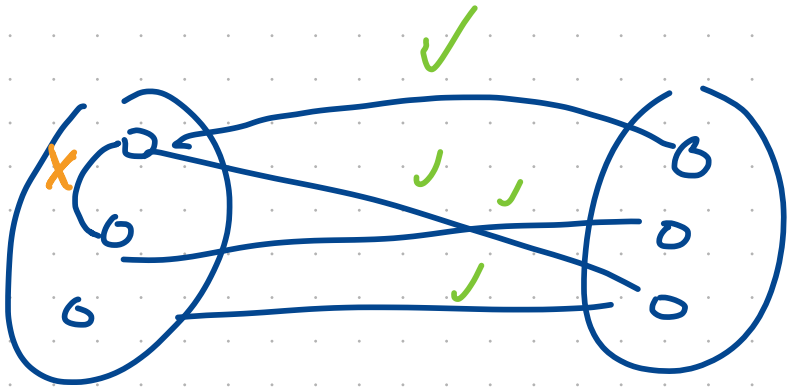
GRAPH THEORY

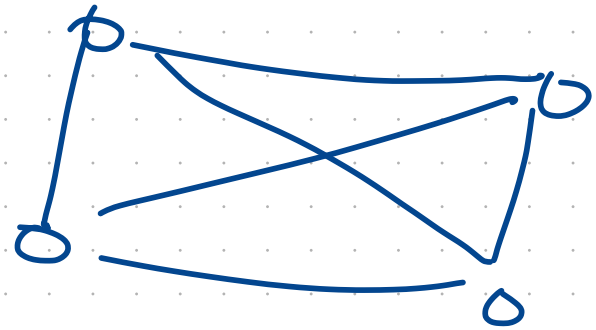
$$G = (V, E)$$

nodes — edges

- Directed
- Undirected
- Bipartite



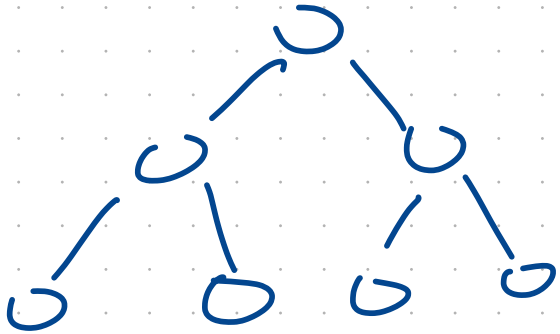
- Complete graph

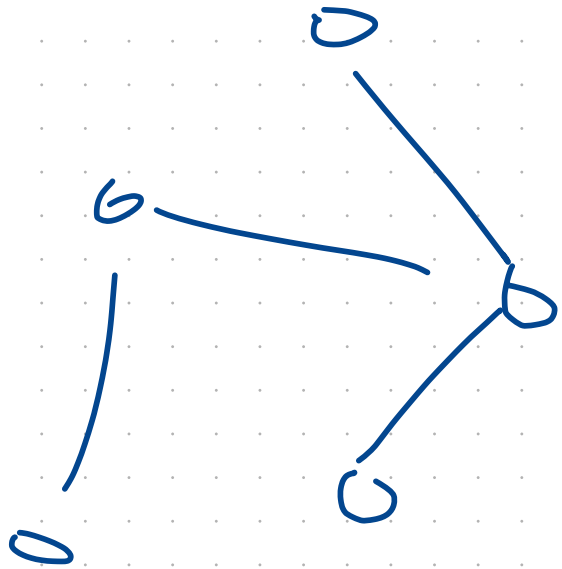


TREES

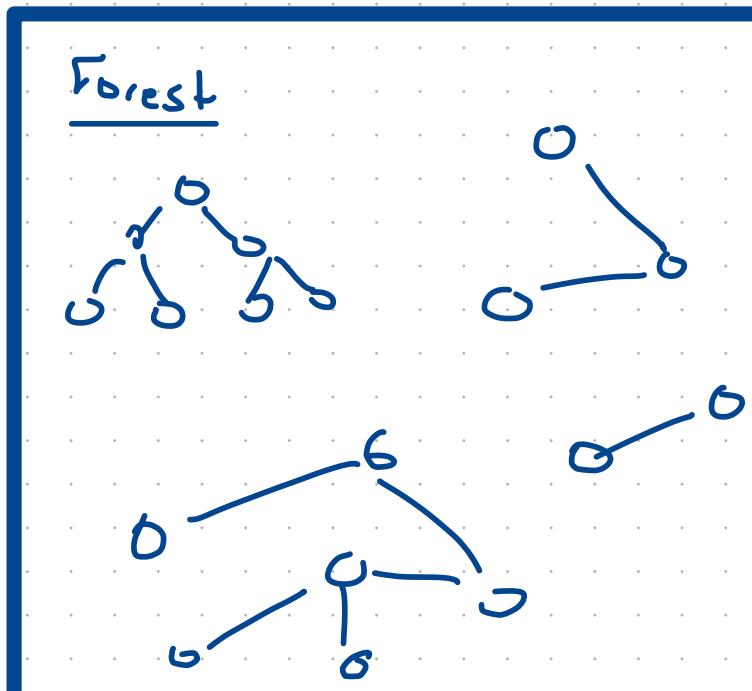
(b) G ist zusammenhängend und kreisfrei,
(c) G ist zusammenhängend und $|E| = |V| - 1$,
(d) G ist kreisfrei und $|E| = |V| - 1$,
(e) für alle $x, y \in V$ gilt: G enthält genau einen x-y-Pfad.

