Graphs and Graph Algorithms

Algorithmics study and revision notes

Jaak Vilo

2026-08-22

Contents

1 Why graphs?

A graph represents entities and relationships without requiring a hierarchy or linear order. Roads connect places, links connect web pages, reactions connect molecules, and dependencies connect tasks. The same abstract model supports very different questions: Is a target reachable? What is the cheapest route? Which links connect every vertex cheaply? Which groups circulate strongly among themselves? How much material can a network carry?

Graph algorithms are only as meaningful as the model. Direction, weights, parallel edges, self-loops, and the meaning of an absent edge must be settled before selecting an algorithm.

2 Learning goals

After studying this note, you should be able to:

- define graph terminology and select a representation;
- derive and trace BFS and DFS from a frontier discipline;
- detect cycles, topologically order a DAG, and compute SCCs;
- justify Kruskal's and Prim's algorithms with the cut property;
- use relaxation to distinguish shortest-path algorithms by their assumptions;
- state the conditions that make A* optimal;
- formulate transitive closure, PageRank, MCL, and maximum flow; and
- connect invariants, correctness arguments, and complexity to implementation choices.

3 Prerequisites

- Lists, matrices, stacks, queues, and priority queues.
- Trees and disjoint-set union (union-find).
- Asymptotic analysis and loop invariants.
- Basic probability and matrix-vector multiplication for the stochastic sections.

4 Graph model and terminology

The note proceeds in layers. First fix the graph model and representation; then use frontier order to derive traversal and structural algorithms. Weighted optimisation adds safe-edge or relaxation arguments. The final sections reinterpret edges as stochastic flow or residual capacity. Keeping those proof ideas separate makes the large catalogue easier to navigate.

For a first pass, treat representations, BFS, DFS, SCCs, MSTs, shortest paths, and maximum flow as the core chain. Then return to PageRank, MCL, distance indexes, and community methods as extensions that introduce additional models and objectives.

A graph is $G = (V, E)$, with vertices V and edges E.

- In an **undirected** graph an edge $\{u, v\}$ has no orientation.
- In a **directed** graph an edge (u, v) goes from u to v.
- A **weighted** graph associates a value $w(e)$ with each edge.
- A **simple** graph has no parallel edges or self-loops; algorithms may need adaptation when these are allowed.

A **walk** follows adjacent edges and may repeat vertices. A **path** is normally taken to have no repeated vertices. A **cycle** returns to its start. In an undirected graph, a connected component is a maximal mutually reachable vertex set. In a directed graph:

- a **weakly connected** component ignores edge directions;
- a **strongly connected component** (SCC) contains vertices that reach one another in both directions.

The degree of an undirected vertex counts incident edges. A directed vertex has indegree and outdegree. State whether an undirected edge is stored once or twice and whether a self-loop contributes one or two to degree.

5 Representations

An **adjacency matrix** uses $\Theta(|V|^2)$ cells and tests an edge in $O(1)$. It is suitable for dense graphs and matrix algorithms. A distinguished value must separate "no edge" from a legitimate zero-weight edge.

An **adjacency list** stores one neighbour record per directed edge, or two per undirected edge. It uses $\Theta(|V| + |E|)$ space and enumerates neighbours of u in $\Theta(deg(u))$ time. Hashing or sorting a neighbour list can accelerate edge tests at additional cost.

An **edge list** uses $\Theta(|E|)$ records and is convenient when algorithms scan or sort all edges, as Kruskal's algorithm does.

Choose a representation according to the operations, not merely the input format. With adjacency lists,

$$\sum_{u \in V} |Adj[u]| = \begin{cases} |E| & \text{directed,} \\ 2|E| & \text{undirected,} \end{cases}$$

which is why a complete adjacency scan is $\Theta(|V| + |E|)$.

6 A unifying idea: process a frontier

Many searches maintain discovered but not fully processed vertices in a **frontier**:

- FIFO queue: breadth-first search;
- LIFO stack or recursion: depth-first search;
- minimum-priority queue: Dijkstra, Prim, or A*;
- randomised container: exploratory search.

