Chapter 2

Notation and Terminology

In this Chapter, we introduce our terminology for graphs, dynamic graphs, and their analysis. First, we define our notation for directed, undirected, and weighted graphs as well as their components in Section 2.1. We introduce subgraphs, paths, as well as connectivity and present trees as a specific type of graph. Then, we discuss different graph representations and define their notation in Section 2.2. Here, we describe adjacency and incidence lists as well as adjacency matrices. Furthermore, we discuss the storage of graphs in memory using these representations. We introduce the notion of graph properties in Section 2.3. Here, we define four different types of properties, namely vertex values, vertex pair values, graph values, and graph distributions. Then, we describe the concept of metrics that group together multiple properties. In Section 2.4, we introduce the notion of dynamic graphs. Here, we define atomic and compound updates that change a dynamic graph over time as well batches that group together consecutive updates. Then, we describe the transition of a dynamic graph and the resulting states. We close this Section with an example of a dynamic graph and its transitions. In Section 2.5, we discuss the problem of dynamic graph analysis. We introduce the concepts of algorithms for computing the properties of a graph and describe three classes of such algorithms: snapshot-, stream-, and batch-based approaches. We end this Section with an input-based classification of algorithms for the analysis of dynamic graphs.

2.1 Graphs

In this Section, we start by introducing the basic characteristics of graphs. First, we describe directed and undirected graphs, the two main types of graphs considered in this thesis, in Section 2.1.1. Then, we outline the concept of weighted graphs in Section 2.1.2. We define types of adjacency and incidence lists in Section 2.1.3 and subgraphs in Section 2.1.4. We describe the concept of paths in Section 2.1.5 and the connectivity of a graph in Section 2.1.6. In Section 2.1.7, we introduce trees, forest, and spanning trees as specific types of graphs.

2.1.1 Directed and Undirected Graphs

A *graph* $G = (V, E)$ is an ordered pair of *vertices* and *edges*. Vertices $V(G) \coloneqq V = \{v_1, v_2, \ldots, v_{|V|}\}$ are the elements of a graph G and are also referred to as *nodes* or *dots*. We refer to the number of vertices $|V|$ as the *size of graph* G. The edges $E(G) \coloneqq E = \{e_1, e_2, \ldots, e_{|E|}\}$ of a graph G describe relations between its vertices, also referred to as *connections* or *arcs*.

Commonly, an edge e *connects* two vertices $v, w \in V$, also referred to as e's *endpoints*. An edge that connects a vertex v to itself is called a *loop*. Edges that connect an arbitrary number of vertices, possibly more than two, are called *hyperedges*. Two or more edges that have the same endpoints are called *multiple* or *parallel* and *simple* otherwise. In this thesis, we only consider graphs that contain simple edges and no loops as they can be used to model hypergraphs as well as multigraphs. We call a graph G' *more densely connected* or *denser* than G if $\frac{|E'|}{|V'|} > \frac{|E|}{|V|}$.

Graphs and their edges can be either *directed* or *undirected*. A *directed edge* is an *ordered pair* of vertices $(v, w) \in \{(v, w) : v, w \in V, v \neq w\}$ with dedicated *source* v and *destination* w (cf. the example in Figure 2.1a). We call $e^{-1} \coloneqq (w, v)$ the *inverse edge* of $e = (v, w)$ and refer to v and w as *connected bidirectionally* if $e, e^{-1} \in E$. An *undirected edge* is an *unordered pair* of vertices $\{v, w\} \in \{\{v, w\} : v, w \in V, v \neq w\}$ (cf. the example in Figure 2.1b). In contrast to directed edges, there exists no particular relation between the endpoints of an undirected edge. Note that an undirected graph $G^u = (V, E^u)$ can

(a) G^d (directed) (b) G^u (undirected)

Figure 2.1: Examples of graphs with 5 vertices

be expressed as a directed graph $G^d = (V, E^d)$ by defining all connections bidirectionally, i.e.:

$$E^d \coloneqq \bigcup_{\{v,w\} \in E^u} \{(v, w), (w, v)\} \tag{2.1}$$

Throughout this thesis, we simply refer to edges as $e = (v, w)$ if we do not specify if the graph is directed or undirected.

2.1.2 Weighted Graphs

Weights or *attributes* represent additional information assigned to vertices or edges from some domain $\mathcal{W}$. Commonly, $\mathcal{W}$ is a set of numbers, e.g., $\mathbb{N}$, $\mathbb{R}$, $[0, 1]$. In this thesis, we allow $\mathcal{W}$ to be an arbitrary set, including:

- sets of vectors, e.g., $[-5, 100]^2 \subseteq \mathbb{N}^2, [0, 1]^3 \subseteq \mathbb{R}^3$,
- sets of labels, e.g., {customer, product, action},
- sets of colors, e.g., {red, green, blue, yellow}, or
- any combination of the above, e.g., $\mathbb{N} \times [0, 1] \times$ {label1, label2}.

This kind of information is commonly referred to as attributes rather than weights. In the following, we refer to any $\mathcal{W}$ as a set of weights.

Weight functions w^V and w^E represent the assignments of vertices and edges to their respective weights in the *vertex weight domain* $\mathcal{W}^V$ and the *edge weight domain* $\mathcal{W}^E$:

$$w^V : V \to \mathcal{W}^V \qquad w^E : E \to \mathcal{W}^E.$$

Weighted graphs are then represented as triples or quadruples:

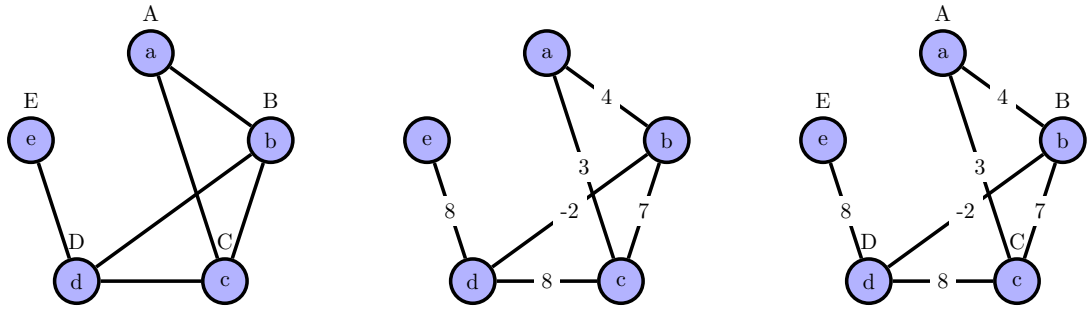
- weighted vertices: $G = (V, E, w^V)$ (cf. example in Figure 2.2a),
- weighted edges: $G = (V, E, w^E)$ (cf. example in Figure 2.2b), or
- weighted vertices and edges: $G = (V, E, w^V, w^E)$ (cf. example in Figure 2.2c).

We write $\mathcal{W}$ and w instead of $\mathcal{W}^V$, $\mathcal{W}^E$, w^V, and w^E whenever the domain is clear from the context.

2.1.3 Incidence and Adjacency Lists

An edge is called *incident to* the two vertices it connects. Two vertices connected by an edge are called *adjacent to* each other or *neighbors*. From this general notion of *incidence* and *adjacency*, we define the following *incidence* and *adjacency lists* for a vertex v in a directed graph $G^d = (V, E^d)$:

- incoming incident edges: $inc^{in}(v) \coloneqq \{(w, v) \in E^d\}$,
- outgoing incident edges: $inc^{out}(v) \coloneqq \{(v, w) \in E^d\}$,
- incident edges: $inc(v) \coloneqq inc^{in}(v) \cup inc^{out}(v)$,
- incoming adjacent vertices: $adj^{in}(v) \coloneqq \{w \in V : (w, v) \in E^d\}$,



(a) $G = (V, E, w^V)$ (b) $G = (V, E, w^E)$ (c) $G = (V, E, w^V, w^E)$

Figure 2.2: Examples of undirected weighted graphs

- outgoing adjacent vertices: $adj^{out}(v) \coloneqq \{w \in V : (v, w) \in E^d\}$, and
- adjacent vertices: $adj(v) \coloneqq adj^{in}(v) \cup adj^{out}(v)$.

