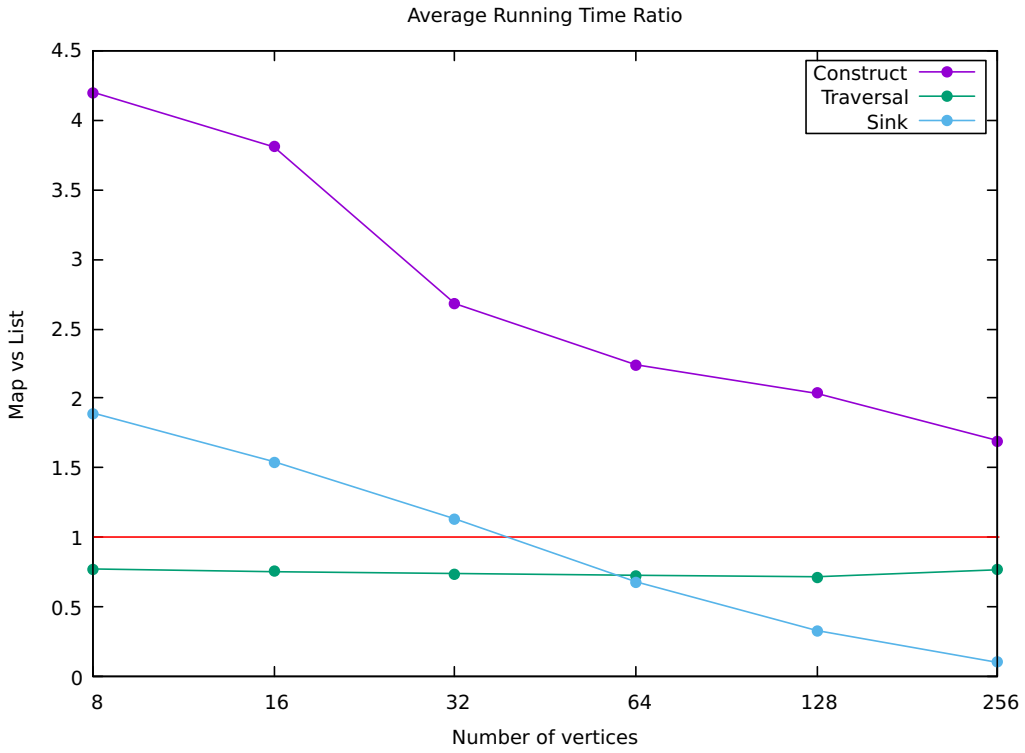

Figure 3. Ratio of the average running time with the adjacency-map over the adjacency-list representation, for the graph construction (violet), breadth-first traversal (green), and universal sink (cyan) algorithms, on random directed graphs with $n = 8, 16, 32, 64, 128, 256$ vertices and $1, \ldots, n(n-1)$ directed edges.

These running times are shown in more detail in Figure 4, where instead of average running times, individual running times are plotted for each of the random directed graphs in the benchmark dataset. Graph construction is almost always slower with adjacency maps for graphs with 8, 16, or 32 vertices, but it is faster with adjacency maps for 170 of the 4032 graphs with 64 vertices, 2870 of the 16,256 graphs with 128 vertices, and 3599 of the 65,280 graphs with 256 vertices in the benchmark dataset.

Breadth-first graph traversal is almost always faster with adjacency maps: for all the 56 graphs with 8 vertices, 239 of the 240 graphs with 16 vertices, 989 of the 992 graphs with 32 vertices, 4022 of the 4032 graphs with 64 vertices, 16,247 of the 16,256 graphs with 128 vertices, and 65,266 of the 65,280 graphs with 256 vertices in the benchmark dataset. The universal sink algorithm is always slower with adjacency maps for graphs with 8 or 16 vertices, but it is faster with adjacency maps for 394 of the 992 graphs with 32 vertices, 2828 of the 4032 graphs with 64 vertices, 13,839 of the 16,256 graphs with 128 vertices, and 60,582 of the 65,280 of the graphs with 256 vertices in the benchmark dataset.

