4 Graphs

A graph is an ordered pair (V, E), where V is a set of vertices, E is a set of edges, and each edge relates two vertices together. In the context of the algorithms introduced in this course, we assume that for every pair of vertices, there is at most one edge connecting them[3].

Figure 1 shows the drawing of a graph. In drawings, the shape, location, or size of a vertex and the length of an edge (e.g. the length of the line) do not carry any information. For the sake of readability, graphs must be drawn in such a way that edges do not cross each other but this is not always possible.

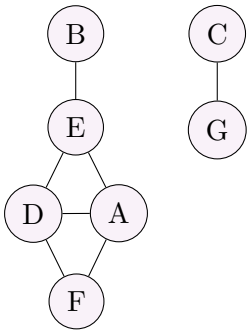


Figure 1: A graph with 7 vertices and 7 edges

We would like to look at graph algorithms in an abstract way. This would enable us to solve a wide range of problems using the same general tool. For example, the graph presented in Figure 1 may be related to a problem in which vertices represent cities and edges represent direct (two-way) flights between pairs of cities. We will see that, for example, a BFS traversal starting from vertex D can be used to find an itinerary with the smallest number of connecting flights from city D to any other city. If the graph in Figure 1 was instead related to a problem in which vertices represented people and edges represented mutual friendship between two people, then the same BFS traversal could have been used to determine the degrees of separation between person D and other people.

4.1 Concepts and terminology

In this section, we look at some graph concepts that are encountered in this course.

[3] We do not study multi-graphs which allow multiple edges between the same pair of vertices.

4.1.1 Size of graphs

We measure the size of a graph in terms of the numbers of vertices and edges. We use n to denote the number of vertices (i.e. $|V| = n$) and m to denote the number of edges (i.e. $|E| = m$). The inequality $m \leq n^2$ always holds because edges connect pairs of vertices, and therefore, there are at most n^2 edges in a graph (which is the case when all pairs of vertices are connected).

We say a graph is *sparse* when $m \in O(n)$. Many real-world graphs (e.g. friendship graphs or road networks) fall in this category. We say a graph is *dense* when $m \in \Theta(n^2)$; that is, a vertex is connected to most other vertices.

4.1.2 Directed and undirected graphs

A graph can be classified as *directed* or *undirected*. In directed graphs, all edges are directed. In undirected graphs, all edges are undirected. In this course we do not consider mixed graphs were some edges are directed and some undirected.

4.1.2.1 Directed graphs

In directed graphs, edges specify one-way relationships. For example, if we want to model a network of roads in which some roads may be one-way, we use a directed graph. If there is a one-way connection from a to b, then the graph will have a directed edge from a to b. If the connection between b and c is two-way, then the graph will have two edges: one from b to c another from c to b.

In directed graphs $E \subseteq V \times V$. This means that edges are *ordered pairs* (two-tuples of the form (u, v) where $u, v \in V$). The edges are ordered because (u, v) (which means $u \to v$) is different from (v, u) (which means $v \to u$).

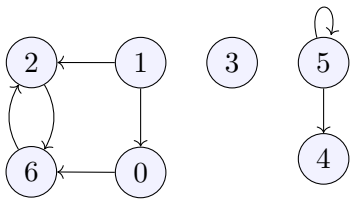


Figure 2: A directed graph $G = (V, E)$, where $V = \{0, 1, \ldots 6\}$ and $E = \{(1, 0), (2, 6), (5, 4), (1, 2), (5, 5), (0, 6), (6, 2)\}$

A directed edge (u, v) can be used to reach vertex v from vertex u but cannot be used to reach vertex u from v. Figure 2 shows an example of a directed graph. Note that directed edges are drawn with arrows.

4.1.2.2 Undirected graphs

In undirected graphs, edges specify two-way (symmetric) relationships. For example, if we want to model a network of roads in which all roads are two-way, we use an undirected graph.

In undirected graphs, $E \subseteq \{\{u, v\} : u, v \in V\}$. This means that edges are *unordered pairs*. The edges are unordered because $\{u, v\}$ (which means

) is the same as $\{v, u\}$. Note that, we are using curly braces to show that $\{u, v\}$ is a set, which implies that the order of elements does not matter. An undirected edge $\{v, u\}$ can be used to reach u from v or to reach v from u; that is, it is equivalent to having two directed edges (v, u) and (u, v) in the directed version of the graph.

Figure 3 shows an example of a directed graph. Note that, undirected edges are drawn as plain line segments (without arrow heads).

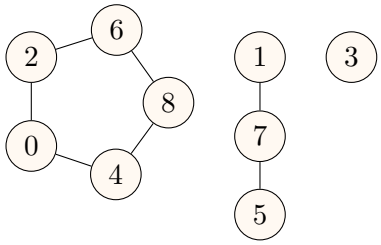


Figure 3: An undirected graph $G = (V, E)$, where $V = \{0, 1, \ldots 8\}$ and $E = \{\{7, 1\}, \{2, 6\}, \{0, 4\}, \{0, 2\}, \{8, 6\}, \{4, 8\}, \{7, 5\}\}$

4.1.3 Weighted graphs

A weighted graph is a triple $(V, E, \mathcal{W})$, where $\mathcal{W} : E \to \mathbb{R}$. The additional component is a function $\mathcal{W}$ that maps each edge to a real value (i.e. assigns a real number to each edge). Weighted graphs can be directed or undirected. Figure 4 shows an example of a weighted undirected graph.