Similarly, we define the following incidence and adjacency lists for a vertex v in an undirected graph $G^u = (V, E^u)$:

- incident edges: $inc(v) \coloneqq \{\{v, w\} \in E^u\}$ and
- adjacent vertices: $adj(v) \coloneqq \{w : \{v, w\} \in E^u\}$.

The *degree* $d(v)$ of a vertex v expresses the number of edges incident to it. For directed graphs, we also define the *incoming* and *outgoing degree*, also referred to as *in-* and *out-degree*:

- degree: $d(v) \coloneqq |inc(v)|$,
- in-degree: $d^{in}(v) \coloneqq |inc^{in}(v)|$, and
- out-degree: $d^{out}(v) \coloneqq |inc^{out}(v)|$.

We refer to vertices w that are adjacent to v as its *neighbors* and to v's adjacency list $adj(v)$ as *neighborhood*. Analogously, we refer to adj^{in} and adj^{out} as *incoming* and *outgoing neighborhoods* and to their elements as *incoming* and *outgoing neighbors*. For a set of vertices $V' \subseteq V$, we define the *neighborhood of* V' as the union of the neighborhoods of all its elements. We denote this as $N(V') \coloneqq \bigcup_{v \in V'} adj(v)$.

As the *k-neighborhood* $N_k(e)$ of an edge $e = \{v, w\}$, we denote the set of all $(k-2)$-tuples N of vertices $u \in V,\ u \neq v, w$ that are connected to vertices v or w in the induced subgraph $G[N \cup \{v, w\}]$. We call each element $N \in N_k(e)$ a *k-neighbor set* of e, which contains $|N| = k - 2$ elements.

Furthermore, we denote the *extended k-neighborhood* $N_k^+(e)$ as the set of all k-neighbor sets united with $\{v, w\}$, i.e., $N_k^+(\{v, w\}) \coloneqq \{N \cup \{v, w\}, N \in N_k(\{v, w\})\}$. Each of these *extended k-neighbor sets* corresponds to the connected subgraph $G[N \cup \{v, w\}]$ of G that contains v and w as well as $k - 2$ other vertices

2.1.4 Subgraphs

A graph $G' = (V', E')$ is called a *subgraph* of the graph $G = (V, E)$, denoted as $G' \subseteq G$, iff:

$$V' \subseteq V \ \wedge \ E' \subseteq E.$$

We call $G' = (V', E')$ the V'*-induced* subgraph of $G = (V, E)$, denoted as $G' = G[V']$, iff:

$$V' \subseteq V \ \wedge \ E' = \{(v, w) \in E : v, w \in V'\}.$$

We call $G' = (V', E')$ the E'*-induced* subgraph of $G = (V, E)$, denoted as $G' = G[E']$, iff:

$$E' \subseteq E \ \wedge \ V' = \{v, w \in V : (v, w) \in E'\}.$$

2.1.5 Paths

A *path* is a sequence of edges $p = (e_1, e_2, ...), e_i \in E$ such that $e_i = (u, v)$, $e_{i+1} = (v, w)$, and $i \neq j \implies e_i \neq e_j$. For any path $p = ((v_0, v_1), \dots, (v_{|p|-1}, v_{|p|}))$, we refer to v_0 as the start and to $v_{|p|}$ as the end vertex of p and refer to p as a *path from* v_0 *to* $v_{|p|}$. The *length* of a path is the number of edges it contains, i.e., $|p|$. In this thesis, we only consider paths that have no loops, i.e., no vertex appears more than once in a path.

The *shortest path length* $spl(v, w)$ from vertex v to w is the shortest length of all existing paths from v to w, i.e., paths of the form $((v, x), \dots (y, w))$. If there exists *no path* from v to w, we define the shortest path length to be ∞.

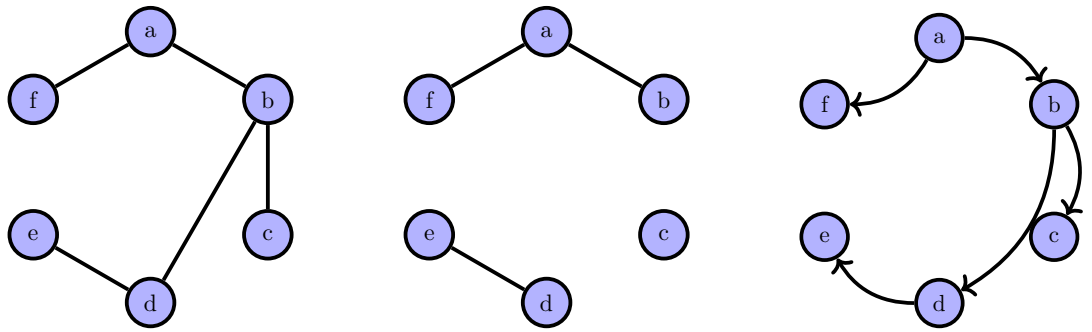
2.1.6 Connectivity

An undirected graph is called *connected* if there exists a path between any pair of vertices, i.e., $\forall\, v, w \in V : spl(v, w) \neq \infty$. Otherwise, it is called *disconnected*. If the same holds for a directed graph, we call it *strongly connected*. We call a directed graph *weakly connected* if it is strongly connected after adding all inverse edges to it.

A subset of vertices $V' \subseteq V$ is called a *(strongly / weakly) connected component of* G if the induced subgraph $G[V']$ is (strongly / weakly) connected but any proper superset $V'' \supset V', V'' \subseteq V$ is not. We refer to the unique (strongly / weakly) connected component V' that contains a vertex v as the *(strong / weak) component of* v. We denote the *set of all (strongly / weakly) connected components of a graph* G as $\mathcal{C}_G = (C_1, C_2, \dots)$. Note that $\mathcal{C}_G$ is a partition of the vertex set $V(G)$.

2.1.7 Trees, Forests, and Spanning Trees

A *tree* is a connected, undirected graph without loops. Hence, there exists exactly one path between any two vertices and $|E| = |V| - 1$ (cf. the example in Figure 2.3a). A *forest* is an undirected graph whose connected components are trees (cf. the example in Figure 2.3b).



(a) Tree (b) Forest with 3 connected components (c) Rooted tree with root a

Figure 2.3: Examples of tree graphs

A *rooted tree* is a weakly connected, directed graph for which $|E| = |V| - 1$ and that contains a dedicated root vertex $r \in V$ such that $\forall\, v \in V : spl(r, v) \neq \infty$. It is often referred to as an *out-tree* because the root has only outgoing edges and all others point away from the root (cf. the example in Figure 2.3c). While the root has no incoming edges, every other vertex v has exactly one incoming edge (w, v). Then, w is referred to as the *predecessor* or *parent* of v. All vertices u incident to v via outgoing edges (v, u) are called *successors* or *children* of v. Vertices v that have no children, i.e., $d^{out}(v) = 0$, are called *leafs* of the tree. In the example shown in Figure 2.3c, the root a has the two children f and b which share a as their parent. In total, the tree has three leafs: c, e, and f.

