

Algorithms, Datastructures & Probability

Cheatsheet for HS-2022 and FS-2023

This document and it’s graphics are licensed under CC BY-SA 4.0. It may be shared, adapted and used for any purpose, as long as appropriate attribution is given and any adaptations are shared under the same license.

The \LaTeX source code is available under github.com/rechenmaschine/eth-cheatsheets. Feel free to contribute or submit issues there.

Contributors
Louis Schell

Graphs

A graph is a pair (V, E) where V is a finite set of vertices and E a finite set of edges which connect the vertices. Graphs are useful in modelling many real-world problems.

Edges are pairs of vertices: $(u, v) \in E$

In a graph without loops and duplicates it holds that:

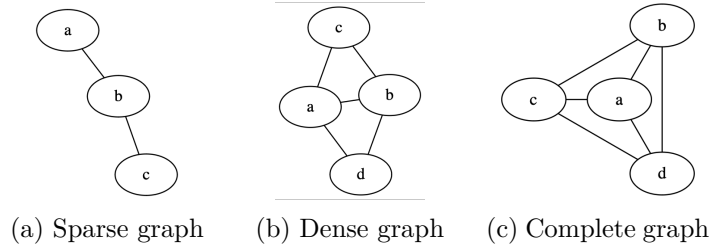
$$|E| \leq \binom{n}{2} = \frac{(n-1) \cdot n}{2} \leq \mathcal{O}(|V|^2)$$

A **complete** graph connects all vertices, ie. for every distinct $v, u \in V$ there exists an edge from v to u .

A **directed** graph has only directed edges. In contrast, an **undirected** graph has bidirectional edges, ie. for every $(v, u) \in E$ there exists $(u, v) \in E$.

A graph is **(strongly) connected**, if there is a (di-rected) path between every $u, v \in V$. Otherwise, the graph is **disconnected**. Additionally, a directed graph is **weakly connected** if the undirected superset of its edges forms a connected graph.

An undirected, connected graph with $|V| \leq |E|$ always has cycles.



Directed Acyclic Graphs (DAG)

A **directed acyclic graph** is a directed graph with no cycles. DAGs are useful for representing dependencies and scheduling tasks.

In particular, the edges in a DAG can be interpreted as a partial order on the vertices, where u precedes v , if $(u, v) \in E$. This partial order defines a **topological ordering**.

Some Problems become easier to solve with DAGs. `ShortestPath()` and `LongestPath()` can be solved in linear time with the help of a topological sorting.

Topological ordering

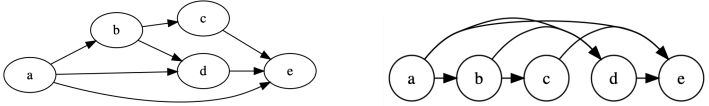
A **topological ordering** (also *topological sorting*) of a directed acyclic graph is a partial order of the vertices, where for each vertex u, v if $(u, v) \in E$, then u comes before v in the order.

The topological ordering of a DAG is **unique**, if and only if a Hamiltonian path exists.

If the vertices of a topological ordering (v_1, \dots, v_k) are consecutively connected by edges, they form a **Hamiltonian path** in the graph.

Finding a topological ordering

A topological ordering can be found either using **Kahns algorithm**¹, or by reversing the post-order of a DFS traversal. This runs in linear time.



On the right a possible topological ordering, obtained from the reverse DFS post-order of the left graph.

Graph traversal

Graph traversal is the process of visiting each vertex in a graph. **Tree traversal** is a special case, applicable to trees.

Breadth first search (BFS)

Breadth first search is a graph traversal method that visits vertices with increasing depth. BFS is implemented iteratively and runs in $\mathcal{O}(|V| + |E|)$.

```
Algorithm 1 Breadth-First Search
1: procedure BFS( $G, s$ )
2:    $Q \leftarrow \emptyset$  ▷ Initialise empty queue
3:   Mark  $s$  as active
4:    $Q.ENQUEUE(s)$ 
5:   while not  $Q.ISEMPY()$  do
6:      $u \leftarrow Q.DEQUEUE()$ 
7:     Mark  $u$  as visited
8:     for  $v \in NEIGHBORS(u, G)$  do
9:       if  $v$  not active and  $v$  not visited then
10:        Mark  $v$  as active
11:         $Q.ENQUEUE(v)$ 
```

Applications of BFS

Due to the nature of BFS, it is suitable for finding the **shortest path** in an unweighted graph. Further applications include cycle detection, checking for bipartite-ness and calculating transitive closure.