The container changes the order, but three correctness rules recur:

1. in once-discovery searches such as BFS and DFS, mark a vertex when it enters the frontier, not after it leaves;
2. record the edge that first or best reached it; and
3. process every relevant outgoing edge according to the algorithm's invariant.

Priority searches distinguish **discovered** from **settled**. Dijkstra and A* may improve a discovered vertex's key before it is permanently settled.

7 Breadth-first search

BFS explores an unweighted graph in nondecreasing number of edges from a source.

```
BFS(G, s):
    for each v: distance[v] = infinity; parent[v] = none
    distance[s] = 0
    Q = FIFO queue containing s
    while Q is not empty:
        u = Q.pop_front()
        for each v in Adj[u]:
            if distance[v] == infinity:
                distance[v] = distance[u] + 1
                parent[v] = u
                Q.push_back(v)
```

Layer invariant. When a vertex of distance d is removed, all vertices already removed have distance at most d, and every queued vertex has distance d or $d+1$. Any path to an undiscovered neighbour v through u has length $d+1$. A shorter path would have discovered v from an earlier layer. Thus first discovery assigns the shortest unweighted distance.

Parent pointers form a BFS tree for the reachable component and reconstruct a shortest path by walking backward from a target. To traverse a disconnected graph, start another BFS from every still-undiscovered vertex.

With adjacency lists, each vertex enters once and every stored edge record is inspected once, giving $\Theta(|V| + |E|)$ time and $O(|V|)$ auxiliary space.

8 Depth-first search, cycles, and topological order

DFS fully explores one branch before returning. Colour vertices white (unseen), gray (active), and black (finished).

```
DFS_visit(u):
    colour[u] = gray
    discovery_time[u] = next_time()
    for each v in Adj[u]:
        if colour[v] == white:
            parent[v] = u
            DFS_visit(v)
        else:
            classify or process edge (u, v)
    colour[u] = black
    finish_time[u] = next_time()
```

Run this visit from every remaining white vertex to obtain a DFS forest. Recursive calls are nested, so discovery/finish intervals are either disjoint or one contains the other.

In a directed graph, an edge to a gray ancestor is a **back edge** and proves a directed cycle. Conversely, if a directed cycle exists, DFS encounters a back edge when it follows the cycle from its first discovered vertex.

In a simple undirected graph, ignore the edge back to the immediate parent; an edge to any other visited vertex proves a cycle. In a multigraph, track edge identities and ignore only the exact tree edge: a second parallel edge to the parent forms a two-edge cycle. A self-loop is a cycle as soon as it is seen.

A **directed acyclic graph** (DAG) admits a topological ordering in which every edge $u \to v$ places u before v. Reverse DFS finish order gives one:

```
topological_sort(G):
    run DFS, rejecting any back edge
    return vertices in decreasing finish time
```

For any DAG edge $u \to v$, DFS either visits v below u, so v finishes first, or v has already finished. It cannot be gray because that would be a cycle. Hence $finish(u) > finish(v)$ and reversed finish order is topological. Kahn's alternative repeatedly removes an indegree-zero vertex; processing fewer than $|V|$ vertices reveals a cycle.

9 Strongly connected components

Contracting every SCC to one vertex produces the **condensation graph**. It must be a DAG: a directed cycle among components would make all vertices on that cycle mutually reachable and therefore one SCC.

Kosaraju's algorithm computes SCCs in linear time:

```
SCC(G):
    order = decreasing_finish_order(DFS(G))
    return DFS_forest(transpose(G), order)
```

The first pass orders source/sink relationships between components. A component with latest remaining finish time is a source in the remaining condensation graph of G and therefore a sink after transposition. Starting there reaches its own component but cannot escape to an unprocessed one. Induction gives one SCC per second-pass tree.

Both passes and transposition take $\Theta(|V| + |E|)$. Tarjan's algorithm obtains the same partition in one DFS using discovery indices, a stack, and low-link values.

10 Minimum spanning trees

For a connected undirected weighted graph, a **spanning tree** connects all vertices with exactly $|V| - 1$ edges. A minimum spanning tree (MST) minimises their total weight. It need not minimise path distances from a source, and equal weights may permit several MSTs.