We use weighted graphs to associate a numeric value to each edge and later use this information to find an optimal solution. For instance, in a graph representing a road network, where the vertices are junctions and the edges are road segments, weights can represent the length of road segments. Using a shortest path algorithm, we can find a path from a source vertex to a target vertex with the smallest total weight.

4.1.4 Paths and cycles

Given a graph, we define a *path* to be a sequence of vertices $(v_0, v_1, \ldots, v_k)$ such that

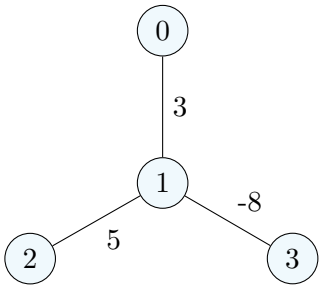


Figure 4: A weighted undirected graph $G = (V, E, \mathcal{W})$, where $V = \{0, 1, 2, 3\}$, $E = \{\{2, 1\}, \{1, 0\}, \{3, 1\}\}$, $\mathcal{W}(\{1, 3\}) = -8$, $\mathcal{W}(\{1, 2\}) = 5$, and $\mathcal{W}(\{0, 1\}) = 3$

i) for each i in $\{0, 1, \ldots, k-1\}$, we have $(v_i, v_{i+1}) \in E$ or $\{v_i, v_{i+1}\} \in E$; and

ii) the path does not use any edge more than once.

We call the first vertex in the path (v_0), *source*, and the last vertex (v_k), *destination*.

The *length* of a path is the number of edges across the path. For instance, the length of the path $(v_0, v_1, \ldots, v_k)$ is k.

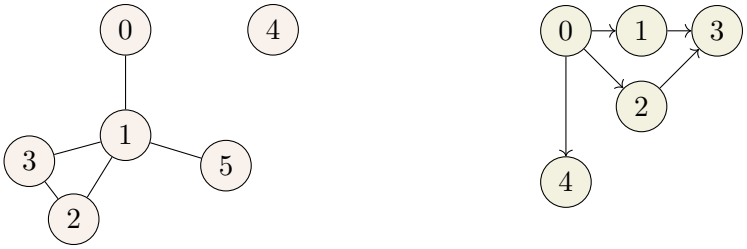


Figure 5: Left: an undirected graph in which $(3, 2, 1, 0)$ is a path of length three; $(2, 3, 1, 2)$ is a path of length three; $(0, 1, 2, 3, 1, 5)$ is a path of length five; $(0, 1, 0)$ is not a path (because the edge $\{0, 1\}$ is used more than once); and the only path that has 4 as its source or destination is (4) which is of length zero. The graph is cyclic. **Right:** a directed graph in which there are two paths from 0 to 3 (both of length two). This graph is a DAG.

A *cycle* is a path of length at least 1 whose source and destination are the same vertex. A graph that contains one or more cycles is called *cyclic*. A graph that does not have any cycle is called *acyclic*. A directed graph that does not have any cycle is called a DAG (directed acyclic graph).

A *loop* or *self-loop* is a (directed or undirected) path of length one that connects a vertex to itself.

See Figure 5 for some examples of paths and cycles.

Remarks:

- Every loop is a cycle.
- There is no undirected graph that has a cycle of length two[4].

4.1.5 Subgraphs and components

4.1.5.1 The empty graph

The graph $(\{\}, \{\})$ is the empty graph. The empty graph does not have any vertices and therefore does not have any edges. The empty graph can be considered to be directed or undirected, weighted or unweighted.

4.1.5.2 Subgraphs

A graph $G' = (V', E')$ is a subgraph of a graph $G = (V, E)$, if $V' \subseteq V$ and $E' \subseteq E$. Figure 6 shows a few examples of subgraphs of a graph.

Remarks:

- The definition implies that a subgraph is a valid graph in its own right.
- For weighted graphs, the weight values remain unchanged in the subgraph.
- Every graph is a subgraph of itself.
- The empty graph is a subgraph of every graph.

4.1.5.3 Components of a graph

Consider an undirected graph $G = (V, E)$. A graph $G' = (V', E')$ is a component of G if:

i) G' is a non-empty subgraph of G;

ii) every pair of vertices in G' are connected by a path in G;

[4] In our definition of graphs, we did not allow parallel edges. If parallel edges were allowed then there could be an undirected cycle of length two when there are two undirected edges between two vertices.

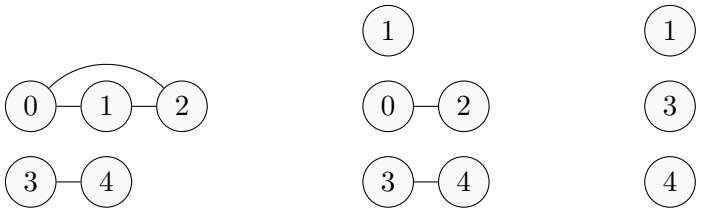


Figure 6: A graph (left) and two (out of many) of its subgraphs (middle and right)

iii) for all $v' \in V'$ and for all $u \in (V \setminus V')$, there is no path between v' and u in G; and

iv) E' contains all edges in E that involve vertices in V'.

