Algorithms, Datastructures & Probability

Cheatsheet for HS-2022 and FS-2023

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:

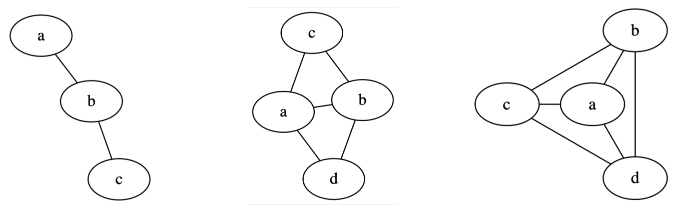
$$|E| \leq \binom{n}{2} = \frac{(n-1) \cdot n}{2} \leq \mathcal{O}(|V|^2)$$

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 (directed) 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.



(a) Sparse graph (b) Dense graph (c) Complete graph

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

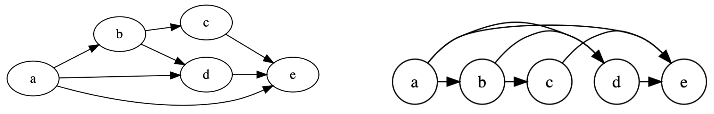
A `topological ordering` (also *topological sorting*) of a directed acyclic graph is a partial order of the vertices, where for each vertex u, v if $(u, v) \in E$, then u comes before v in the order.

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

If the vertices of a topological ordering $(v_1, ..., v_k)$ 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

```
procedure BFS($G, s$)
    $Q \leftarrow \emptyset$                          ▷ Initialise empty queue
    Mark $s$ as active
    $Q$.ENQUEUE($s$)
    while not $Q$.ISEMPTY() do
        $u \leftarrow Q$.DEQUEUE()
        Mark $u$ as visited
        for $v \in$ NEIGHBORS($u, G$) do
            if $v$ not active and $v$ not visited then
                Mark $v$ as active
                $Q$.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 bipartiteness 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 backtracking. DFS runs in $\mathcal{O}(|V|+|E|)$, since as it visits each vertex and edge exactly once.

Algorithm 2 Depth-First Search (Recursive)

```
procedure DFS($G, s$)
    Mark $s$ as visited                        ▷ Pre-number is assigned
    for $v \in$ NEIGHBORS($s, G$) do
        if $v$ not visited then
            DFS($G, v$)
                                               ▷ Post-number is assigned
```

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

Algorithm 3 Iterative Depth-First Search

```
procedure DFS($G, s$)
    $S \leftarrow \emptyset$                          ▷ Initialise empty stack
    $S$.PUSH($s$)
    while $S$ is not empty do
        $u \leftarrow S$.POP()
        if $u$ not visited then
            Mark $u$ as visited
            for $v \in$ NEIGHBORS($u, G$) do
                $S$.PUSH($v$)
```

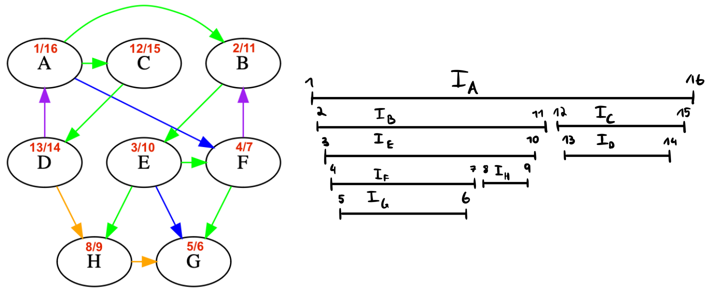
Applications of DFS

- 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 descending 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



Graph and intervals for DFS from vertex A

Tree traversal

- **Pre-order**: Visit the current node, then recursively 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 iteratively based on the use case.

[1] `https://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 connected[2] graph:

$\exists$ Eulerian cycle $\iff \forall v \in V$ inDeg(v) = outDeg(v)

$\exists$ Eulerian path $\iff$ There exist at most 2 vertices such that $|$ inDeg(v) $-$ outDeg(v) $| = 1$. For all other vertices inDeg(v) = 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.

Hamiltonian paths and cycles

A Hamiltonian path is a path through all vertices of a graph. A graph is called Hamiltonian if there exists a Hamiltonian cycle.
Common examples of Hamiltonian graphs include:

- Icosahedral graph
- Complete graph with $n \geq 3$
- A grid with $n \times m$ vertices where nm is even.

A Peterson graph is *not* Hamiltonian.
In general it holds that every Hamiltonian graph is 2-connected.

Determining Hamiltonicity

Finding out whether a graph is Hamiltonian is very difficult. In fact, for general graphs this problem is **NP-Complete**. However, for some graphs it is possible to make statements.

Dirac's theorem

A simple graph with $n \geq 3$ vertices is Hamiltonian if every vertex has degree $\frac{n}{2}$ or greater.

Dirac is very helpful in many exercises. Further, it also implies with the pigeonhole principle that almost complete graphs with $|E| \geq \binom{n}{2} - \lfloor \frac{n}{2} \rfloor + 1$ contain a hamiltonian cycle.

Ore's theorem

A simple graph with $n \geq 3$ vertices is Hamiltonian if for every pair of non-adjacent vertices, the sum of their degrees is n or greater.

Calculating Hamiltonicity

A trivial brute-forces solution requires $\mathcal{O}(n!)$ time. However, it is possible to find a much better algorithm which runs in $\mathcal{O}(2^n n^{2.81} \log n)$ and requires $\mathcal{O}(n^2)$ space, making it viable for small inputs.

Algorithm 4 Counting Hamiltonian Cycles