The core correctness fact is the **cut property**:

> If a cut respects the already chosen forest, then a lightest edge crossing that cut is safe: it belongs to some MST containing the chosen edges.

To prove it, take an MST T containing the chosen forest. If T already contains the light edge e, nothing is needed. Otherwise add e to T, creating a cycle. That cycle contains another crossing edge f. Replacing f by e preserves a spanning tree and cannot increase weight, so the new tree is also minimum and contains e.

Kruskal grows a forest:

```
Kruskal(G):
    make_set(v) for every vertex v
    A = empty set
    for each edge (u, v) in nondecreasing weight:
        if find(u) != find(v):
            add (u, v) to A
            union(u, v)
    return A
```

Each chosen edge is the lightest crossing between two current components. Sorting dominates at $O(|E| \log |E|)$; for a simple graph this is $O(|E| \log |V|)$. Union-find adds near-linear time.

Prim grows one tree. For every outside vertex v, maintain the lightest known edge from the current tree to v as its priority-queue key. Extracting the smallest key selects a light edge across the current tree cut; relax incident edges with decrease-key. A binary heap gives $O((|V| + |E|) \log |V|)$, commonly written $O(|E| \log |V|)$ for a connected graph. A dense matrix implementation takes $O(|V|^2)$.

On a disconnected graph these algorithms produce a minimum spanning forest. Negative edge weights cause no problem; the cut proof does not require nonnegative weights.

11 Weighted shortest paths and relaxation

The weight of a walk or path is the sum of its edge weights. Shortest-path algorithms conventionally minimise over source-to-target walks, although a finite optimum can be represented by a path: any repeated cycle that does not lower the cost can be removed.

Let $\delta(s, v)$ be the infimum weight of an s-to-v walk, or infinity when v is unreachable. If no relevant negative cycle exists, this infimum is attained by a path. If a negative-weight cycle is reachable from s and can reach v, then $\delta(s, v) = -\infty$: traversing the cycle repeatedly makes walks arbitrarily cheap.

Shortest-path algorithms maintain estimates $d[v]$, initially $d[s] = 0$ and all others infinity. **Relaxing** $u \to v$ tests

$$d[u] + w(u, v) < d[v].$$

If true, set $d[v] = d[u] + w(u, v)$ and $parent[v] = u$. Every finite estimate is the weight of an actual discovered walk, so it is an upper bound on the true shortest distance. Different algorithms arrange enough relaxations in an order that makes those bounds final.

11.1 DAG shortest paths

Topologically order a DAG and relax every outgoing edge in that order. Every predecessor of a vertex is processed first, so its final distance is known when needed. This permits negative edges and takes $\Theta(|V| + |E|)$.

11.2 Dijkstra's algorithm

For nonnegative edges, store unsettled vertices in a min-priority queue keyed by d:

```
Dijkstra(G, s):
    initialise d and parent; put s in a min-priority queue
    while the queue is not empty:
        u = extract_min()
        if u was already settled: continue
        mark u settled
        for each edge (u, v):
            if d[u] + w(u, v) < d[v]:
                update d[v] and parent[v]
                insert or decrease_key(v)
```

Settled invariant. When u is extracted, $d[u] = \delta(s, u)$. If a cheaper path existed, consider the first unsettled vertex y on that path and its settled predecessor x. Relaxing $x \to y$ would give $d[y]$ no larger than the path prefix, and nonnegative remaining edges make that at most the supposed cheaper distance to u. Then u could not have the minimum queue key.

A binary heap gives $O((|V|+|E|) \log |V|)$; a Fibonacci heap gives $O(|E|+|V| \log |V|)$ amortised. Dijkstra is invalid with negative edges because an extracted distance may later improve.

11.3 Bellman-Ford

Bellman-Ford makes $|V| - 1$ complete passes over the edge list. Any simple shortest path has at most $|V| - 1$ edges, so after pass i, every shortest path using at most i edges has been propagated. A further improving relaxation identifies a negative cycle reachable from the source.