Depth first search (DFS)

Depth first search is a graph traversal method that explores as far as possible along each branch before back-tracking. DFS runs in $\mathcal{O}(|V| + |E|)$, since as it visits each vertex and edge exactly once.

```
Algorithm 2 Depth-First Search (Recursive)
1: procedure DFS( $G, s$ )
2:   Mark  $s$  as visited ▷ Pre-number is assigned
3:   for  $v \in NEIGHBORS(s, G)$  do
4:     if  $v$  not visited then
5:       DFS( $G, v$ ) ▷ Post-number is assigned
```

In code, DFS should be implemented iteratively. Iterative implementations are preferable due to risk of stack overflow.

```
Algorithm 3 Iterative Depth-First Search
1: procedure DFS( $G, s$ )
2:    $S \leftarrow \emptyset$  ▷ Initialise empty stack
3:    $S.PUSH(s)$ 
4:   while  $S$  is not empty do
5:      $u \leftarrow S.POP()$ 
6:     if  $u$  not visited then
7:       Mark  $u$  as visited
8:       for  $v \in NEIGHBORS(u, G)$  do
9:          $S.PUSH(v)$ 
```

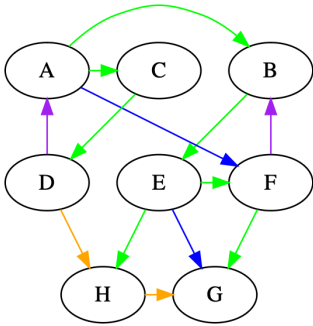
Applications of DFS

- Finding bridges and articulation points
- Topological sorting
- Finding connected components
- Cycle detection

Edge types

To further understand DFS, **pre-** and **post-numbers** are introduced. The **postorder** of a DFS is given by the de-scending order of post-number. With the help of these numbers, one can identify special edges:

- **Tree edge**: In DFS tree
- **Forward edge**: Skips forward in DFS tree
- **Back edge**: Links back to vertex in same tree
- **Cross edge**: Jumps between trees



(a) DFS from vertex a

Tree traversal

- **Pre-order**: Visit the current node, then recur-sively visit the left and right subtrees.
- **In-order**: Recursively visit the left subtree, then visit the current node, then recursively visit the right subtree.
- **Post-order**: Recursively visit the left and right subtrees, then visit the current node.

Each method can be implemented recursively or itera-tively based on the use case.

¹https://en.wikipedia.org/wiki/Topological_sorting#Kahn's_algorithm

Eulerian paths and cycles

An **Eulerian path**/**Eulerian cycle** is a trail/cycle in which every edge of the graph is visited exactly once. A graph is called **Eulerian**, if and only if it contains a Eulerian cycle.

Eulerian paths/cycles are defined for both undirected and directed graphs. In an undirected, connected graph:

- \exists **Eulerian cycle** \iff Every vertex has even degree
- \exists **Eulerian path** \iff 2 or 0 vertices have odd degree

In a directed, weakly connected² graph:

- \exists **Eulerian cycle** $\iff \forall v \in V \text{ inDeg}(v) = \text{outDeg}(v)$
- \exists **Eulerian path** \iff There exist at most 2 vertices such that $|\text{inDeg}(v) - \text{outDeg}(v)| = 1$. For all other vertices $\text{inDeg}(v) = \text{outDeg}(v)$

Constructing Eulerian cycles

Eulerian cycles can be constructed using Hierholzer’s Algorithm in $\mathcal{O}(|E|)$ time.

1. **Iteraitvely calculate cycles:** Start from any vertex v and traverse the graph, following a trail of unexplored edges until returning to v .
2. **Merge cycles:** We merge the new found cycle into the solution.
3. **Repeat** until all cycles are merged.

After executing the algorithm and under the assumption that the graph is Eulerian, we are left with an Eulerian cycle.