For a connected, undirected graph G, we refer to any subgraph $G[E'], E' \subseteq E$ that contains all vertices and is a tree as a *spanning tree of graph* G. Note that such a spanning tree does not exist if the graph is disconnected, i.e., it consists of more than a single connected component. For a directed graph G and a vertex $v \in V$, we refer to any subgraph which is a tree rooted at v and has maximum size as a *spanning tree of vertex* v. Such a spanning tree of v contains all vertices w for which $spl(v, w) \neq \infty$.

2.2 Graph Representations

In this Section, we introduce approaches for the representation of graphs and how they are stored in memory. First, we describe the representation of graphs using adjacency and incidence lists for each vertex in Section 2.2.1. Then, we introduce adjacency matrices as another way to represent the connections between vertices in Section 2.2.2. Finally, we discuss the storage of graphs in memory using these two representations in Section 2.2.3.

2.2.1 Incidence and Adjacency Lists

Commonly, the lists of all vertices V and edges E are used to represent all elements of a graph G. In addition, incidence and adjacency lists are used for immediate access to the connections of each vertex $v \in V$ (cf. Section 2.1.3).

Which lists actually need to be represented in memory depends on the application. In case an analysis only requires access to the adjacent vertices, the representation of $adj(v)$ in memory can be sufficient and might be complemented by $adj^{in}(v)$ or $adj^{out}(v)$. If an analysis also requires information about incident edges of vertices, their connections must be represented using $inc(v)$, $inc^{in}(v)$, or $inc^{out}(v)$. An example is the computation of shortest paths that take edge weights into account. Here, edges are needed to obtain the respective weight from the edge weight function w^E.

Note that incidence lists carry more information than adjacency lists. All adjacent vertices are represented as part of the incident edges which can in addition carry weights. Hence, the use of incidence lists covers all scenarios that adjacency lists would and could be seen as the more general representation.

2.2.2 Adjacency Matrices

As an *adjacency matrix* $A(G)$, we consider a $|V| \times |V|$ matrix denoting the adjacencies of all vertex pairs in the graph G. We always write A instead of $A(G)$ if the graph G is clear from the context.

The adjacency matrix A^d of a directed graph is defined as follows:

$$A^d_{ij} \coloneqq \begin{cases} - & \text{if } i = j \\ 1 \text{ (true)} & \text{if } (v_i, v_j) \in E \\ 0 \text{ (false)} & \text{if } (v_i, v_j) \notin E \end{cases}$$

Analogously, the adjacency matrix A^u of an undirected graph is defined as follows:

$$A^u_{ij} \coloneqq \begin{cases} - & \text{if } i = j \\ 1 \text{ (true)} & \text{if } \{v_i, v_j\} \in E \\ 0 \text{ (false)} & \text{if } \{v_i, v_j\} \notin E \end{cases}$$

Note that the adjacency matrix of an undirected graph is symmetric. Therefore, we only print the upper triangle of an undirected graph's adjacency matrix to clearly differentiate them from directed graphs as shown by the examples in Tables 2.1a and 2.1b

	a	b	c	d	e
a	-	1	1	0	0
b	0	-	1	0	0
c	1	0	-	1	0
d	0	1	0	-	1
e	0	0	0	0	-

(a) $A(G^d)$ (directed)

	a	b	c	d	e
a	-	1	1	0	0
b		-	1	1	0
c			-	1	0
d				-	1
e					-

(b) $A(G^u)$ (undirected)

Table 2.1: Examples of directed and undirected adjacency matrices

With $concat(A^d)$, we denote the row-by-row *concatenation* of all boolean values of the adjacency matrix A^d of a directed graph, i.e., $concat(A^d) = (a_1, a_2, \dots a_{k\cdot(k-1)}) \coloneqq (A_{1,2}, A_{1,3}, \dots A_{k,k-1})$. For undirected graphs, $concat(A^u)$ denotes the tow-by-row concatenation of all values above the diagonal, i.e., $concat(A^u) = (a_1, a_2, \dots a_{\frac{k\cdot(k-1)}{2}}) \coloneqq (A_{1,2}, A_{1,3}, \dots A_{k-1,k})$. For example, the concatenations of the adjacency matrices given in Tables 2.1a and 2.1b are $concat(A(G^d)) = 11000100101001010000$ and $concat(A(G^u)) = 1100110101$.

We define the *key* of an adjacency matrix A as the numerical value of the binary interpretation of its concatenation, i.e., $key(A) \coloneqq concat(A)_2 \in \mathbb{N}$. Analogously, we define the key of a graph

as $key(G) \coloneqq key(A(G))$. The keys of the examples given in Tables 2.1a and 2.1b are $key(A^d) = 11000100101001010000_2 = 805{,}456$ and $key(A^u) = 1100110101_2 = 821$.

We denote the *set of all undirected adjacency matrices of size k* as $\mathcal{A}_k$, $|\mathcal{A}_k| = 2^{\frac{k\cdot(k-1)}{2}}$. The set of all adjacency matrices of *connected undirected graphs* of size k is denoted as $\mathcal{A}_k^{con} \subset \mathcal{A}_k,\ k \geq 2$. We denote the set of all keys of undirected adjacency matrices of size k as $\mathcal{N}_k = [0, 2^{\frac{k\cdot(k-1)}{2}} - 1]$. Furthermore, we denote the set of all keys of connected undirected graphs of size k as $\mathcal{N}_k^{con} \subset \mathcal{N}_k$.

As an example, consider $\mathcal{A}_3 = \{A_0, A_1, \dots A_7\}$, the set of all eight adjacency matrices of undirected 3-vertex graphs, shown in Figure 2.4. Their indexes relate to their respective key. A 3-vertex graph is connected if it contains at least two edges. Hence, there are four adjacency matrices of connected undirected 3-vertex graphs, i.e., $\mathcal{A}_3^{con} = \{A_3, A_5, A_6, A_7\}$ and $\mathcal{N}_k^{con} = \{3, 5, 6, 7\}$.

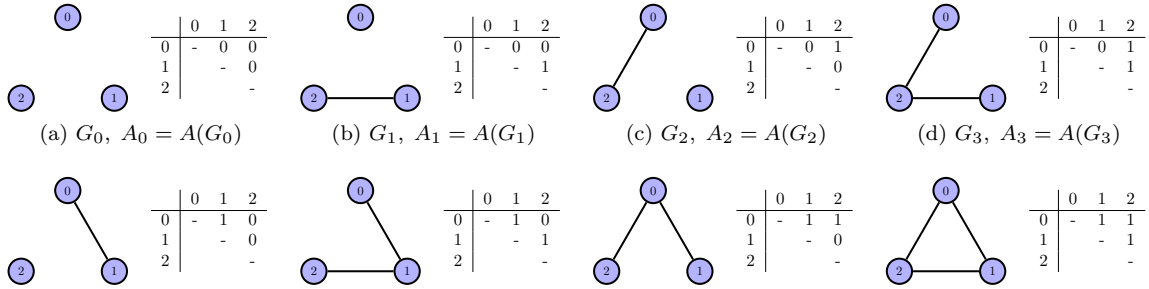


Figure 2.4: $\mathcal{A}_3$ - all adjacency matrices of undirected 3-vertex graphs

2.2.3 Storing a Graph in Memory

The representation of a graph as an adjacency matrix is beneficial when computing measures like the eigenvector centrality [53, 289], a graph property commonly used in economics and social sciences [236, 150]. Also, testing the existence of an edge is possible in $O(1)$ time as it only implies the lookup of a bit at a deterministic location in memory. But adjacency matrices are not well-suited for the execution of operations required for the computation of many other graph measures [139, 108]. The computation of basic measures like, e.g., connected components and all-pairs shortest paths, requires the iteration over incidence or adjacency lists which can be done in $O(|d(v)|)$ time using adjacency lists but requires $O(|V|)$ time on adjacency matrices. Often, algorithms iterate over all edges $e \in E$ which requires $O(|V|^2)$ time on an adjacency matrix. Furthermore, the memory footprint of storing a graph as adjacency matrix is $O(|V|^2)$ and thereby higher for large graphs than $O(|V|+|E|)$, the memory footprint of storing adjacency lists.