The time is $O(|V||E|)$ and space is $O(|V|)$. Stop early if one full pass changes nothing. Bellman-Ford detects only negative cycles reachable from the chosen source unless an artificial source connects to every vertex.

11.4 A* as goal-directed exact search

A* searches for one target in a finite graph with nonnegative edge costs using

$$f(v) = g(v) + h(v),$$

where $g(v)$ is the best known source-to-v cost and $h(v)$ estimates the remaining cost. It expands the smallest f.

A heuristic is **admissible** if $0 \le h(v) \le h^*(v)$, the true remaining cost; in particular, $h(t) = 0$ at a target t. It is **consistent** if every edge satisfies

$$h(u) \le w(u, v) + h(v).$$

If A* tests for a goal when that state is removed from the priority queue, admissibility gives an optimal result when distinct paths are retained as distinct tree-search nodes. On a cyclic graph, such tree search also needs a termination condition, such as all edge costs being bounded below by a positive constant.

In the common graph-search implementation that merges equal states and permanently closes expanded vertices, consistency makes f nondecreasing along paths and ensures closed distances are final. With an admissible but inconsistent heuristic, optimal graph search must be prepared to reopen a vertex whose g improves. Setting $h = 0$ reduces A* to Dijkstra.

12 Matrix path algorithms

An adjacency matrix can encode reachability over the Boolean semiring: multiplication becomes AND and addition becomes OR. Warshall's transitive-closure algorithm asks whether a path exists using only the first k vertices as internal vertices.

Floyd-Warshall applies the same intermediate-vertex idea to all-pairs shortest paths. Initialise $D^{(0)}[i, j]$ with edge weights, zero on the diagonal, and infinity for absent edges. Then

$$D^{(k)}[i, j] = \min\left(D^{(k-1)}[i, j], D^{(k-1)}[i, k] + D^{(k-1)}[k, j]\right).$$

Either an optimal permitted path avoids k, or it reaches k once and splits into two smaller permitted paths. The algorithm uses $O(|V|^3)$ time and $O(|V|^2)$ space, permits negative edges, and reveals a negative cycle if some final diagonal entry is negative.

13 Random walks, PageRank, and MCL

Let P be a row-stochastic transition matrix: P_{uv} is the probability of moving from u to v. A dangling vertex with no outgoing edges has no probability distribution, so first replace its row with a chosen distribution v^T, often uniform.

For damping $0 < \alpha < 1$ and a positive teleport distribution v, PageRank is the probability vector satisfying

$$r = \alpha P^T r + (1 - \alpha)v.$$

Power iteration repeatedly applies the right-hand side and renormalises. Dangling-row repair defines every transition; teleportation then prevents closed regions from trapping all mass and, under the usual positive choice, gives a unique well-behaved stationary ranking. State the row/column convention when implementing the equation.

The **Markov Cluster Algorithm (MCL)** uses random-walk flow for clustering. Commonly, first add self-loops to a nonnegative adjacency matrix and normalise every column, obtaining a column-stochastic matrix M. Then alternate:

1. **expansion:** matrix powering, commonly $M \leftarrow M^2$, to propagate multi-step flow;
2. **inflation:** apply

$$M_{ij} \leftarrow \frac{M_{ij}^r}{\sum_k M_{kj}^r}, \qquad r > 1,$$

   independently in every column.

Inflation strengthens dominant within-region flow and suppresses weak cross-region flow. Repeat expansion and inflation until a stated convergence tolerance is met; implementations extract clusters from the resulting attractor/support pattern and prune tiny entries to keep matrices sparse. The parameter r controls granularity. MCL means Markov clustering, not Monte Carlo clustering. Its column-stochastic convention is separate from the row-stochastic PageRank convention above.

14 Maximum flow

A flow network is a directed graph with source s, sink t, and capacities $c(u, v) \geq 0$. A feasible flow satisfies

$$0 \leq f(u, v) \leq c(u, v)$$

and flow conservation at every vertex other than s, t. Its value is net flow leaving s.

