Graph Algorithms

Josef Barabash, Will Zahary Henderson

April 2, 2022

This is an augmented transcript of two lectures given by Luc Devroye on the 29th and 31st of March 2022 for the Honours Algorithms and Data Structures class (COMP 252, McGill University). The subject was graph algorithms.

Graph Definitions

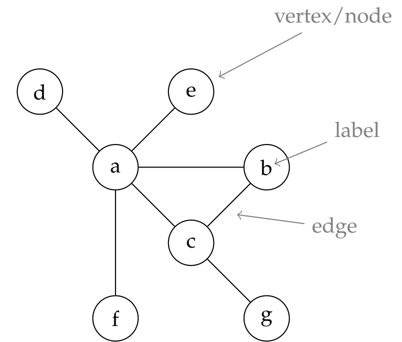


Figure 1: A graph with seven vertices and seven edges. The degree of **c** is 3.

Definition 1. A **graph**, $G = (V, E)$, is an abstract data type made up of a set of **vertices** (also called nodes) denoted V, and a set of **edges** denoted by E.

Definition 2. An **edge** can be thought of as a connection between a pair of vertices.

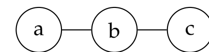


Figure 2: A connected component; also a path from **a** to **c**.

Definition 3. A **path** from vertex a to vertex b is a subset of distinct edges that connect the two vertices. A path is called a **cycle** when it starts and ends at the same vertex.

Definition 4. A **complete graph** of n vertices, denoted K_n, is a graph where all pairs of vertices have an edge, giving $\binom{n}{2}$ total edges.

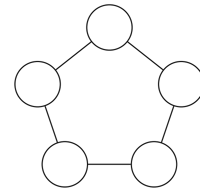


Figure 3: The cycle C_5.

Definition 5. An **undirected graph** has edges that do not specify directions. If i and j are two connected vertices in an undirected graph, (i, j) and (j, i) represent the same edge. Note that $|E| \leq \binom{|V|}{2}$.

Definition 6. A **directed graph** has edges with directions indicating a "to" and "from" relationship between a pair of vertices. The edge from vertex i to vertex j is denoted (i, j), and $(i, j) \neq (j, i)$.

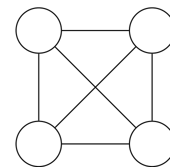


Figure 4: The complete graph K_4.

Definition 7. The **degree** of a vertex i, d_i, is equal to the number of neighbors it has. Note the relationship $\sum_{i \in V} d_i = 2|E|$. In a directed graph, we specify the **in-degree** for edges pointing toward a vertex, and the **out-degree** for edges pointing away from a vertex.

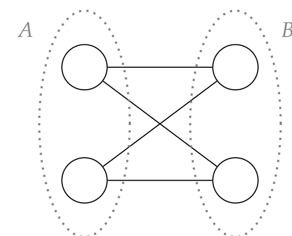


Figure 5: C_4 is an example of a bipartite graph.

Definition 8. A graph is **bipartite** if the set of vertices V can be constructed as the disjoint union $A \sqcup B$ and the set of edges E is a subset of the Cartesian product $A \times B$.

Remark 9. For the rest of this document, and in general, "linear time" in the context of graphs refers to $\mathcal{O}(|V| + |E|)$.

Implementations

We can implement a graph with a few different data structures. Namely, adjacency matrices and adjacency lists.

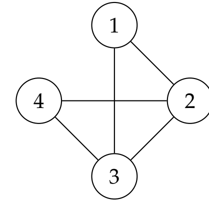
Adjacency Matrix

An **adjacency matrix** A stores whether or not an edge is present between two vertices i and j in the form of a two-dimensional matrix. Where $n = |V|$, we define A as

$$A = \{A[i,j]\}_{i,j=1}^{n}$$

such that

$$A[i,j] = \begin{cases} 1 & \text{edge } (i,j) \text{ present} \\ 0 & \text{otherwise} \end{cases}.$$

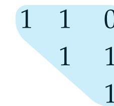


	1	2	3	4
1	0	1	1	0
2	1	0	1	1
3	1	1	0	1
4	0	1	1	0

Figure 6: A graph and its corresponding adjacency matrix implementation.