Therefore, graphs are commonly represented using adjacency or incidence lists [139, 108]. Hence, we must maintain at least two lists: V and E. For directed graphs, up to six different lists can be maintained for each vertex: $inc^{in}(v)$, $inc^{out}(v)$, $inc(v)$, adj^{in}, adj^{out}, and adj. For undirected graphs, we can maintain up to two lists per vertex: $inc(v)$ and $adj(v)$. We denote the set of all potential lists for the representation of directed and undirected graphs in memory as follows:

$$\begin{aligned} \mathcal{L}^d &\coloneqq \{V, E, inc^{in}, inc^{out}, inc, adj^{in}, adj^{out}, adj\} \\ \mathcal{L}^u &\coloneqq \{V, E, inc, adj\} \end{aligned}$$

We simply write $\mathcal{L}$ instead of $\mathcal{L}^d$ or $\mathcal{L}^u$ if we do not need to differentiate. Each list stores either vertices or edges, referred to as its *element type* t_v, or t_e. We denote these two types as $\mathcal{T}$ and refer to the type stored in a specific list by the function

$$t : \mathcal{L} \to \mathcal{T} \coloneqq \{t_v, t_e\}$$

with

$$\begin{aligned} t(V) = t(adj^{in}) = t(adj^{out}) = t(adj) &\coloneqq t_v \\ t(E) = t(inc^{in}) = t(inc^{out}) = t(inc) &\coloneqq t_e. \end{aligned}$$

In our notation, vertex and edge weights are represented as separate mappings, i.e., $w^V(v)$ and $w^E(e)$ (cf. Section 2.1.2). Often, weighted vertices and edges are considered as pairs instead: $(v, w^V(v))$ and $(e, w^E(e))$. Hence, weights can be represented by adding separate weight mappings or appending the respective weight to vertices or edges.

2.3 Graph Properties

In this Section, we introduce different types of graph properties and their grouping as metrics. First, we describe the general notion of properties in Section 2.3.1. Then, we define four types of properties that cover all properties commonly considered in graph analysis: vertex values, vertex pair values, graph values, and graph distributions. Then, we introduce the notion of metrics that group together sets of graph properties in Section 2.3.2.

2.3.1 Properties

We denote a *property* as a function $p : dom \to cod$ that maps each element from the domain *dom* to an element in the codomain *cod*. In this thesis, we consider properties with domain $\mathcal{G}$, V, and $V \times V$. Here, $\mathcal{G}$ denotes the set of all graphs, V is the set of vertices of a graph, and $V \times V$ is the set of all their pairs. As codomain, we consider the set of real numbers $\mathbb{R}$ as well as the set of all frequency distributions, denoted as $\mathcal{F}$. We denote the set of all properties as $\mathcal{P}$.

As *vertex values*, we consider properties that define a single value $r \in \mathbb{R}$ as property of any vertex $v \in V$. Hence, a vertex value is a function $p : V \to \mathbb{R}$, i.e., $p(v)$ is the property of v. With $\mathcal{P}_{V \to \mathbb{R}} \subset \mathcal{P}$, we denote the set of all vertex values. Examples of vertex values are the degree of a vertex, the average shortest path length from a vertex to any other, and the sum of edge weights assigned to all outgoing edges of a vertex.

As *vertex pair values*, we denote properties that define a single value $r \in \mathbb{R}$ for each pair of vertices $(v, w) \in V \times V$. Therefore, a vertex pair value is a function $p : V \times T \to \mathbb{R}$, i.e., $p(v, w)$ is the property of v and w. We denote the set of all vertex pair values as $\mathcal{P}_{V \times V \to \mathbb{R}} \subset \mathcal{P}$. Examples of vertex pair values for v and w are the Jaccard index of v's and w's neighborhoods, the shortest path length from v to w, the number of disjoint paths between v and w, and the Sørensen-Dice coefficient of their adjacency lists.

As a *graph value*, we consider a property that defines a single value $r \in \mathbb{R}$ for a whole graph. Hence, graph values are defined as function $p : \mathcal{G} \to \mathbb{R}$, i.e., $p(G)$ is the property of G. Here and in the following, $\mathcal{G}$ denotes the set of all graphs. We denote the set of all graph values as $\mathcal{P}_{\mathcal{G} \to \mathbb{R}} \subset \mathcal{P}$ Examples of graph values are the average degree of all vertices, the diameter, the assortativity coefficient, and the global clustering coefficient.

A *frequency distribution* is a function $F : \mathbb{N} \to \mathbb{N}, F(k) \coloneqq$ occurrences of value k for property p. In case the property p is not expressed as a natural number, we transform each value k to $k' \coloneqq \lceil \frac{k}{b} \rceil$, where b is the so-called *bin-size*. We denote such a *binned frequency distribution* as F^b and note that we can always write $F(p)$ as $F^1(p)$. With $\mathcal{F}^b$, we denote the *set of all (binned) distributions*. In the following, we simply write F and $\mathcal{F}$ instead of F^b and $\mathcal{F}^b$ when the bin-size is clear from the context. As a *graph distribution*, we consider a property p of a whole graph that is expressed as a frequency distribution, i.e., $p : \mathcal{G} \to \mathcal{F}$, i.e., $p(G)$ is the property of G. We denote the set of all graph distributions as $\mathcal{P}_{\mathcal{G} \to \mathcal{F}} \subset \mathcal{P}$. Examples are the degree distribution, the distribution of all-pairs shortest path lengths, the distribution of vertex weights, or the distribution of local clustering coefficients.

2.3.2 Metrics

As a *metric*, we consider a set of properties that can be computed in conjunction. We denote the *set of all metrics* as $\mathcal{M}$. We define the set of all its vertex values as $P_{V \to \mathbb{R}}(m) \subseteq \mathcal{P}_{V \to \mathbb{R}}$, the set of all its vertex pair values as $P_{V \times V \to \mathbb{R}}(m) \subseteq \mathcal{P}_{V \times V \to \mathbb{R}}$, the set of all its graph values as $P_{\mathcal{G} \to \mathbb{R}}(m) \subseteq \mathcal{P}_{\mathcal{G} \to \mathbb{R}}$, and the set of all its graph distributions as $P_{\mathcal{G} \to \mathcal{F}}(m) \subseteq \mathcal{P}_{\mathcal{G} \to \mathcal{F}}$. Furthermore, we define the set of all properties of a metric as $P(m) \coloneqq P_{V \to \mathbb{R}}(m) \cup P_{V \times V \to \mathbb{R}}(m) \cup P_{\mathcal{G} \to \mathbb{R}}(m) \cup P_{\mathcal{G} \to \mathcal{F}}(m)$.

As a first example, consider the *degree* metric. It contains the degree distribution, which is a graph distribution, and four graph values: the minimum, median, average, and maximum degree. For directed graphs, we could extend the metric to also include the in-degree distribution, the out-degree distribution, and the corresponding graph values. Note that we can consider the degree distribution as a distribution over node values and directly compute the graph values from the respective distribution.

As a second example, consider the *clustering* metric. It contains the local clustering coefficients, a vertex value property, a binned distribution over those, and two graph values: the global and the average clustering coefficient.