In other words, a component of a graph is a maximal connected subgraph. See Figure 7 for an example of a graph with multiple components.

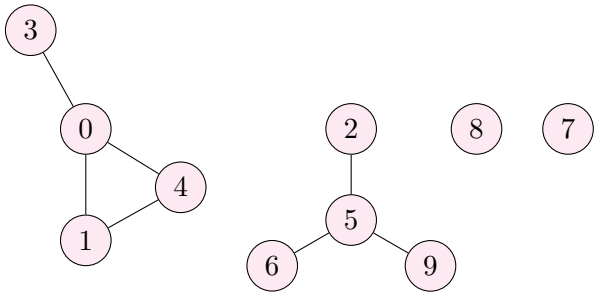


Figure 7: A graph with four components. One component includes vertices 0, 1, 3, 4, and all the edges connecting them. Another component includes vertices 2, 5, 6, 9 and all the edges connecting them. The other two components are vertices 7 and 8.

Since the empty graph does not have any non-empty subgraph, by definition it does not have any components.

4.1.6 Trees and forests

A *tree* is a (directed or undirected) graph in which one vertex is considered to be the *root* and for every vertex in the graph there is one (and only one) path between the vertex and the root[5]. In a directed tree, all the edges are either towards the root or away from it. Figure 8 shows some examples of trees.

[5]In graph theory, a tree does not need to have a distinct root. What we are defining here is known as a *rooted tree* in graph theory.

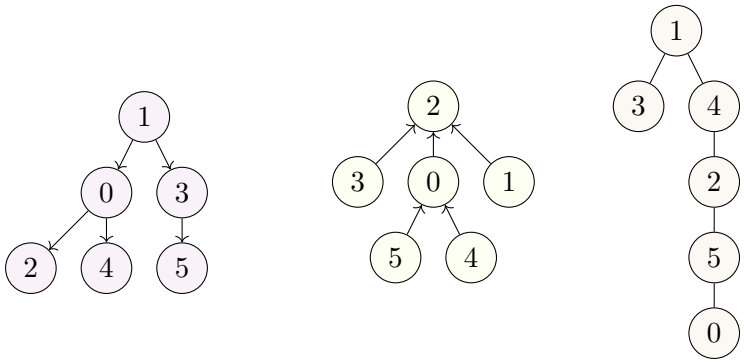


Figure 8: Three examples of trees: two directed trees (left and middle) and one undirected tree (right)

A *forest* is a graph whose every component is a tree. Figure 9 shows an example of a forest.

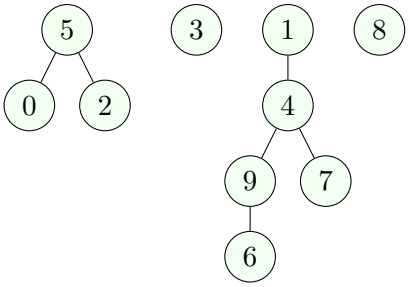


Figure 9: This graph is a forest. The graph has four components all of which are trees. Note that, two of these trees are just single leaves.

Remarks:

- Every graph that is a single vertex is a tree.
- Every tree is a forest.
- Every forest is a graph.

4.2 Textual representation of graph objects

In order for computer programs to be able to take graphs as input, we must define a precise format for graphs. In this course, we use a string-based (text)

format. A textual format (as opposed to binary) allows graph inputs to be easily inspected and edited by humans.

For a graph with n vertices, we assume that the vertices have been assigned natural numbers from 0 to $n - 1$. The assignment of integers to vertices is arbitrary and does not carry any semantics. For example, vertex 9 is not greater (or more important) than vertex 8. We only have to guarantee that there is a one-to-one mapping from $\{0, 1, \ldots, n - 1\}$ to the n vertices in our application domain (e.g. cities in a road network).

4.2.1 Specification

The string will have one or more lines. The first line is the header which describes the graph. The remaining lines (if any) describe the edges of the graph.

Each line has a number of fields separated from each other by one or more blank spaces. There may be spaces before the first field and after the last field. Each line is terminated by a newline character.

4.2.1.1 The header

The header describes the type of the graph and the number of vertices.

The first field is either the character `D` indicating a directed graph, or the character `U` indicating an undirected graph.

The second field of the header is a natural number n indicating the number of vertices in the graph. All the vertices in the graph are numbered from 0 to $n - 1$.

If the graph is weighted, there will be a third field: the character `W`.

4.2.1.2 The edges

If the graph string has more than one line, each remaining line (starting from the second line) describes an edge. The first two fields of an edge line are two natural numbers between 0 and $n - 1$, describing an edge between the two vertices identified with these numbers.

If the graph is directed, the two numbers must be interpreted as an ordered pair describing an edge from the first vertex to the second vertex.