The **residual network** records allowed changes. A used forward edge has residual capacity $c(u, v) - f(u, v)$; its reverse residual edge has capacity $f(u, v)$ and permits undoing a previous decision.

Ford-Fulkerson repeats three steps: find an s-t residual path P; let δ be the minimum residual capacity on P; then augment δ units along P, updating both forward and reverse residual edges.

With integral capacities, every augmentation increases flow value by at least one, so the method terminates; its running time can depend on the numeric maximum-flow value. With irrational capacities and arbitrary path choices, it need not terminate. Edmonds-Karp always chooses a shortest residual path by BFS and runs in $O(|V||E|^2)$.

For any cut $(S, V - S)$ with $s \in S$ and $t \notin S$, flow value is at most the capacity of edges from S to $V - S$. When no residual path exists, let S be the vertices reachable from s in the residual graph. Every forward cut edge is saturated and every reverse contribution is zero, so the current flow value equals that cut capacity. The flow and cut therefore certify each other's optimality: this is the max-flow/min-cut theorem.

Contracting vertices can change internal bottlenecks. In particular, summing all capacities between two SCCs does not generally preserve maximum flow; a component's entrances and exits may be connected through limited internal capacity.

15 Enrichment: distance indexes and communities

For repeated distance queries in a large undirected graph, precompute distances to selected **landmarks**. Triangle inequalities give lower bounds such as

$$|d(\ell, u) - d(\ell, v)| \leq d(u, v)$$

and upper bounds through ℓ. Landmark bounds can guide A* or avoid a full search for approximate queries. Preprocessing time, storage, landmark selection, updates, and approximation error must all be measured.

Community detection asks for densely related groups, but "community" needs an objective. Girvan-Newman removes high-betweenness edges; modularity-based methods compare within-group edges with a degree-preserving null model; Louvain greedily aggregates modularity-improving groups. These are modelling and optimisation methods, not replacements for the exact definition of an SCC.

16 Worked trace: Dijkstra

Consider directed edges

$$s \to a : 1, \quad s \to b : 4, \quad a \to b : 2, \quad a \to t : 5, \quad b \to t : 1.$$

Initial estimates are $d[s] = 0$ and infinity elsewhere.

1. Extract s. Relaxing its edges gives $d[a] = 1$ and $d[b] = 4$.
2. Extract a. The route through a improves $d[b]$ to 3 and sets $d[t] = 6$.
3. Extract b. Its edge improves $d[t]$ to 4.
4. Extract t. The predecessor chain is $t \leftarrow b \leftarrow a \leftarrow s$, so the shortest path is s, a, b, t of weight 4.

The direct-looking route s, a, t weighs 6. Dijkstra commits vertices in increasing final distance, not by the number of edges or by the locally smallest outgoing edge.

17 Complexity and trade-offs

Problem	Method	Time with usual representation	Required condition
Unweighted distances	BFS	$\Theta(\|V\| + \|E\|)$	unit edge cost
DFS forest / cycle test	DFS	$\Theta(\|V\| + \|E\|)$	none
Topological order	DFS or Kahn	$\Theta(\|V\| + \|E\|)$	graph must be a DAG
SCCs	Kosaraju or Tarjan	$\Theta(\|V\| + \|E\|)$	directed graph
MST	Kruskal / Prim	$O(\|E\| \log \|E\|)$ / $O((\|V\| + \|E\|) \log \|V\|)$	undirected weighted graph
Nonnegative SSSP	Dijkstra	$O((\|V\| + \|E\|) \log \|V\|)$	nonnegative edges
Goal-directed SSSP	A*	$O((\|V\| + \|E\|) \log \|V\|)$ worst case with a binary heap	nonnegative edges; consistent heuristic for permanent closing
General SSSP	Bellman-Ford	$O(\|V\|\|E\|)$	reports reachable negative cycle
All-pairs shortest paths	Floyd-Warshall	$O(\|V\|^3)$	no negative cycle for finite answers
PageRank	power iteration	$O(\|V\| + \|E\|)$ per sparse iteration	repaired transitions, damping, and tolerance
MCL	sparse expansion and inflation	data-dependent; fill-in may dominate	nonnegative matrix, pruning, and tolerance
Maximum flow	Edmonds-Karp	$O(\|V\|\|E\|^2)$	nonnegative capacities