Example 10. See fig. 6 for an example of an adjacency matrix for an undirected graph. Notice that this matrix is *symmetric*; this is true of all undirected graphs. Notice also that the main diagonal is filled with zeroes; this is true of all graphs where loops are not allowed.

Any adjacency matrix with the same symmetric property as in Example 9 (i.e., an adjacency matrix for an undirected graph) can be represented linearly in the form of a binary vector. See fig. 7 for the binary vector representation of Example 9.



$$\equiv (110111)_2 = (55)_{10}$$

Figure 7: Rewriting the adjacency matrix in fig. 6 as a binary vector and as a decimal vector.

Exercise 11. Using the binary vector representation, perform:

1. Is edge (i,j) present?
2. Add edge (i,j).
3. Delete edge (i,j).
4. Find all neighbors of node i.

Exercise 12. A **sink** has in-degree $n-1$ and out-degree 0 (see fig. 8). Given an adjacency matrix, determine in $\mathcal{O}(|V|)$ time whether some graph G has a sink.

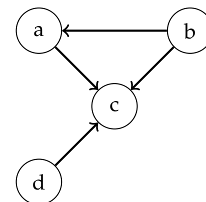


Figure 8: Vertex **c** is a sink.

Adjacency List

The standard implementation of an **adjacency list** is an array of size $|V|$ of pointers to linked lists containing the neighbors of each vertex.

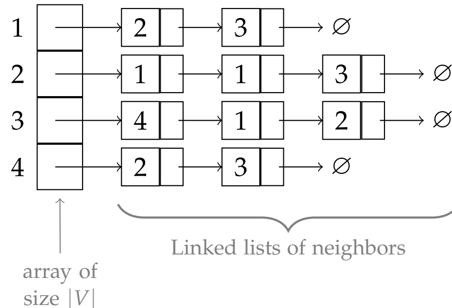


Figure 9: The adjacency list implementation of the graph from fig. 6.

Remark 13. It is important to note that the adjacency matrix uses $\Theta(|V|^2)$ space, while the adjacency list only requires $\mathcal{O}(|V| + |E|)$ space.

Exercise 14. Order these linked lists in linear time.

Huge Graphs

When graphs reach a particularly large scale, it may be impossible to store them as an adjacency matrix or an adjacency list. In these cases, we often have only local information:

$$\forall v \in V, \text{ we know the neighbors of } v.$$

Some examples of such a situation are as follows.

Example 15. Let V be the set of all websites. On a webpage v, every link from v to another webpage w forms the directed edge (v, w) on the web graph.

Example 16. There is a bipartite graph between the set of Facebook groups and Facebook users; edges are shared between a user and the groups to which it belongs.

Example 17. There is a directed graph mapping "follower" relationships on social media. If user a follows user b, the edge (a, b) exists.

Example 18. All positions in a strategy game (e.g., chess) can be seen as a graph, where V is the set of all possible game states, and E corresponds to the set of moves. Note that Claude Shannon devised a lower bound of 10^{120} possible games of chess.[1]

[1] Shannon [1950]

Depth-First Search

Definition 19. A **depth-first search** follows a strategy whereby the order of vertex traversal is determined by visiting the next not-yet-visited vertex connected to the current vertex.[2]

[2] Cormen et al. [1989]

For every vertex $u \in V$, the following attributes will be stored:

- color[u]: the color of a vertex belongs to the set {white, gray, black} and represents its "status" of visitation in the traversal. A *white* vertex has not yet been visited, a *gray* vertex has been iterated over once, and a *black* vertex has been iterated over twice (and will not be visited again).
- d[u]: the time of discovery of u.
- f[u]: the time when finished with u.
- p[u]: the parent vertex of u.

When performing a depth-first search, a "clock" is running; at every iteration in the search, this clock is incremented. This is where the notion of *time* arises for the attributes d[u] and f[u].

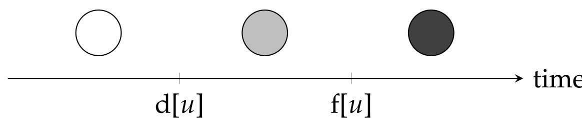


Figure 10: Visualizing color and time in a depth-first search.