As a third example, consider the *all-pairs shortest paths* metric. It contains the graph distribution of the shortest paths between any two vertices that could also be considered as vertex pair values. In addition it contains the characteristic path length and the diameter. Both properties can again be computed directly from the distribution.

2.4 Dynamic Graphs

In this Section, we introduce the notion of dynamic graphs, how their changes are modeled, how their transitions are represented, and how they can be related to real-world dynamic graphs. First, we introduce atomic updates that model the changes occurring to a dynamic graph over time in Section 2.4.1. Based on that, we describe compound updates in Section 2.4.2 and introduce batches that group together consecutive updates in Section 2.4.3. Then, we describe two approaches to model the transitions of a dynamic graph over time in Section 2.4.4: the stream-based and the batch-based transition. In Section 2.4.5, we introduce the relation between the states of a real-world dynamic graph and a model of it. Finally, we present an example of a dynamic graph and use it to describe the different transitions and states in Section 2.4.6.

2.4.1 Atomic Updates

There exist four atomic operations that change the topology of a given graph: adding a new vertex (without edges), removing an existing vertex (that has no edges), adding a new edge, and removing an existing edge. We refer to them as *topology updates*. In case of weighted graphs, two additional atomic operations exist: changing the weight of a vertex and changing the weight of an edge. We refer to them as *weight updates*.

Update	**Argument Scope**	**Application**
$add^V(v)$	$v \notin V$	$V' \coloneqq V \cup \{v\}$
$rem^V(v)$	$v \in V, d(v) = 0$	$V' \coloneqq V \setminus \{v\}$
$add^E(e)$	$e \notin E$	$E' \coloneqq E \cup \{e\}$
$rem^E(e)$	$e \in E$	$E' \coloneqq E \setminus \{e\}$
$wgt^V(v, w)$	$v \in V, w \in \mathcal{W}^V, w \neq w^V(v)$	$w'^V(v) = \begin{cases} w & v = v' \\ w^V(v') & \text{otherwise} \end{cases}$
$wgt^E(e, w)$	$e \in E, w \in \mathcal{W}^E, w \neq w^E(e)$	$w'^E(e') = \begin{cases} w & e = e' \\ w^V(e') & \text{otherwise} \end{cases}$

Table 2.2: Notation and argument scope of topology and weight updates

A list of these six atomic update operations is given in Table 2.2. Each *topology update* takes as argument the vertex or edge that should be added to or removed from the graph. *Weight updates* take as argument a vertex or edge and the new weight.

When applying an update u to a graph G, the graph is transformed into G'. We denote this transformation as $G \xrightarrow{u} G'$. For each type of update, the result of its application is shown in Table 2.2.

2.4.2 Compound Updates

In addition to the six atomic updates, it is common that a vertex is removed at the same time as all its edges. Hence, we define a *compound vertex removal* update $rem^V(v) : v \in V, d(v) \geq 0$ that removes the vertex v ($V' \coloneqq V \setminus \{v\}$) and all edges incident to it ($E' \coloneqq E \setminus inc(v)$). This can be considered as the simultaneous application of $rem^E(e), e \in inc(v))$ and $rem^V(v)$ (cf. Table 2.3).

Update	**Argument Scope**	**Application**
$rem^V(v)$	$v \in V, d(v) \geq 0$	$rem^E(e) : e \in inc(v), rem^V(v)$
$add^V(v, w)$	$v \notin V$	$add^V(v), wgt^V(v, w)$
$add^E(e, w)$	$e \notin E$	$add^E(e), wgt^E(e, w)$

Table 2.3: Notation and argument scope for compound updates

For weighted graphs, new vertices and edges are added with an initial weight. We account for this by defining *compound vertex addition* and *compound edge addition* updates $add^V(v, w)$ and $add^E(e, w)$. Their application adds the respective element ($add^V(v)$ or $add^E(e)$) and defines its weight ($wgt^V(v, w)$ or $wgt^E(e, w)$) as described in Table 2.3. When discussing weighted graphs, we write $add^V(v)$ and $add^E(e)$ to indicate the addition of a vertex or edge with some default weight w_V or w_E, i.e., $add^V(v, w_V)$ or $add^E(e, w_E)$.

2.4.3 Batches of Updates

As a *batch* of updates B, we consider a set of updates $B = \{u, u', u'', \dots\}$. We denote the subsets of vertex additions, removals, and weight changes as $V^+(B)$, $V^-(B)$, and $V^w(B)$. Analogously, we denote the subsets of edge additions, removals, and weight changes as $E^+(B)$, $E^-(B)$, and $E^w(B)$. Then, a batch can also be described as $B = V^+(B) \cup V^-(B) \cup V^w(B) \cup E^+(B) \cup E^-(B) \cup E^w(B)$. As the *size of a batch* B, we consider the number of updates contained therein, i.e., $|B|$.

The *application of a batch* B to a graph G means to apply each update $u \in B$. We assume, that they are applied in the following order:

$$E^-(B),\ V^-(B),\ V^+(B),\ E^+(B),\ V^w(B),\ E^w(B).$$

The application of a batch B transforms the set of vertices as follows:

$$V' \coloneqq (V \setminus \{v : rem^V(v) \in B\}) \cup \{v : add^V(v) \in B\}.$$

Similarly, the set of edges is transformed to:

$$E' \coloneqq (E \setminus \{e : rem^E(e) \in B\}) \cup \{e : add^E(e) \in B\}.$$

The application of the vertex and edge weight changes updates the respective weight function accordingly.

2.4.4 Transitions of Dynamic Graphs

The transition of a dynamic graph over time can be described stream- or batch-based. In the *stream-based transition*, we consider the application of each update separately. In the *batch-based transition*, we only consider the states of the dynamic graph before and after the application of complete batches and not in between.

Each change in a dynamic graph can be described as an atomic (or compound) update. Hence, the transition of a dynamic graph over time can be described as a potentially infinite *stream of updates* $(\dots, u_i, u_{i+1}, \dots)$. The application of an update u_{i+1} transforms a dynamic graph G from *state* G_i into the next state G_{i+1}. We denote the transition implied by u_{+1} as $G_i \xrightarrow{u_{i+1}} G_{i+1}$. Then, a *dynamic graph* is described in its entirety with a stream-based transition by an initial state G_0 and the stream of update $(u_1, u_2, u_3, \dots)$:

$$G_0 \xrightarrow{u_1} G_1 \xrightarrow{u_2} G_2 \xrightarrow{u_3} G_3\ \dots$$

Batches group together multiple updates. We summarize all changes required for the transition from state G_i to $G_j, i < j$ as a batch $B_{i,j}$ as follows:

$$\begin{aligned}
V^+(B_{i,j}) &\coloneqq \{add^V(v) : v \notin V_i\ \wedge\ v \in V_j\} \\
V^-(B_{i,j}) &\coloneqq \{rem^V(v) : v \in V_i\ \wedge\ v \notin V_j\} \\
V^w(B_{i,j}) &\coloneqq \{wgt^V(v, w_j^V(v)) : w_i^V(v) \neq w_j^V(v)\} \\
E^+(B_{i,j}) &\coloneqq \{add^E(e) : e \notin E_i\ \wedge\ e \in E_j\} \\
E^-(B_{i,j}) &\coloneqq \{rem^E(e) : e \in E_i\ \wedge\ e \notin E_j\} \\
E^w(B_{i,j}) &\coloneqq \{wgt^E(e, w_j^E(e)) : w_i^E(e) \neq w_j^E(e)\}
\end{aligned}$$