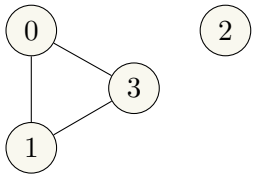
If the graph is undirected, the two numbers must be interpreted as an unordered pair describing an edge between the two vertices. An undirected edge appears only once in the text. For instance, in an unweighted undirected graph, if there is an edge between vertices 0 and 2, the text will contain either the line `0 2` or `2 0` but not both.

If the graph is weighted, there will be a third field which will be an integer indicating the weight of the edge.

Example 4.1. The following text describes an undirected graph (which is also unweighted since there is no `W` in the header) with four vertices and three edges.

```
U 4
0 1
3 1
0 3
```

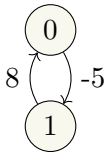
The drawing of the graph is



Example 4.2. The following describes a weighted directed graph with two vertices and two edges.

```
D 2 W
0 1 −5
1 0 8
```

The drawing of the graph is



Example 4.3. The following describes an unweighted directed graph with four vertices and no edge.

```
D 4
```

The drawing of the graph is

4.3 Graph data structures

When a computer program reads the textual description of a graph, it has to parse the input and then construct an appropriate data structure in the RAM so that graph algorithms can work with the graph.

A naive solution would be to construct two sets V and E similar to the formal (mathematical) definition we provided. This is not ideal because some operations become expensive. For example, it would take $\Theta(m)$ time to find edges that are connected to a certain vertex. For a dense graph this amounts to $\Theta(n^2)$.

Two common data structures for graphs are *adjacency lists* and *adjacency matrices*.

4.3.1 Adjacency matrix

An unweighted graph with n vertices can be represented by an $n \times n$ binary matrix A, such that

$$a_{i,j} = \begin{cases} 1, & \text{if } (i,j) \in E \text{ (in directed graphs) or } \{i,j\} \in E \text{ (in undirected graphs)} \\ 0, & \text{otherwise,} \end{cases}$$

where $a_{i,j}$ is the element in row i and column j.

Remarks:

- For an undirected edge $\{i,j\}$, both $a_{i,j}$ and $a_{j,i}$ are set to 1.
- The adjacency matrix of an undirected graph is symmetric.
- The elements on the main diagonal are 0 unless there is a loop.
- Instead of 0/1 other pairs of symbols (e.g. false/true) can be used.
- For weighted graphs, when there is an edge from i to j, $a_{i,j}$ is the weight of the edge. When there is no edge, $a_{i,j}$ is a special symbol like NULL (in Python `None`).
- For implementing the matrix in Python, one could use NumPy, or a list of size n where each element is a list of size n itself. The elements of the inner lists are 0/1 or weights depending on the type of the graph.

Example 4.4. Consider the following graph.

```
U 4
2 3
1 0
2 1
```

Using a list of lists in Python, the adjacency matrix is:

```
[
  [0, 1, 0, 0],
  [1, 0, 1, 0],
  [0, 1, 0, 1],
  [0, 0, 1, 0],
]
```

Example 4.5. Consider the following graph.

```
D 3 W
1 2 7
1 0 −3
2 1 9
2 2 1
0 2 0
```

Using a list of lists in Python, the adjacency matrix is:

```
[
  [None, None,  0],
  [  -3, None,  7],
  [None,    9,  1],
]
```

4.3.2 Adjacency lists

The idea behind the adjacency list data structure is to have n lists, one for each vertex, where the lists hold edges associated with each vertex. There are different variations of this data structure. The variation we use in this course is an array (list) Adj of length n, where $Adj[i]$ itself is a list containing directed edges from vertex i or undirected edges involving vertex i.

If (and only if) the graph has a directed edge (i, j) or an undirected edge $\{i, j\}$, then the list at $Adj[i]$ contains j. If the edge is weighted, then the list contains (j, w), where w is the weight of the edge connecting i to j.

Remarks The above definition implies the following:

- If the graph contains an undirected edge $\{i, j\}$, then $Adj[i]$ contains j and $Adj[j]$ contains i.
- The inner lists can have different lengths.
- The inner list for a vertex that is not connected to any vertices is empty.

Also note the following:

- In principle, the order of edges in the inner lists does not matter but for the sake of consistency and testing, the elements must appear in the order they appear in the textual representation of the graph.
- When implementing adjacency lists, in order to be able to use the same data structure for both weighted and unweighted graphs, the elements of the inner lists are always of the form (j, NULL), where NULL (or `None` in Python) in this context, indicates there is no weight.
- In some variations of this data structure, the outer collection is a dictionary instead of a list. This would allow identifying vertices with things other than natural numbers.
- In some variations of this data structure, the inner lists are linked lists.

Example 4.6. Consider the following graph.

U 4 W
3 1 −4
1 0 7

The adjacency list of this graph using Python lists is:

```
[
  [(1, 7)],
  [(3, -4), (0, 7)],
  [],
  [(1, -4)]
]
```

Example 4.7. Consider the following graph.