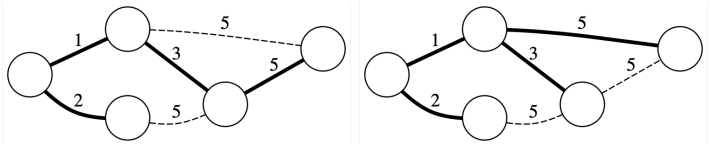
²Note that the existence of a Eulerian cycle in a directed graph implies that the graph is *strongly connected*

Minimum Spanning trees

A **minimum spanning tree (MST)** of a graph is a tree that spans all vertices with minimal total weight, ie. a subset of edges such that the graph is connected and the sum of weights is minimal.

MSTs are particularly useful for modelling network design problems, such as laying telecommunication cables or designing transportation networks.

By definition of a tree, a MST has $|V| - 1$ edges. The MST of a graph is **unique**, if all edges have distinct weight.



Both MSTs have the same total weight

Cycle property: For any cycle in a graph, the maximum weight edge in that cycle cannot be part of any minimum spanning tree.

Contraction: If T is a tree of MST edges, then we can contract T into a single vertex while maintaining the invariant that the MST of the contracted graph plus T gives the MST for the graph before contraction. ie. $MST(G) = T \cup MST(G \text{ contract } T)$.

Kruskal’s Algorithm

Kruskal’s algorithm is an MST algorithm that runs in $\mathcal{O}(|E|\log|E|)$ time.

1.

Sort the edges in ascending order by edge weight
2.

For each edge $(u, v) \in E$, if u and v are in different components, add the edge to the MST and merge the components.

Algorithm 4 Kruskal’s algorithm

1: **procedure** KRUSKAL(G)

2: $T \leftarrow \emptyset$

3: $E \leftarrow$ edges of G sorted by weight

4: **for** $(u, v) \in E$ **do**

5: **if** u, v in different components **then**

6: $T \leftarrow T \cup \{(u, v)\}$

Kruskal’s algorithms is very **simple** to implement. For an optimal runtime, it requires a **Union-Find** data structure to quickly find and merge components.

Prim’s Algorithm

Prim’s algorithm runs in $\mathcal{O}((|V| + |E|)\log|V|)$. It is the most commonly used algorithm and is very similar to **Dijkstra’s algorithm**.

Prim’s builds a spanning tree starting from a specific vertex. In each step, it chooses the minimum edge incident to the visited vertices which connects to an unvisited vertex, and adds it to the MST.

Algorithm 5 Prim’s algorithm

1: **procedure** PRIM(G, s)

2: $T \leftarrow \emptyset$

3: $S \leftarrow s$ ▷ Component of s in T

4: **while** T not spanning **do**

5: $(u, v) \leftarrow$ min edge to $S, u \in S, v \notin S$

6: $T \leftarrow T \cup \{(u, v)\}$ ▷ Add (u, v) to tree

7: $S \leftarrow S \cup \{v\}$

For optimal runtime, Prim’s algorithm requires a **min-heap**, so that the minimum edge to S can be quickly found.

Boruvka’s Algorithm

Boruvka’s algorithm was the first MST algorithm to be discovered. It runs in $\mathcal{O}((|V| + |E|)\log|V|)$. Boruvka’s algorithm is particularly fast on sparse graphs and plays a key role in randomized MST-algorithms³.

Boruvka’s algorithm maintains a forest of spanning trees, where each component is initially a single vertex. In each iteration, the algorithm adds the lightest edge incident to each component to the forest. After the last iteration, the forest is a MST.

Algorithm 6 Boruvka’s algorithm

1: **procedure** BORUVKA(G)

2: $F \leftarrow \emptyset$ ▷ Initialise forest

3: **while** F not spanning **do**

4: $(S_1, \dots, S_k) \leftarrow$ components of F

5: $(e_1, \dots, e_k) \leftarrow$ minimal edges to S_1, \dots, S_k

6: $F \leftarrow F \cup \{e_1, \dots, e_k\}$

To achieve the optimal runtime, Boruvka’s algorithm needs an efficient way to store the forest and determine the lightest edges incident to the components. This can be done using a **Union-find** data structure and a **min-heap**.

³See https://en.wikipedia.org/wiki/Expected_linear_time_MST_algorithm