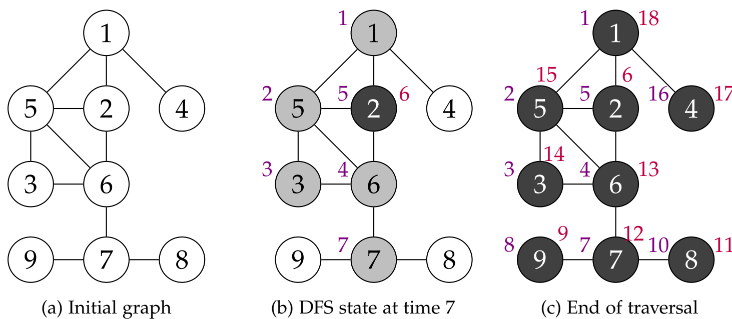


Figure 11: An example of a depth-first search traversal. Purple text represents d[u], magenta text represents f[u]. The order of this DFS is determined by the d[u] values, and is $\{1, 5, 3, 6, 2, 7, 9, 8, 4\}$. Note that all times, the gray nodes form a chain as each gray node represents a recursive call.

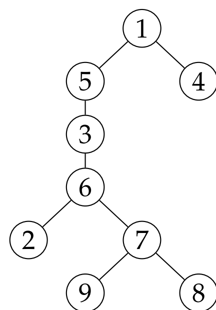


Figure 12: The depth-first search tree corresponding to the traversal in fig. 11.

The Algorithm

The setup for the algorithm requires initializing the time to zero, setting all vertex colors to white, and setting all parent pointers to *nil*. To perform a depth-first search, we make sure to visit all connected components.

```
1   time ← 0
2   ∀ u do: (color[u], p[u]) ← (white, nil)
3   ∀ u do: if color[u] = white then DFS(u)
```

DFS(u):

```
1    time ← time+1               }
2    color[u] ← gray             }  done once
3    d[u] ← time                 }
4    ∀ v adjacent to u do
5        if color[v] = white then                       }
6            p[v] ← u        }                          }  done degree[u]
7            DFS(v)          }  done once               }  times
8    time ← time+1               }
9    color[u] ← black            }  done once
10   f[u] ← time                 }
```

It is clear that the bottleneck in this algorithm occurs at line 5. Thus, the total cost of this operation is $\sum_{u \in V}$ (degree[u]+1). Because each edge is counted twice when summing the degrees of all nodes, the resulting cost is $2|E| + |V| = \Theta(|E| + |V|)$.

Theorem 20 (White path theorem)**.** *v is a descendant of u in the DFS tree $\iff$ at time d[u], there exists a white path connecting u to v.*

Remark 21 (Nesting)*.* Let u and v be two vertices in a graph such that d[u] > d[v] in a depth-first search. The interval where u is gray is either disjoint from the interval where v is gray (i.e., d[v] < f[v] < d[u] < f[u]), or is nested within the interval where v is gray (i.e., d[v] < d[u] < f[u] < f[v]). Nesting is understood by considering the recursive nature of depth-first search.

Edge Classification

For an edge (u, v), we classify the edge when it is considered for the first time in the DFS traversal.

Remark 22. Every edge is looked at *once* in a directed graph and *twice* in an undirected graph.

Let u and v be connected vertices such that $\mathrm{d}[u] < \mathrm{d}[v]$. Without loss of generality, we define the following classes:

- If v is white, (u, v) is a *tree edge*.
- If v is gray, (u, v) is a *back edge*.
- If v is black, (u, v) is a *cross edge* or a *forward edge*[3].

[3] Specifically, (u, v) is a forward edge if v is a non-child descendant of u.

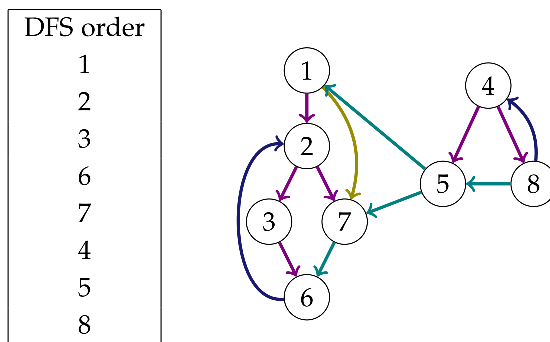
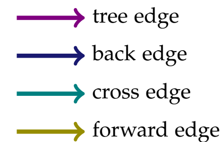


Figure 13: Edge classification in a directed graph. See color legend below.