D 5
2 3
0 4
2 4

The adjacency list of this graph using Python lists is:

```
[
  [(4, None)],
  [],
  [(3, None), (4, None)],
  [],
  []
]
```

4.3.3 On the order of edges in a graph

The mathematical representation of a graph (as a set of vertices and a set of edges) does not specify an order for edges (outgoing from a vertex). However, when working with graphs on a machine, there is always an order (even if it is unspecified). The order is determined by the data structures and procedures used to store and retrieve graph objects.

In the context of this course, the order of vertices in the adjacency list (which is based on the order in which edges appear in the textual representation) dictates which vertex to visit first. For example[6], in the graph

```
U 3
0 1
1 2
2 0
```

a DFS starting from 0 produces `[None, 0, 1]` as the parent array. Whereas in the graph

```
U 3
2 0
0 1
1 2
```

which is mathematically the same graph but has a different adjacency list, a DFS starting from 0 yields `[None, 2, 0]`.

4.3.4 Comparison between adjacency matrices and lists

The following table provides a comparison between the space complexity of adjacency matrices and lists and the time complexity of various operations on these two data structures.

Aspect	Adj. matrix	Adj. list
Determine whether there is an edge (i, j) in the graph	$\Theta(1)$	$O(n)$
Adding an edge	$\Theta(1)$	$O(n)$
Deleting an edge	$\Theta(1)$	$O(n)$
Iterating over edges from a given vertex	$\Theta(n)$	$O(n)$
Iterating over all edges	$\Theta(n^2)$	$\Theta(n+m) = \Theta(max(n, m))$
Space	$\Theta(n^2)$	$\Theta(n+m) = \Theta(max(n, m))$

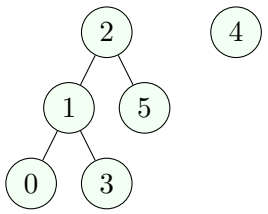
[6]The example makes forward references to the parent array and the DFS algorithm which will be introduced later.

Comments:

- Time for adding an edge in an adj. list is the sum of a time in $O(n)$ (to determine if the edge is already there) and a time in $\Theta(1)$ (to append the edge if necessary).
- Time for deleting an edge in an adj. list is the sum of a time in $O(n)$ (to find the edge) and either a time in $O(n)$ (for deletion in inner lists) or a time in $\Theta(1)$ (for deletion in inner linked lists).
- Since graph traversal involves iterating over all edges and many problems have a sparse graph, the adjacency list is very often the preferred data structure.

4.3.5 Data structure for forests

While we can use adjacency lists or matrices to represent any graph, for forests we can have a more convenient data structure that (as you will see later) makes working with them easier. Recall that, in a forest, every component is a rooted tree. Each vertex in a forest has a single "parent" or has no parent at all. Therefore, the edges of a forest can be described by a function *parent* : $V \to V \cup \text{NULL}$ which maps every vertex to its parent or a special symbol NULL, if the vertex does not have a parent. Since we use natural numbers to refer to vertices, the parent function is of the form *parent* : $\{0, 1, \ldots, n-1\} \to \{0, 1, \ldots, n-1\} \cup \text{NULL}$. Figure 10 provides an example.



$parent(0) = 1$
$parent(1) = 2$
$parent(2) = \text{NULL}$
$parent(3) = 1$
$parent(4) = \text{NULL}$
$parent(5) = 2$

Figure 10: A forest (left) and a function idparent encoding the forest (right)

Some data structures can be viewed as functions. For example, a dictionary can be seen as a function that maps its keys to its values. An array (Python list) is a function that maps natural numbers (from zero up to the length of the list minus one) to values. Since the parent (or predecessor) function of forests is like a lookup table (as opposed to requiring some computation to return a value) and since in some algorithms we want to able to gradually construct a forest (or a tree) as we explore a graph, it is appropriate to use one of these data structures for the parent function. In this course, we use a

parent array of length n, where the value of `parent[v]` is `u`, if $parent(v) = u$. In Python, we use `None` for NULL.

Example 4.8. The code below shows a Python array (list) that represents the parent function in Figure 10. Observe that `parent[i]` in the code has the same value as $parent(i)$ in the figure.

```
parent = [1, 2, None, 1, None, 2]
```

5 Graph traversal

Graph traversal is the process of traversing the edges of a graph in a particular order. The process provides useful information and data structures about the graph that can be used in various applications (for example, finding the number of components, shortest paths, and topological orderings).

5.1 Traversal concepts

5.1.1 States of vertices during traversal

In the traversal process, some (or all) of the vertices of the graph go through the following stages in order:

1. UNDISCOVERED: the vertex has not been encountered yet;
2. DISCOVERED : the vertex has been encountered (discovered), but the algorithm has not finished processing it yet;
3. PROCESSED: the algorithm has finished processing the vertex.

Initially all the vertices in the graph are undiscovered. The traversal starts from a given vertex which will be the first one to be discovered. A traversal algorithm follows the edges going out of discovered vertices (in a certain order) to discover more vertices. Once the traversal algorithm discovers all the vertices connected to a vertex, it marks the vertex as processed.