```
$W_S :=$ count of paths of length $n$ from $s = 1$ to $s = 1$
that don't visit a vertex in $S$

procedure COUNT($G = ([n], E)$)
    $s \leftarrow 1$                                  ▷ Chosen arbitrarily
    $Z \leftarrow |W_\emptyset|$
    for $S \subseteq [n]$ with $s \notin S$ and $S \neq \emptyset$ do
        Calculate $|W_S|$ with Adj.matrix of $G[V \setminus S]$
        $Z \leftarrow Z + (-1)^{|S|} |W_S|$
    return $\frac{Z}{2}$                              ▷ Hamiltonian cycles in $G$
```

Graph colouring

A graph colouring in a graph $G = (V, E)$ with k colours is a mapping $c : V \rightarrow [\mathrm{k}]$ such that $c(u) \neq c(v)$ for all $(u, v) \in E$.

Graph colouring is an important problem in graph theory, useful for scheduling timetables, register allocation and even in solving Sudoko puzzles.

The chromatic number $\chi(G)$ of a graph is the minimum number of colours needed for a graph colouring.

$$\chi(G) \leq k \iff G \text{ is } k\text{-partite}$$

For general $k \geq 3$, determining whether a colouring exists is **NP-Complete**. There are however special cases. Most notably:

- **Bipartite graphs** can be coloured with 2 colours. A colouring can be determined in linear time using DFS or BFS.
- **Planar graphs** can be coloured using at most 4 colours. A colouring can be found in $\mathcal{O}(n^2)$.
- **Trees** can be coloured using at most 2 colours in linear time.
- **Graphs** withs $\chi(G) = 3$ can be coloured with $\mathcal{O}(\sqrt{|V|})$ colours in $\mathcal{O}(|V| + |E|)$ time.

More general graphs require exponential time for determining a colouring. Approximations are *not* very good, since for every $\varepsilon \geq 0$ it is **NP-hard** to find an $n^{1-\varepsilon}$ approximation.

Algorithm 5 Greedy Graph Colouring