Theorem 23. *In an undirected graph, all edges are tree or back edges.*

Proof. Let u and v be vertices such that $\mathrm{d}[u] < \mathrm{d}[v]$, so v is a descendant of u. If v is looked at for the first time from u, then (u, v) is a tree edge. Otherwise, if u is looked at for the first time from v, then (u, v) is a back edge since u is gray at this time. □

Theorem 24. *In a directed graph,*

$$\begin{aligned} &\textit{a cycle exists} \\ \iff &\textit{ there exists one DFS with a back edge} \\ \iff &\textit{ for all DFS, there exists a back edge.} \end{aligned}$$

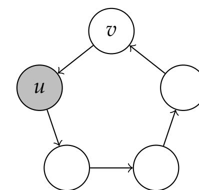


Figure 14: The cycle C. u is the first node discovered in the cycle.

Proof. We will prove the implication that if G has a cycle, then every DFS traversal has a back edge. The two other implications are trivial. Let u and v be two adjacent vertices in a cycle C such that at time $\mathrm{d}[u]$, there exists a white path from u to v, and so v is a descendant of u. Since u is visited first, it is already gray when the edge (v, u) is traversed. Thus, (v, u) is a back edge. □

Euler Graphs

Definition 25. An **Euler tour** is a cyclical tour that visits all edges in a graph exactly once, and returns to the starting point of the tour. If such a tour exists in a graph G, then G is said to be **Eulerian**.

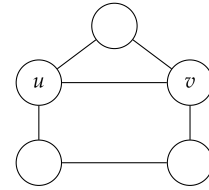


Figure 15: A non-Eulerian, undirected graph. Vertices u and v have odd degrees.

Remark 26. The following properties are true of Euler tours:

- In an undirected graph, an Euler tour exists if and only if all degrees are even
- In a directed graph, an Euler tour exists if and only if the in-degree equals the out-degree for all vertices in the graph

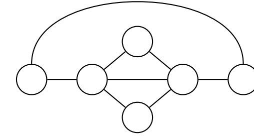


Figure 16: An Eulerian, undirected graph. All vertices have even degrees.

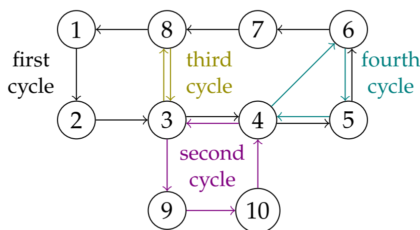
Definition 27. An **Euler walk** is a walk that is not allowed to visit an edge twice, and returns to to the start node.

Remark 28. Every Euler walk started at vertex u must return to u.

EULERWALK(u): / *returns a queue* $Q = u, \ldots, u$ /
1 MAKENULL(Q) / *initializes a queue* Q /
2 ENQUEUE(u, Q); $v \leftarrow u$
3 **repeat**
4 $v \leftarrow$ DELETE–FROM–FRONT–OF(adjacency **list** of v)
5 ENQUEUE(v, Q)
6 **until** $v = u$
7 **return** Q

EULERTOUR(G):
1 MAKENULL(S) / *initializes a stack* S /
2 PUSH($1, S$) / 1 *is the start of the Euler tour* /
3 **while** $|S| > 0$:
4 $u \leftarrow$ POP(S)
5 **if** degree$[u] = 0$ **then** output u
6 **else** PUSH(**EULERWALK**(u), S) / *push elements onto* S /

The complexity of this algorithm is linear, $\mathcal{O}(|V| + |E|)$.



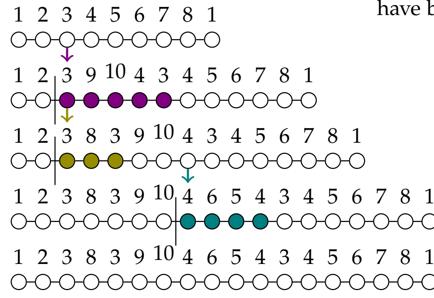


Figure 17: An example of the EULERTOUR algorithm on a graph. At each row, the contents left of the vertical line have been outputted.

Breadth-First Search

Definition 29. A **breadth-first search** follows a strategy whereby the order of vertex traversal is determined in a level-by-level order, traversing from shortest to longest path distance from the starting vertex.