A graph traversal algorithm needs one or more data structures to keep track of the states of vertices during traversal. In this course, where vertices are identified by numbers $\{0, 1, \ldots, n-1\}$, one could use an array (list) *state* of size n, where each element is either UNDISCOVERED, DISCOVERED, or PROCESSED (or alternatively characters `U`, `D`, or `P`). Another alternative is to have two Boolean arrays *discovered* and *processed* each of size n. This would provide four states for each vertex, where one of them would be invalid (because $discovered[i]$ cannot be false if $processed[i]$ is true).

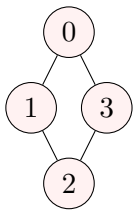
5.1.2 Predecessor tree

One of the data structures that is generated during a graph traversal is the *predecessor tree*. The root of the tree is the vertex at which the traversal starts. While processing vertex u, if we encounter an edge (u, v) (or $\{u, v\}$) such that v is undiscovered, the edge is added to the tree (which makes v a child of u). Those edges of the graph that appear in the tree are called *tree edges* and those that do not appear in the tree are called *non-tree edges*. The predecessor tree can be represented using the *parent* function. If $parent(v) = u$, then the edge (u, v) is a tree edge and u is the parent of v in the predecessor tree.

Example 5.1. Consider the following graph.

```
U 4
3 0
1 2
2 3
0 1
```

Suppose we perform a breadth-first traversal starting at vertex 0. First vertex 0 is discovered. Then vertices 1 and 3 are discovered, which make $\{0, 1\}$ and $\{0, 3\}$ tree edges. At this point, vertex 0 is processed. Next, from vertex 3, vertex 2 is discovered. This makes $\{3, 0\}$ a tree edge. In the next steps, 3, 1, and 2 are marked as processed. Figure 11 visualises the result.




```
[None, 0, 3, 0]
```

Figure 11: Running BFS on this undirected graph starting at vertex 0 produces the parent array on the right.

5.2 Breadth-first search

In BFS, vertices are discovered in order of increasing distance from the starting vertex. We break the BFS algorithm into two procedures:

- BFS-Tree: initialises the required data structures and calls the main loop of the algorithm;
- BFS-Loop: is the main loop of the algorithm and can be reused in BFS applications.

The traversal starts from a starting vertex s.

```
1  procedure BFS-Tree(Adj, s)
2    n ← number of vertices (the length of Adj)
3    state ← an array of length n filled with Undiscovered
4    parent ← an array of length n filled with null
5    Q ← an empty queue
6    state[s] ← Discovered
7    Enqueue(Q, s)
8    return BFS-Loop(Adj, Q, state, parent)
```

```
1  procedure BFS-Loop(Adj, Q, state, parent)
2    while Q is not empty
3      u ← Dequeue(Q)
4      for v in Adj[u]
5        if state[v] = Undiscovered
6          state[v] ← Discovered
7          parent[v] ← u
8          Enqueue(Q, v)
9      state[u] ← Processed
10   return parent
```

5.2.1 Analysis

The size of an input graph $G = (V, E)$ is measured by two metrics: $|V| = n$ and $|E| = m$. Recall that $m \leq n^2$.

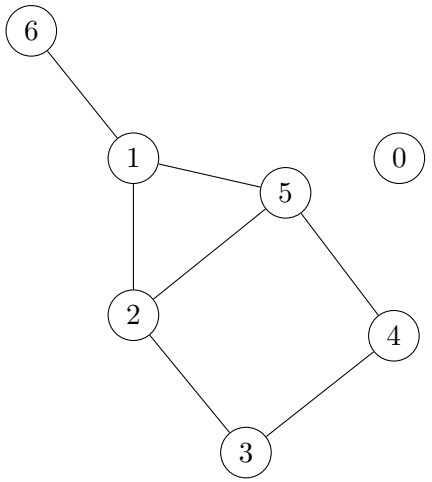
The time complexity of BFS-Tree up until BFS-Loop is called, is $\Theta(n)$.

The time complexity of BFS-Loop is proportional to the number of times the body of the inner loop is executed which is at most as many as the number of edges in the graph, that is, $O(m)$.

Therefore the complexity of BFS-Tree is in $O(n + m) = O(\max(n, m))$.

Example Trace of BFS

```
graph_str = """\
U 7
1 2
1 5
1 6
2 3
2 5
3 4
4 5
"""
```




```
adj_list = adjacency_list(graph_str)

0 [[],
1  [(2, None), (5, None), (6, None)],
2  [(1, None), (3, None), (5, None)],
3  [(2, None), (4, None)],
4  [(3, None), (5, None)],
5  [(1, None), (2, None), (4, None)],
6  [(1, None)]]

bfs_tree(adj_list, 1)
```

	State							Parent						
	0	1	2	3	4	5	6	0	1	2	3	4	5	6
deque([1])	'U'	'D'	'U'	'U'	'U'	'U'	'U'	None	None	None	None	None	None	None
deque([2, 5, 6])	'U'	'P'	'D'	'U'	'U'	'D'	'D'	None	None	1	None	None	1	1
deque([5, 6, 3])	'U'	'P'	'P'	'D'	'U'	'D'	'D'	None	None	1	2	None	1	1
deque([6, 3, 4])	'U'	'P'	'P'	'D'	'D'	'P'	'D'	None	None	1	2	5	1	1
deque([3, 4])	'U'	'P'	'P'	'D'	'D'	'P'	'P'	None	None	1	2	5	1	1
deque([4])	'U'	'P'	'P'	'P'	'D'	'P'	'P'	None	None	1	2	5	1	1
deque([])	'U'	'P'	'P'	'P'	'P'	'P'	'P'	None	None	1	2	5	1	1

Note: The first line shows the value of the variables before entering the while-loop (after initialisation). The remaining lines show the values after completion of each iteration of the while-loop. In the example above, the while-loop has iterated 6 times.