```
procedure GREEDYCOLOURING($G$)
    choose $v_1, ..., v_n$ as an arbitrary vertex sequence
    $c[v_1] \leftarrow 1$
    for $i = 2..n$ do
        $c[v_i] \leftarrow$ smallest colour not adjacent to $v_i$
```

For every vertex sequence $V = \{v_1, ..., v_n\}$ the greedy algorithm above needs at most $\Delta(G) + 1$ colours, where $\Delta(G)$ is the maximum degree in G. However, there exists a vertex sequence such that the algorithm finds a solution with $\chi(G)$ colours.

Brook's theorem

If $G \neq K_n$, $G \neq C_{2n+1}$ and G connected, then G is colourable with $\Delta(G)$ colours in $\mathcal{O}(|E|)$ time.

Travelling Salesman Problem (TSP)

The Travelling Salesman Problem (TSP) is an **NP-Hard**[3] combinatorial optimisation problem defined on a weighted, undirected graph, with the objective of finding the shortest Hamiltonian cycle.

$$\min_{H:\text{ Ham. cycle}} \sum_{e \in E(H)} l(e)$$

2-Approximation

Let $G = (V, E)$ be a complete graph and $l(e)$ a metric weight function. We can give an approximation that is at most 2 times worse than the optimal Hamiltonian cycle in $\mathcal{O}(|E| \log |V|)$ time with the following algorithm:

1. Calculate a minimum spanning tree T of G
2. Double all edges of T
3. Compute a Eulerian tour E in T
4. Truncate E: skip vertices used more than once and connect them via. a different edge. A shortcut is always possible because of the triangle inequality.

This approximation guarantees $\text{cost}(C) \leq 2 \cdot \text{cost}(OPT)$.

Christofide's algorithm

Christofide's algorithm is an approximation scheme for metric TSP. Let $G = (V, E)$ be a complete graph and $l(e)$ a metric weight function. Christofide's algorithm yields a Hamiltonian cycle at most 1.5 times worse than the optimal Hamiltonian cycle.

1. Calculate a minimum spanning tree T of G
   $l(T) \leq opt(k_n, l)$
2. Find a minimal matching M for all vertices with uneven degree.
   $l(M) \leq \frac{1}{2} opt(k_n, l)$
3. Find Eulerian tour W
   $l(W) = l(T) + l(M) \leq \frac{3}{2} opt(K_n, l)$
4. Traverse W with shortcuts, such that every vertex is visited only once $\rightsquigarrow$ Hamiltonian cycle C
   $l(C) \leq l(W) = l(T) + l(M) \leq \frac{3}{2} opt(K_n, l)$

This approximation guarantees $\text{cost}(C) \leq \frac{3}{2} \cdot \text{cost}(OPT)$.

Human approximation

Humans can give approximations for TSP in near-linear-time, varying from 1% less efficient for graphs with 10-20 nodes to 11% less efficient for those with 120 nodes. Research is still being done, but results suggest that computer performance may be improved through understanding the heuristics humans use.

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

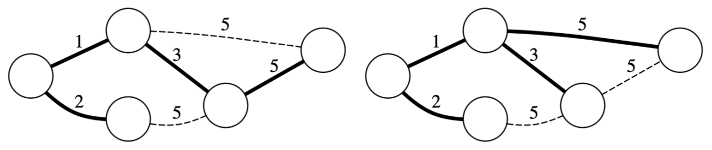
[3] The decision version of TSP is **NP-complete**

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 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 6 Kruskal's algorithm

```
procedure KRUSKAL($G$)
    $T \leftarrow \emptyset$
    $E \leftarrow$ edges of $G$ sorted by weight
    for $(u, v) \in E$ do
        if $u, v$ in different components then
            $T \leftarrow T \cup \{(u, v)\}$
```

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 7 Prim's algorithm

```
procedure PRIM($G, s$)
    $T \leftarrow \emptyset$
    $S \leftarrow s$                                  ▷ Component of $s$ in $T$
    while $T$ not spanning do
        $(u, v) \leftarrow$ min edge to $S$, $u \in S, v \notin S$
        $T \leftarrow T \cup \{(u, v)\}$              ▷ Add $(u, v)$ to tree
        $S \leftarrow S \cup \{v\}$
```

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-algorithms[4].

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 8 Boruvka's algorithm

```
procedure BORUVKA($G$)
    $F \leftarrow \emptyset$                           ▷ Initialise forest
    while $F$ not spanning do
        $(S_1, ..., S_k) \leftarrow$ components of $F$
        $(e_1, ..., e_k) \leftarrow$ minimal edges to $S_1, ..., S_k$
        $F \leftarrow F \cup \{e_1, ..., e_k\}$
```

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`.

[4] See `https://en.wikipedia.org/wiki/Expected_linear_time_MST_algorithm`