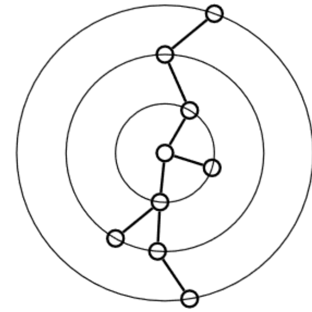


Figure 18: A visualization of the idea of "levels" in a graph, where the center vertex is the starting point.

In a breadth-first search on the graph in fig. 18, the order of traversal depends on each "level" (visualized as a ring) of the graph. First, the three vertices on the innermost ring will be visited, then the three on the second ring, and then finally the two on the outermost ring. To achieve this, we use a queue Q.

For every vertex $u \in V$, the following attributes, much alike those of the depth-first search, will be stored:

- color[u]: the color of a vertex belongs to the set {white, gray, black} and represents its "status" of visitation in the traversal. A *white* vertex has not yet been visited, a *gray* vertex is being dealt with, and a vertex we are done with is *black*.
- d[u]: the minimal path distance of u from the starting point s, i.e., the "level" on which u is located with respect to s.
- p[u]: the parent vertex of u.

Remark 30. To further understand the notion of *levels*, note that the starting vertex is on level 0, the direct children of the starting vertex are on level 1, the grandchildren of the starting vertex are on level 2, and so on.

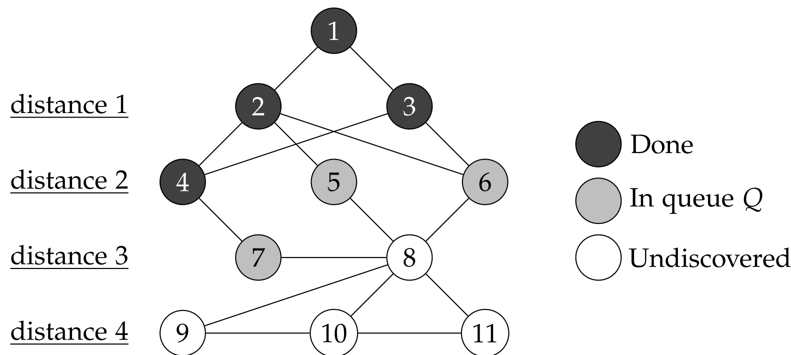


Figure 19: The status of a breadth-first search after having visited vertices $1, 2, 3, 4$.

The Algorithm

The setup for the algorithm is very similar to that of the depth-first search. We initialize the graph by setting all vertex colors to white, setting all parent pointers to *nil*, and setting all distances to ∞.

```
BFS(s):  / s is the starting vertex /
1    ∀ u do: (color[u], p[u],d[u]) ← (white, nil, ∞)
2    d[s] ← 0; color[s] ← gray
3    MAKENULL(Q); ENQUEUE(s, Q)
4    while |Q| > 0
5        v ← DEQUEUE(Q)
6        for all neighbors w of v do
7            if color[w] = white then
8                color[w] ← gray
9                ENQUEUE(w, Q)
10               d[w] ← d[v] + 1
11               p[w] ← v
12       color[v] ← black
```

Similarly to depth-first search, the breadth-first search algorithm is $\Theta(|E| + |V|)$ as the **if** statement on line 7 is performed $2|E|$ times.

Note that breadth-first search also constructs a breadth-first search tree, i.e., a tree in which the path from node u to root s is the (possibly not unique) shortest path from u to s. Its length is d[u].

Exercises

Exercise 31. Determine in $\mathcal{O}(|V| + |E|)$ time if a graph G is bipartite. *Hint: use BFS.*

Exercise 32. The **hypercube** H_d is a graph with $V = \{0,1\}^d$ (all bit vectors of length d), and $E = \{(u,v) \in V \times V : \text{Hamming}(u,v) = 1\}$[4]. Show that H_d is bipartite.

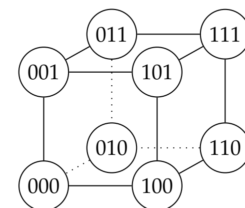


Figure 20: The hypercube H_3.

[4] Recall that the *Hamming distance* is the number of bits that differ position-wise between two binary numbers.

References

T.H Cormen, C.E. Leiserson, R.L.Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 1989. ISBN 9780262033848.

C. E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41:4, March 1950.