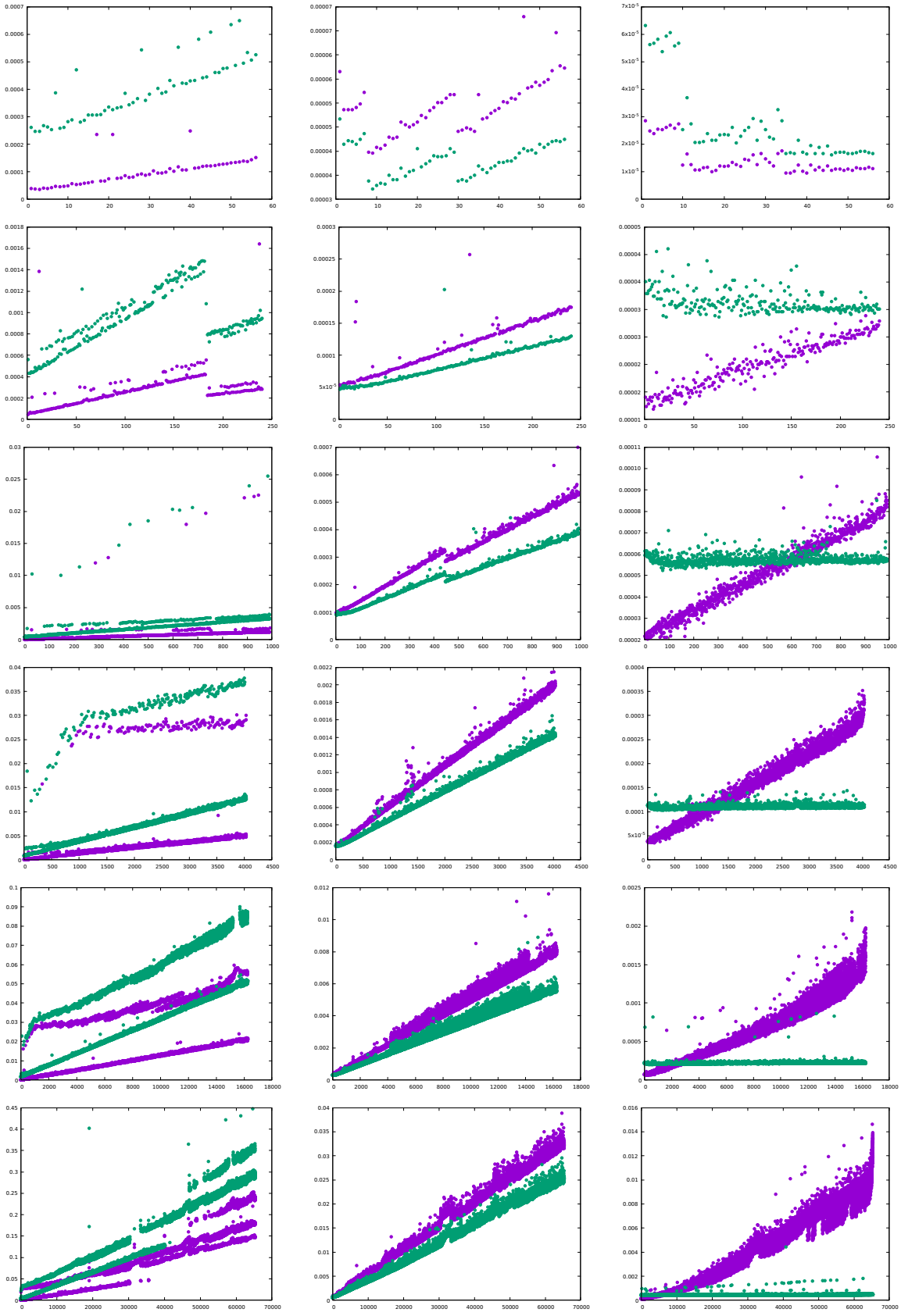

Figure 4. Running time (seconds) for the graph construction (**left**), breadth-first traversal (**center**), and universal sink (**right**) algorithms on the benchmark dataset, for the adjacency-list (violet) and adjacency-map (green) representation, on random directed graphs with (**top** to **bottom**) $n = 8, 16, 32, 64, 128, 256$ vertices and $1, \ldots, n(n-1)$ directed edges.

4. Discussion

We have implemented three graph algorithms (graph construction, breadth-first graph traversal, and universal sink) using a small collection of abstract operations, with both the adjacency-list and the adjacency-map representation, and run them upon a benchmark dataset of random directed graphs with $n = 8, 16, 32, 64, 128, 256$ vertices and a number of directed edges ranging from $m = 1$ to the maximum possible number $m = n(n-1)$ of directed edges. The abstract operations take $O(1)$ worst-case time with the adjacency-list representation and $O(1)$ expected time with the adjacency-map representation, with the only exception of the adjacency test, which takes worst-case linear time in the degree of the vertices with the adjacency-list representation. With the adjacency-map representation, the abstract operations used in the algorithm for graph construction require dictionary lookup and update, and the abstract operations used in the algorithms for breadth-first graph traversal and for finding a universal sink require dictionary lookup, iteration over dictionary keys, and iteration over dictionary values.