5.3 Depth-first search (DFS)

Similar to BFS, we break the DFS algorithm into two procedures: DFS-Tree and DFS-Loop. While in BFS we use a queue to achieve FIFO (first in, first out) in visiting vertices, in DFS we use a (call) stack to achieve LIFO (last in, first out) and thus proceed to the deepest discovered vertex first.

The traversal starts from a starting vertex s.

```
1  procedure DFS-Tree(Adj, s)
2    n ← number of vertices (the length of Adj)
3    state ← an array of length n filled with Undiscovered
4    parent ← an array of length n filled with null
5    state[s] ← Discovered
6    DFS-Loop(Adj, s, state, parent)
7    return parent
```

```
1  procedure DFS-Loop(Adj, u, state, parent)
2    for v in Adj[u]
3      if state[v] = Undiscovered
4        state[v] ← Discovered
5        parent[v] ← u
6        DFS-Loop(Adj, v, state, parent)
7    state[u] ← Processed
```

Each call to DFS-Loop starts a DFS traversal. Changes to *state* and *parent* are visible across calls (in programming terms, they are references to mutable objects).

The for-loop in DFS-Loop considers edges of the form (u, v). We are interested in a vertex v that has not been discovered before. A DFS is then started from v.

5.3.1 Analysis

The time complexity of DFS-Tree up until DFS-Loop is called, is in $\Theta(n)$.

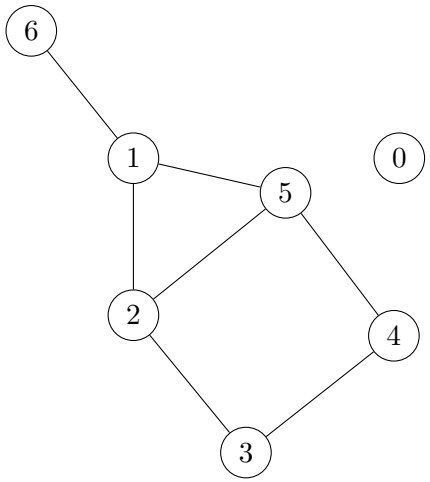
DFS-Loop is called at most n times because it is called when a vertex is discovered and each vertex can be discovered only once.

If DFS-Loop is called on a vertex u, then the for-loop iterates as many times as the number of outgoing edges from u. Therefore the total number of times the body of the for-loop is executed (for all vertices) is in $O(m)$.

Therefore the time complexity of DFS-Tree is in $O(n+m) = O(\max(n, m))$.

Example Trace of DFS

```
graph_str = """\
U 7
1 2
1 5
1 6
2 3
2 5
3 4
4 5
"""
```




```
adj_list = adjacency_list(graph_str)

0 [[],
1  [(2, None), (5, None), (6, None)],
2  [(1, None), (3, None), (5, None)],
3  [(2, None), (4, None)],
4  [(3, None), (5, None)],
5  [(1, None), (2, None), (4, None)],
6  [(1, None)]]

dfs_tree(adj_list, 1)
```

Call-Stack	State							Parent						
	0	1	2	3	4	5	6	0	1	2	3	4	5	6
[1]	'U'	'D'	'U'	'U'	'U'	'U'	'U'	None	None	None	None	None	None	None
[1, 2]	'U'	'D'	'D'	'U'	'U'	'U'	'U'	None	None	1	None	None	None	None
[1, 2, 3]	'U'	'D'	'D'	'D'	'U'	'U'	'U'	None	None	1	2	None	None	None
[1, 2, 3, 4]	'U'	'D'	'D'	'D'	'D'	'U'	'U'	None	None	1	2	3	None	None
[1, 2, 3, 4, 5]	'U'	'D'	'D'	'D'	'D'	'D'	'U'	None	None	1	2	3	4	None
[1, 2, 3, 4, 5]	'U'	'D'	'D'	'D'	'D'	'P'	'U'							
[1, 2, 3, 4]	'U'	'D'	'D'	'D'	'P'	'P'	'U'							
[1, 2, 3]	'U'	'D'	'D'	'P'	'P'	'P'	'U'							
[1, 2]	'U'	'D'	'P'	'P'	'P'	'P'	'U'							
[1, 6]	'U'	'D'	'P'	'P'	'P'	'P'	'D'	None	None	1	2	3	4	1
[1, 6]	'U'	'D'	'P'	'P'	'P'	'P'	'P'							
[1]	'U'	'P'	'P'	'P'	'P'	'P'	'P'							

Note: In this trace, the state and parent arrays are printed before entering the for-loop.
Additionally, the state array is printed right before (a call to) the DFS function returns.