Binary Search Trees + AVL Trees are Trees in the graph theory sense.





Forest



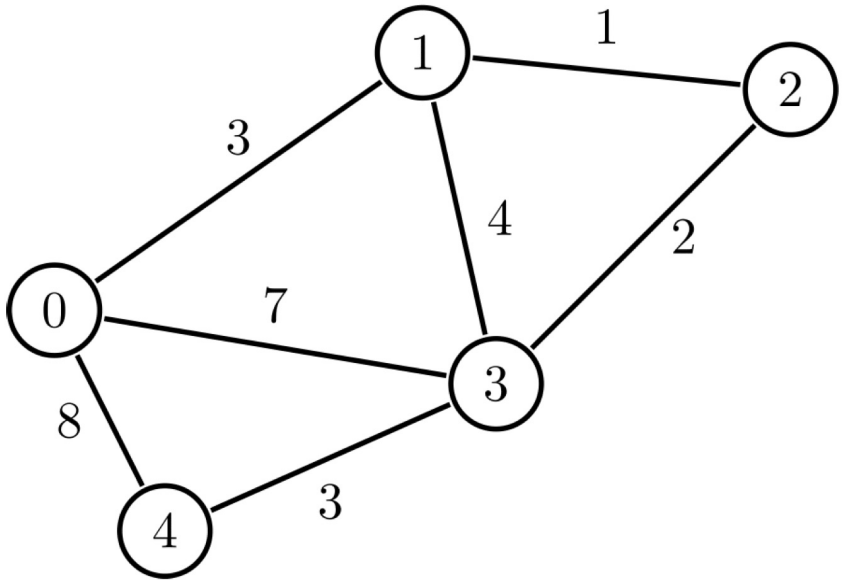
DATA STRUCTURES

List of edges

from to weight

{(0,1,3), (1,2,4),

(2,3,2) ... }



Adj. Matrix

$$\begin{bmatrix} 0 & 3 & 0 & 7 & 8 \\ 3 & 0 & 1 & 4 & 0 \\ 0 & 1 & 0 & 2 & 0 \\ 7 & 4 & 2 & 0 & 3 \\ 8 & 0 & 0 & 3 & 0 \end{bmatrix}$$

Adj. List

dest cost

0 → (1,3), (3,7), (4,8)

1 → (0,3), (2,1), (3,4)

2 → (1,1), (3,2)

3 → (0,7), (1,4), (2,2), (4,3)

4 → (0,8), (3,3)

DATA STRUCTURES

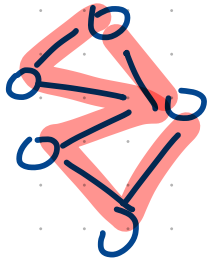
Adj. Matrix

- **Memory space:** $\mathcal{O}(|V|^2)$
- **Add vertex:** $\mathcal{O}(|V|^2)$
- **Remove vertex:** $\mathcal{O}(|V|^2)$
- **Add edge:** $\mathcal{O}(1)$
- **Remove edge:** $\mathcal{O}(1)$
- **Are u and v adjacent?** $\mathcal{O}(1)$
- **Given a node v, find $deg(v)$:** $\mathcal{O}(|V|)$
- **Given a node v, give an arbitrary neighbour of v:** $\mathcal{O}(|V|)$
- **Given a node v, find all incidents edges to v:** $\mathcal{O}(|V|)$