We denote the transition from state G_i to G_j via a batch $B_{i,j}$ as $G_i \xrightarrow{B_{i,j}} G_j$. Then, a *dynamic graph* is described in its entirety with a batch-based transition by an initial state G_0 and a list of batches $(B_{0,i}, B_{i,j}, B_{j,k}, \dots)$:

$$G_0 \xrightarrow{B_{0,i}} G_i \xrightarrow{B_{i,j}} G_j \xrightarrow{B_{j,k}} G_k\ \dots$$

Note that a stream-based transition can be represented as a batch-based transition with batches of size 1, i.e., $B_{i,i+1} = \{u_{i+1}\}$. Furthermore, note that $B_{i,j} \subseteq (u_i, u_{i+1}, \dots, u_{j-1}, u_j)$. As an example, consider the updates $u_{i'} = add^E(e), u_{j'} = rem^E(e), i \leq i' < j' \leq j$. While vertex v is present from state $G_{i'}$ through $G_{j'-1}$, it is neither contained in V_i nor V_j and hence, $u_{i'}, u_{j'} \notin B_{i,j}$. As another example, consider the updates $u_{i'} = wgt^V(v, w_1)$ and $u_{j'} = wgt^V(v, w_2)$. In case no weight change for v occurs afterwards, $u_{j'}$ basically overrides $u_{i'}$ at the analysis-frequency represented by $B_{i,j}$ such that $u_{i'} \notin B_{i,j}$ but $u_{j'} \in B_{i,j}$.

2.4.5 States of Dynamic Graphs

In real-world dynamic graphs, each change happens at a specific *point in time* we refer to as a *timestamp* t. Without loss of generality, we assume that these timestamps represent the Unix timestamp (UTS) at a frequency that fits the nature of the dynamic graph, e.g., in seconds (s), milliseconds (ms = 10^{-3} s) microseconds (μs = 10^{-6} s), nanoseconds (ns = 10^{-9} s), or picoseconds (ps = 10^{-12} s). We denote the Unix timestamp of an update u_i as a function $UTS : \mathbb{N} \to \mathbb{N}, UTS(i) =$ timestamp of u_i.

We refer to the state G_i of a dynamic graph G as the *snapshot of G at index i*. Assume $UTS^{-1}(t)$ to be the index of the dynamic graph G at timestamp t, i.e., $UTS^{-1}(t) \coloneqq \max_{i \in \mathbb{N}} UTS(i) \leq t$. Then, we refer to the state $G_{@t} \coloneqq G_{UTS^{-1}(t)}$ of a dynamic graph G as the the *snapshot of G at timestamp t*. For simplicity, we write G_t instead of $G_{@t}$ if it is clear that t refers to a timestamp.

2.4.6 Example of a Dynamic Graph and its Transitions

As an example, consider the 6 consecutive states $G_0, G_1, \ldots, G_5$ of the dynamic graph G shown in Figure 2.5. Initially, the graph G_0 consists of 5 vertices connected by 6 edges: $V_0 = \{a, b, c, d, e\}$ and $E_0 = \{(a, b), (a, c), (a, d), (c, d), (d, c), (d, e)\}$.

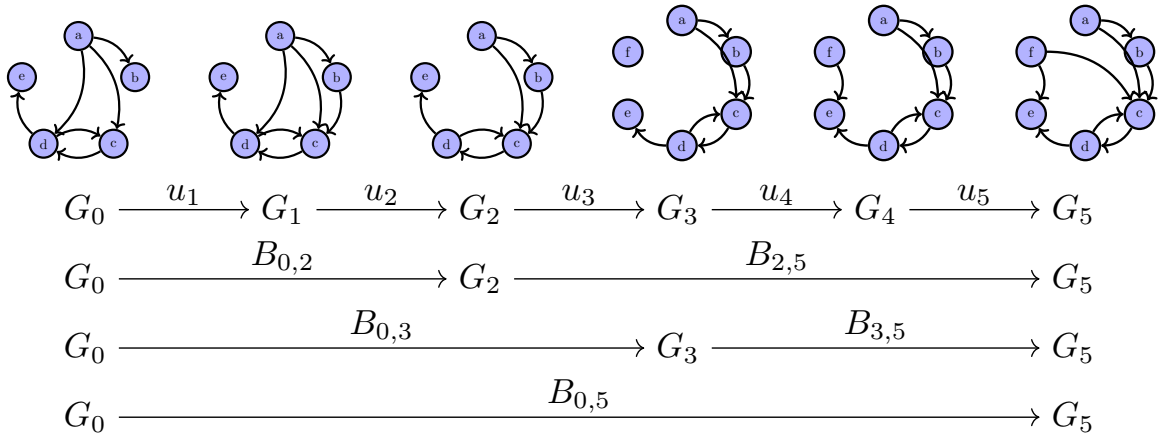


Figure 2.5: Transitions of dynamic graph G from G_0 to G_5

The stream-based transition is described as a list of 5 updates (cf. Figure 2.5): ($u_1 = add^E((b, c))$, $u_2 = rem^E((a, d))$, $u_3 = add^V(f)$, $u_4 = add^E((f, e))$, and $u_5 = add^E((f, c))$. This results in a model where all 6 states are represented.

In addition, three different batch-based transitions are presented in the example: $(B_{0,2}, B_{2,5})$, $(B_{0,3}, B_{3,5})$, and $(B_{0,5})$. The number of states modeled by them depends on the partitioning of updates into batches. While the first two batch-based transitions model three states, the third example represents only two.

2.5 Dynamic Graph Analysis - Problem Statement

Given a dynamic graph G, a set of metrics $M \subseteq \mathcal{M}$, and a list of timestamps $T = (t_0, t_1, \ldots)$, the *problem of dynamic graph analysis* is to compute the properties of all metrics $m \in M$ for each timestamp $t \in T$, i.e., to compute: (1) vertex values ($p \in P_{V \to \mathbb{R}}(m)$), (2) vertex pair values ($p \in P_{V \times V \to \mathbb{R}}(m)$), (3) graph values ($p \in P_{\mathcal{G} \to \mathbb{R}}(m)$), and (4) graph distributions ($p \in P_{\mathcal{G} \to \mathcal{F}}(m)$). We refer to the descriptions of this problem given by G, M, and T as a *scenario* or an *application*.

In the following, we generalize the notation for the value of a property $p \in P(m)$ of metric m at timestamp t as $p(G_t)$. This value is defined depending on its type, i.e.:

$$p(G_t) \coloneqq \begin{cases} p(v) \in \mathbb{R} : v \in V_t & \text{if} \quad p \in P_{V \to \mathbb{R}}(m) \\ p(v, w) \in \mathbb{R} : (v, w) \in V_t \times V_t & \text{if} \quad p \in P_{V \times V \to \mathbb{R}}(m) \\ p(G_t) \in \mathbb{R} & \text{if} \quad p \in P_{\mathcal{G} \to \mathbb{R}}(m) \\ p(G_t) \in \mathcal{F} & \text{if} \quad p \in P_{\mathcal{G} \to \mathcal{F}}(m) \end{cases}$$

2.6 Algorithms for Dynamic Graph Analysis

For a metric $m \in \mathcal{M}$, different algorithms exist that compute the corresponding properties $P(m)$. We denote the set of all algorithms that compute a metric m as $\mathcal{A}(m)$ and the metric computed by an

algorithm a as $m(a)$. For a property p of a metric m, i.e., $p \in P(m)$, let $p(G_t, a)$ denote the vertex value, vertex pair values, graph value, or graph distribution computed by algorithm a for the dynamic graph G at timestamp t. Then, $P(G_t, a) \coloneqq \{p(G_t, a) : p \in P(m(a))\}$ is the set of all results computed by the algorithm a where $P(m(a))$ denotes the set of all properties of the metric $m(a)$ computed by algorithm a.