The experimental results show that graph construction is slower (by a small constant factor) with adjacency maps than with adjacency lists. This can be explained by the $O(1)$ amortized running time of the underlying update operations on dynamic lists and dictionaries. Adding a new vertex to a graph involves one list-append operation with adjacency lists, and one insertion in a dictionary and the creation of two new dictionaries with adjacency maps, while adding a new edge to a graph involves three list-append operations with adjacency lists and two insertions in a dictionary with adjacency maps. Nevertheless, extending the adjacency-map representation with an operation to build a graph from a list of edges, instead of adding vertices and edges one-by-one to an initially empty graph, might result in a faster graph construction algorithm.

Experimental results also show that graph algorithms that do not test adjacencies (breadth-first graph traversal) run faster (by a small constant factor) with adjacency maps than with adjacency lists, and graph algorithms that test adjacencies (universal sink) run much faster (by one or two orders of magnitude) with adjacency maps than with adjacency lists. These results further reinforce the choice of the adjacency-map representation over the adjacency-list representation in recent textbooks [5,8] and software libraries [9].

While the experimental results were obtained with a Python implementation of the algorithms and graph data structures, adjacency maps can be easily implemented in any modern programming language, as they only need dictionaries as the underlying data structures. However, it is possible that the differences in running time between adjacency lists and adjacency maps become smaller with compiled programming languages once the overhead of compiling the source code into bytecode by the Python interpreter is removed. The influence of the programming language in the efficiency of the adjacency-map representation is an open line of future research.

Funding: This research was partially supported by the Spanish Ministry of Science, Innovation and Universities and the European Regional Development Fund through project PGC2018-096956-B-C43 (FEDER/MICINN/AEI), and by the Agency for Management of University and Research Grants (AGAUR) through grant 2017-SGR-786 (ALBCOM).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The Python script used to generate the 86,856 random directed graphs used in this study is available on request from the corresponding author.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Valiente, G. Trading uninitialized space for time. *Inf. Process. Lett.* **2004**, *92*, 9–13. [CrossRef]
2. Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C. *Introduction to Algorithms*, 3rd ed.; MIT Press: Cambridge, MA, USA, 2009.

3. Mehlhorn, K.; Näher, S. *The LEDA Platform of Combinatorial and Geometric Computing*; Cambridge University Press: Cambridge, UK, 1999.
4. Siek, J.G.; Lee, L.Q.; Lumsdaine, A. *The Boost Graph Library: User Guide and Reference Manual*; Addison-Wesley: Reading, MA, USA, 2001.
5. Valiente, G. *Algorithms on Trees and Graphs*, 2nd ed.; Texts in Computer Science; Springer Nature: Cham, Switzerland, 2021.
6. Tarjan, R.E. Depth-first search and linear graph algorithms. *SIAM J. Comput.* **1972**, *1*, 146–160. [CrossRef]
7. Tarjan, R.E. *Data Structures and Network Algorithms*; CBMS-NSF Regional Conference Series in Applied Mathematics; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 1983; Volume 44.
8. Goodrich, M.T.; Tamassia, R.; Goldwasser, M.H. *Data Structures and Algorithms in Python*; John Wiley & Sons, Inc.: Hoboken, NJ, USA, 2013.
9. Hagberg, A.A.; Schult, D.A.; Swart, P.J. Exploring network structure, dynamics, and function using NetworkX. In Proceedings of the 7th Python in Science Conference, Pasadena, CA, USA, 19–24 August 2008; Varoquaux, G., Vaught, T., Millman, J., Eds.; SciPy.org: Pasadena, CA, USA, 2008; pp. 11–16.
10. Bollobás, B. *Random Graphs*, 2nd ed.; Number 73 in Cambridge Studies in Advanced Mathematics; Cambridge University Press: Cambridge, UK, 2001.
11. Coolen, A.C.C.; Annibale, A.; Roberts, E.S. *Generating Random Networks and Graphs*; Oxforf University Press: Oxford, UK, 2017.