18 Connections

- Queues, stacks, and priority queues create different frontier orders.
- Union-find supplies Kruskal's component test; heaps supply Prim, Dijkstra, and A*.
- Topological order connects DAG paths to dynamic-programming evaluation order.
- Floyd-Warshall is a dynamic program over permitted intermediate vertices.
- PageRank and MCL reinterpret edges as stochastic flow.
- Residual edges in maximum flow formalise the ability to revise earlier choices.
- Heuristic search will revisit how A* heuristics reduce explored state space.

19 Common mistakes

- Failing to state whether a graph is directed, weighted, or simple.
- Marking BFS vertices only when removed, allowing repeated queue entries.
- Using an undirected parent edge as evidence of a cycle.
- Running Dijkstra with negative edges.
- Confusing an MST with a shortest-path tree.
- Claiming admissibility alone justifies a closed-set A* implementation without reopening.
- Applying PageRank before defining dangling rows and matrix orientation.
- Forgetting reverse residual capacity in maximum flow.
- Treating SCCs, stochastic communities, and modularity communities as the same concept.

20 Self-check

1. When is an adjacency matrix preferable to adjacency lists?
2. Prove that first BFS discovery gives a minimum-edge-count path.
3. Why does a directed DFS back edge characterise a cycle?

4. Why must the SCC condensation graph be acyclic?
5. State and prove the MST cut property by exchange.
6. What fails in Dijkstra's settled-vertex proof when an edge is negative?
7. When may A* safely avoid reopening a closed vertex?
8. How are dangling-row repair and teleportation different?
9. What does a reverse residual edge mean?
10. How does the final residual reachable set certify a minimum cut?

21 Revision summary

- Model edge direction, weight, multiplicity, and absence before choosing an algorithm.
- BFS and DFS derive their guarantees from frontier order and discovery invariants.
- Finish order organises DAGs and SCCs.
- MST correctness comes from safe cut edges; shortest paths come from ordered relaxation.
- A* is exact only under explicit heuristic and graph-search conditions.
- Matrix methods organise paths by allowed intermediate vertices.
- PageRank and MCL use normalised flow for ranking and clustering.
- Residual networks make flow decisions reversible and expose an optimal cut.

22 Implementations and hands-on exploration

- NetworkX offers Python graph types and traversal, shortest-path, spanning-tree, and flow algorithms. It suits modelling and reference checks; Python-object overhead makes it a poor low-level baseline for very large graphs.
- igraph's C library provides efficient construction and network analysis, with bindings for other languages. Check vertex numbering, direction, parallel edges, and each routine's weight convention.
- Boost.Graph supplies generic C++ traversals, components, shortest paths, MST, A*, and flow. Its templates require specific graph and property-map concepts, so representation is part of the interface.
- Rust's petgraph provides adjacency-list graphs and common algorithms. Removing from compact `Graph` may shift indices; use `StableGraph` when external indices must survive deletion.
- JGraphT supplies Java graph variants and algorithms for paths, spanning structures, connectivity, and flow. Direction, parallel-edge, and self-loop policies depend on the chosen graph type.

22.1 Small experiments

1. Represent the same generated graphs as an adjacency matrix and adjacency lists. Run your own BFS while counting neighbour cells or edge records examined, and compare sparse and dense cases before comparing wall-clock time.
2. Differential-test small random instances against a library: BFS distances should equal Dijkstra distances when every edge weighs one; Prim and Kruskal should return equal total forest weight; and a reported maximum flow should equal the capacity of its residual reachable cut.

23 Sources and further study

- Graphs I deck and Graphs II deck, Jaak Vilo.
- (Cormen et al. 2022), Chapters 20-24.
- (Kleinberg and Tardos 2006), Chapters 3-4 and 7.

References

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2022. *Introduction to Algorithms*. 4th ed. MIT Press.

Kleinberg, Jon, and Éva Tardos. 2006. *Algorithm Design*. Pearson.