In the remainder of this Section, we describe the comparison of algorithms and different algorithm types. First we introduce concepts for the comparison, correctness, and quality of algorithms in Section 2.6.1. Then, we describe three types of algorithms for the analysis of dynamic graphs. We introduce snapshot-based algorithms in Section 2.6.2, stream-based algorithms in Section 2.6.3, and batch-based algorithms in Section 2.6.4. We close this Section with an input-based classification of algorithms for the analysis of dynamic graphs in Section 2.6.5.

2.6.1 Algorithm Comparison

We call two algorithms a and a' *comparable* iff they compute the same metric, i.e., $m(a) = m(a')$. We say that two comparable algorithms a and a' *compute the same results* for graph G at timestamp t iff $\forall\, p \in P(m(a)) : p(G_t, a) = p(G_t, a')$, i.e., all properties computed by a are same as those computed by a'.

We call a deterministic algorithm a *correct* or *precise* iff $\forall\, p \in P(m(a)), G \in \mathcal{G}, t \in \mathbb{N} : p(G_t) = p(G_t, a)$ and *incorrect* or *imprecise* otherwise. An incorrect algorithm can either be *flawed* or a *heuristic*, i.e., an approximation that is known to not always compute exact results. Note that such a differentiation is not applicable to indeterministic algorithm. Examples are clustering or community detection algorithms.

For a heuristic $h \in \mathcal{A}(m)$, we define the *approximation quality of a graph value* $p \in P_{\mathcal{G}\to\mathbb{R}}(m)$ for a dynamic graph G at timestamp t as $q(p, G, t, h) \coloneqq \frac{p(G_t,h)}{p(G_t)}$, i.e., the relative value compared to the correct result. The approximation quality of vertex values, vertex pair values, and graph distributions is defined likewise.

2.6.2 Snapshot-based Algorithms

Snapshot-based algorithms are commonly used for the computation of a metric m on a static graph $G = (V, E)$. We denote the *set of all snapshot-based algorithms* as $\mathcal{A}_S$ and the set of those computing metric m as $\mathcal{A}_S(m)$.

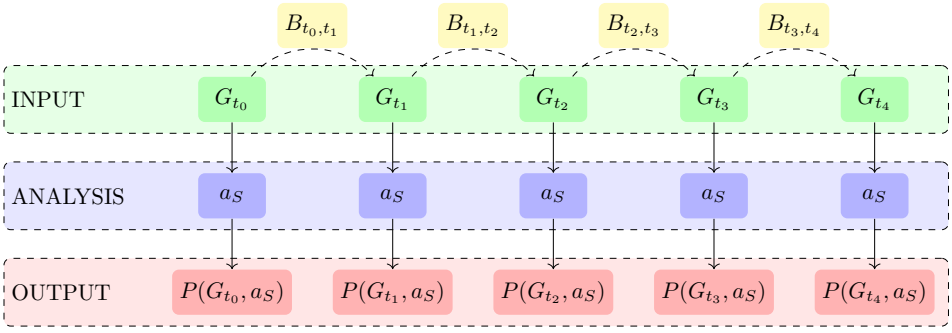


Figure 2.6: Workflow of snapshot-based algorithms for dynamic graph analysis

To compute the properties $p(G_t), t \in T$ for a dynamic graph G, a snapshot-based algorithm a_S is executed once for each snapshot $G_t, t \in T$ as shown in Figure 2.6. As input, snapshot-based algorithms take only the snapshot of interest G_t and output the set of computed properties $P(G_t, a_S)$.

Since snapshot-based algorithms only take the state G_t as input, they are often processed separate from each other. Then, each state of the graph is read from a single file without any connection to other states. Nevertheless, the transition between two consecutive states $G_{t_{i-1}}$ and G_{t_i} can be represented as a batch B_{t_{i-1},t_i}. Therefore, it is also possible to read only the initial graph G_{t_0} in its entirety and generate consecutive states by reading and applying the corresponding batch. We refer to the first approach as *separate snapshot-based processing* and to the second one as *consecutive snapshot-based processing*.

2.6.3 Stream-based Algorithms

Stream-based algorithms apply the general idea of stream processing [257, 19, 4] to the problem of computing the properties of dynamic graphs based on a stream of updates [59, 244] as introduced in Section 2.4.

We denote the *set of all stream-based algorithms* as $\mathcal{A}_U$ and the set of those computing a metric m as $\mathcal{A}_U(m)$.

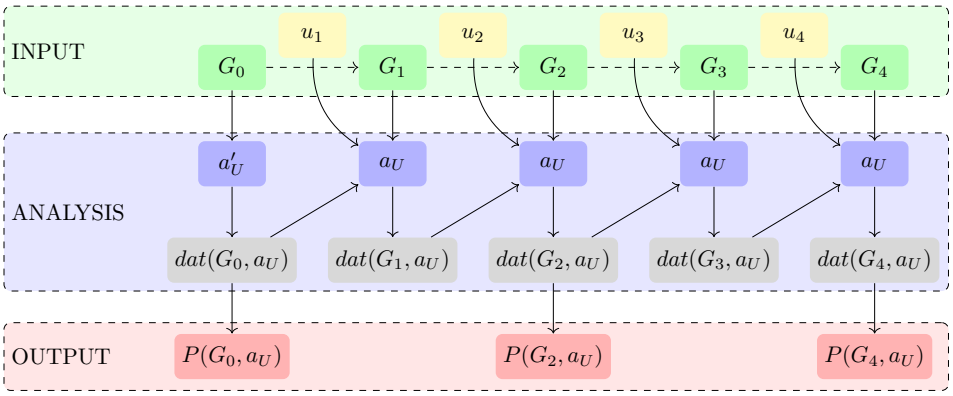


Figure 2.7: Workflow of stream-based algorithms for dynamic graph analysis

For each state G_i of the analyzed dynamic graph G, a *stream-* or *update-based algorithm* $a_U \in \mathcal{A}_U$ computes some data $dat(G_i, a_U)$ as indicated in Figure 2.7. It contains results, partial results, and auxiliary information required for the next steps.

As initialization, $dat(G_0, a_U)$ is computed by an algorithm a'_U for the initial snapshot G_0. In case $dat(G_i, a_U)$ only consists of the desired properties and no auxiliary data, i.e., $dat(G_i, a_U) = P(G_i, a_U)$, any snapshot-based algorithm can be used, i.e., $a'_U \in \mathcal{A}_S(m(a_U))$. Another possibility for initializing $dat(G_0, a_U)$ is to start the analysis with an empty graph $G'_0 = (\emptyset, \emptyset)$ and build G_0 from a stream of vertex and edge additions $add^V(v) : v \in V_0$ and $add^E(e) : e \in E_0$. Then, a_U can simply be executed for each of these updates. The initialization of auxiliary data and properties for an empty graph depends on each metric and algorithm.

For any further state, a stream-based algorithms a_U takes as input the current state G_i of the graph, the latest update u_i, and the previous data $dat(G_{i-1}, a_U)$ as shown in Figure 2.7. Using this information, the algorithm outputs the data for the current state, i.e., $dat(G_i, a_U)$.

The problem of dynamic graph analysis requires an algorithm to output the results for a list of timestamps $T = (t_0, t_1, \dots)$ but not all states. Hence, for every state G_i with $t(i) \in T$, the desired results $P(G_{t(i)}, a_U)$ are extracted, aggregated, or computed from $dat(G_i, a_U)$. Commonly, this step is referred to as a query [88]. In case the results are already maintained as part of $dat(G_i, a_U)$, this step does not require any further computation.