5.4 Properties and applications of BFS and DFS

Prior to looking at some of the applications of BFS and DFS note the following:

- BFS and DFS can operate on both directed and undirected graphs.
- BFS and DFS can be used on weighted graphs. They simply ignore the weight information.
- DFS can be implemented by replacing the queue in BFS with a stack and replacing the Enqueue and Dequeue operations with Push and Pop respectively.
- In an undirected graph, after a call to BFS-Tree or DFS-Tree is finished, all the vertices that are in the same component as the starting vertex are marked as processed. In a directed graph, after the call is finished, all the vertices reachable from the starting vertex are marked as processed.
- A call to BFS-Tree or DFS-Tree creates a single tree represented by the parent array. Even though the parent array can represent any forest, a single call to BFS-Tree or DFS-Tree only creates a single tree.
- Traversal can be performed in an order that is neither depth-first nor breadth-first. The reason BFS and DFS are predominant traversal algorithms is that they have properties that are useful for various applications.
- In BFS, vertices are discovered and processed in order of increasing distance from the starting vertex.
- In DFS, a discovered vertex is marked as processed only after all of its undiscovered adjacent vertices are discovered and processed. In other words, a vertex remains in the discovered state until all the vertices reachable from that vertex are discovered and processed.

5.4.1 Shortest paths

In BFS, vertices are discovered in order of increasing distance from the starting vertex. Therefore, if the root of the predecessor tree is vertex s, and a node in the tree is vertex t, then the unique path from s to t in the tree, is a path with the smallest number of edges from s to t in the graph.

The following algorithm takes a predecessor tree and constructs a path from s to t. Note that, this is applicable when the tree is generated by starting a BFS at s.

procedure TREE-PATH($parent, s, t$)
  if $s = t$
    return the single-vertex path s
  else
    return the path TREE-PATH($parent, s, parent[t]$) followed by t

Example 5.2. Consider the following parent array which represents a predecessor tree constructed by doing a BFS starting at vertex 1 of a graph.

```
[None, None, 1, 2, 5, 1, 1 ]
```

The shortest path from 1 to 4 is:

TREE-PATH($parent, 1, 4$) =
TREE-PATH($parent, 1, 5$) followed by 4 =
TREE-PATH($parent, 1, 1$) followed by 5 followed by 4 =
1 followed by 5 followed by 4.

5.4.2 Connected components

Recall that components (aka connected components) of an undirected graph are maximal subgraphs in which all vertices are reachable from each other.

When BFS or DFS is executed from a starting vertex s, all vertices reachable from s will be discovered and processed. Thus the set of vertices that have been processed forms a component. This process can be repeated until all the vertices in the graph are processed (and consequently all the components are identified). This leads to the following algorithm which uses BFS to find the vertices of the components of an undirected graph.

```
procedure CONNECTED-COMPONENTS(Adj)
  n ← number of vertices (the length of Adj)
  state ← an array of length n filled with UNDISCOVERED
  Q ← an empty queue
  components ← {}
  for i from 0 to n − 1
    if state[i] = UNDISCOVERED
      previous-state ← state
      state[i] ← DISCOVERED
      ENQUEUE(Q, i)
      BFS-LOOP(Adj, Q, state)
      new-component ← all vertices whose state has changed
      components ← components ∪ {new-component}
  return components
```

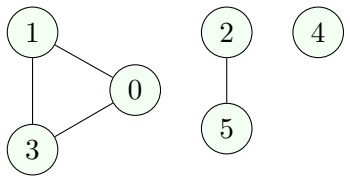
Comments:

- The order of the 'for' loop does not change the functionality of the algorithm but may lead to a different order of discovering components.
- The time complexity of this algorithm is $\Theta(n^2)$ because of the 'for' loop and the line that finds vertices whose state has changed.
- This algorithm can be improved to have the same time complexity as the graph traversal. One way of achieving this is to have an array that maintains a component number for each vertex. The component number can start from 1, for example, and it is incremented inside the loop every time an undiscovered vertex is encountered. The component number is then passed onto BFS which assign it to vertices as they are processed.

Example 5.3. Consider the following undirected graph.

```
U 6
3 0
2 5
1 0
1 3
```

The drawing of the graph is as follows.



The following is the trace of Connected-Components on this graph. The last two columns show the values of *state* and *components* after the iteration.

iteration	call	*state*	*components*
(initialisation)		`[U, U, U, U, U, U]`	{}
0	BFS-Loop(0)	`[P, P, U, P, U, U]`	{{0,1,3}}
1		`[P, P, U, P, U, U]`	{{0,1,3}}
2	BFS-Loop(2)	`[P, P, P, P, U, P]`	{{0,1,3},{2,5}}
3		`[P, P, P, P, U, P]`	{{0,1,3},{2,5}}
4	BFS-Loop(4)	`[P, P, P, P, P, P]`	{{0,1,3},{2,5},{4}}
5		`[P, P, P, P, P, P]`	{{0,1,3},{2,5},{4}}