2.6.4 Batch-based Algorithms

Batch-based algorithms perform the analysis of a dynamic graph similar to stream-based approaches. Instead of investigating each update separately, they consider the changes at the more coarse-grained view of whole batches. We denote the *set of all batch-based algorithms* as $\mathcal{A}_B$ and the set of those computing metric m as $\mathcal{A}_B(m)$.

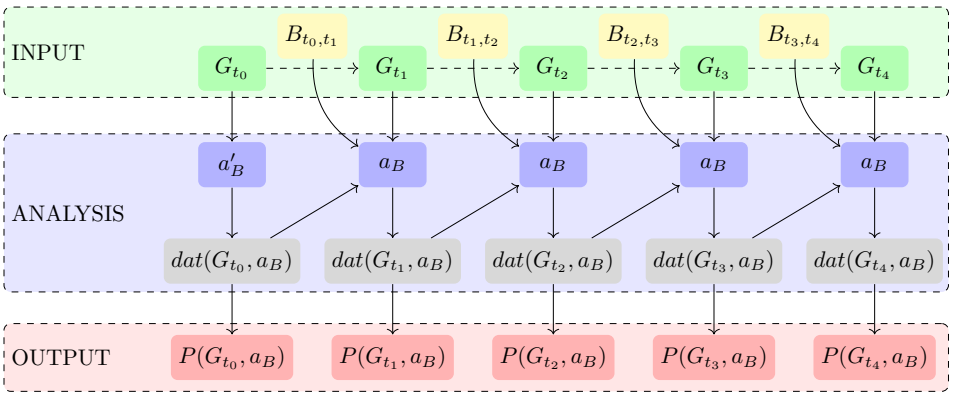


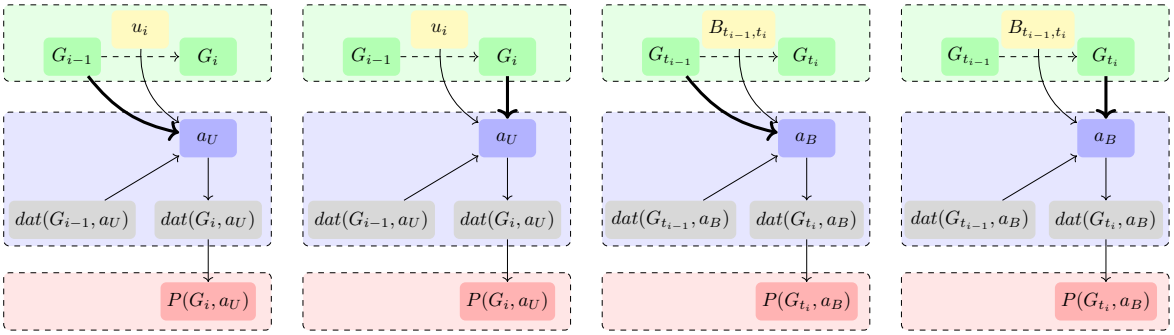
Figure 2.8: Workflow of batch-based algorithms for dynamic graph analysis

For each timestamp $t \in T$, a batch-based algorithm $a_B \in \mathcal{A}_B$ outputs the data $dat(G_t, a_B)$ which is then transformed into $P(G_t, a_B)$ as shown in Figure 2.8. Similar to stream-based algorithms, the initial

data $dat(G_{t_0}, a_B)$ is computed by a'_B which can again be a snapshot-based algorithm $a'_B \in \mathcal{A}_S(m(a_B))$ that takes G_0 as input or a stream-based algorithm $a'_B \in \mathcal{A}_U(m(a_B))$ which starts from an empty graph. For all other timestamps t_i, a batch-based algorithm a_B takes as input the current snapshot G_{t_i} of the graph, the batch B_{t_{i-1},t_i}, and the previously computed data $dat(G_{t_{i-1}}a_B)$. It outputs $dat(G_{t_i}, a_B)$ which is then used to generate $P(G_{t_i}, a_B)$ analogously to the query step of stream-based algorithms.

2.6.5 Input-based Classification of Algorithms

The application of a single update u_i changes the in-memory representation of the analyzed dynamic graph from the state G_{i-1} to G_i. So far, we simply assumed that a stream-based algorithm $a_U \in \mathcal{A}_U$ is executed *after the application of an update* as shown in Figures 2.7 and 2.9b. Then, a_U takes as input the update u_i, the previously computed data $dat(G_{i-1}, a_U)$ and the state G_i of the graph after the application of u_i. It is also possible to execute a_U *before the application of an update* u_i. In that case, the algorithm takes G_{i-1} as input in addition to u_i and $dat(G_{i-1}, a_U)$ as shown in Figure 2.9a.



(a) a_U before u_i (b) a_U after u_i (c) a_B before B_{t_{i-1},t_i} (d) a_B after B_{t_{i-1},t_i}

Figure 2.9: Execution of stream-/batch-based algorithms before and after application of update/batch

Similarly, the application of a whole batch B_{t_{i-1},t_i} transforms the state of the graph stored in memory from $G_{t_{i-1}}$ to G_{t_i}. Again, we assumed that batch-based algorithms $a_B \in \mathcal{A}_B$ are executed *after the application of a batch* as shown in Figures 2.8 and 2.9d. Hence, a_B takes the state after the application G_{t_i} as input as well as the data $dat(G_{t_{i-1}}, a_B)$ and the batch B_{t_{i-1},t_i}. We can also consider the case of executing a_B before the application of the batch. Then, the algorithms takes $G_{t_{i-1}}$ as input as well as B_{t_{i-1},t_i} and $dat(G_{t_{i-1}}, a_B)$ as shown in Figure 2.9c.

In general, a stream-based algorithm can be executed either only before, only after, or before and after the application of an update. More precisely, this distinction can be made for each of the six update types. For example, a stream-based algorithm might need to be executed before and after the removal of vertices but only before the addition of edges. In case an algorithm does not consider edge weights, it must not be executed at all for updates that change them. Hence, algorithms can specify whether or not they are executed before or after the application of each type of update.

Batch-based algorithms can be executed before or after the application of a batch. In contrast to stream-based algorithms, a specification for specific update types is not possible. Therefore, three different scenarios can be distinguished: either before, after, as well as before and after the application of a batch.

Algorithm Type	**Notation**	**Execution**	**Input**			**Output**
Snapshot-based	$a_S \in \mathcal{A}_S$	-	G_{t_i}			$P^{a_S}(G_{t_i})$
Stream-based	$a_U \in \mathcal{A}_U$	before	G_{i-1}	u_i	$dat^{a_U}_{i-1}$	$dat^{a_U}_i$
		after	G_i	u_i	$dat^{a_U}_{i-1}$	$dat^{a_U}_i$
Batch-based	$a_B \in \mathcal{A}_B$	before	$G_{t_{i-1}}$	B_{t_{i-1},t_i}	$dat^{a_B}_{t_{i-1}}$	$dat^{a_B}_{t_i}$
		after	G_{t_i}	B_{t_{i-1},t_i}	$dat^{a_B}_{t_{i-1}}$	$dat^{a_B}_{t_i}$

Table 2.4: Input-based classes of algorithms for processing a dynamic graph

For snapshot-based algorithms, the input is always a single snapshot of the graph, i.e., G_{t_i}. The five general classes of inputs and their corresponding output are given in Table 2.4. Note that the only difference between the before and after versions of stream- and batch-based algorithms is the state of the graph taken as input, i.e., G_{i-1} compared to G_i or $G_{t_{i-1}}$ compared to G_{t_i}.