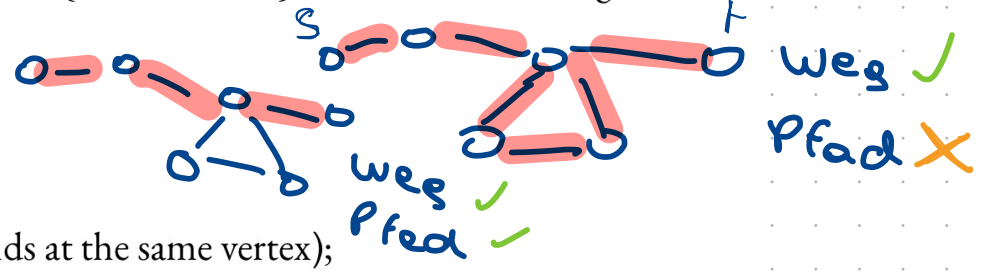
Adj. List

- **Memory space:** $\mathcal{O}(|V| + |E|)$
- **Add vertex:** $\mathcal{O}(1)$
- **Remove vertex:** $\mathcal{O}(|E|)$.
- **Add edge:** $\mathcal{O}(1)$
- **Remove edge:** $\mathcal{O}(|V|)$
- **Are u and v adjacent?** $\mathcal{O}(\min(deg(u), deg(v)))$
- **Given a node v, find $deg(v)$:** $\mathcal{O}(deg(v))$ [O(1)]
- **Given a node v, give an arbitrary neighbour of v:** $\mathcal{O}(1)$
- **Given a node v, find all incidents edges to v:** $\mathcal{O}(deg(v))$

Zyklus ✓
Kreis ✗



- A **walk** (Weg) iff there are edges between v_i and v_{i+1} for all $i \in \{1, \ldots, k-1\}$; in this case, the length of the walk is $k-1$;
- A **path** (Pfad) iff it is a walk whose vertices are all distinct;
- A **tour** (Reise) iff it is a walk whose edges are all distinct;
- A **circuit** (Zyklus) iff it is a walk with $v_1 = v_k$ (starts and ends at the same vertex);
- A **cycle** (Kreis) iff it is a circuit for which no vertex, except the first/last one, is visited more than once;
- A **loop** (Schleife) iff it is a cycle $\langle v_i, v_i \rangle$ of length 1.



When the considered objects cover all vertices or all objects, this gives rise to the following definitions:

Zyklus ✓
Kreis ✓

- A **Eulerian walk** (Eulerweg) is a walk that visits all edges exactly once, i.e. a walk of length $m = |E|$.
- A **Eulerian circuit** (Eulerkreis[1]) is a circuit that visits all edges exactly once, i.e. a walk of length $m = |E|$.
- A **Hamiltonian path** (Hamiltonpfad) is a path that visits all vertices exactly one, i.e. a path of length $n-1$.
- A **Hamiltonian circuit** (Hamiltonkreis[2]) is a cycle that visits all vertices, i.e. a cycle of length n.

P vs NP

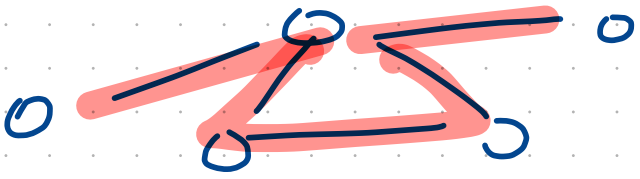
Degree

Neighbourhood

$$\sum_v \deg(v) = 2E$$

Eulerian Walk $\Leftrightarrow$ ∃ at most 2 nodes with odd degree

Eulerian Circuit $\Leftrightarrow$ All nodes have even degree



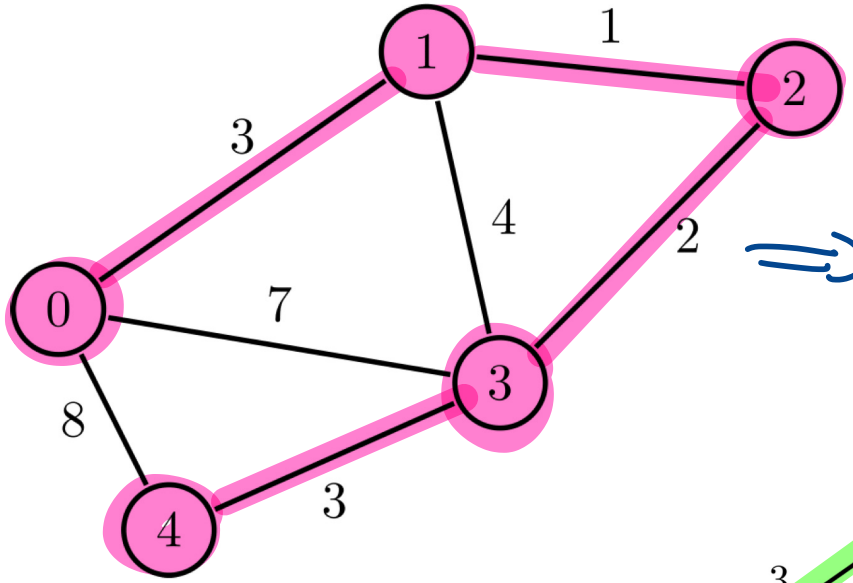
Eulerian walk but not eulerian circuit

DFS + BFS

Complexity: O(|V|+|E|)

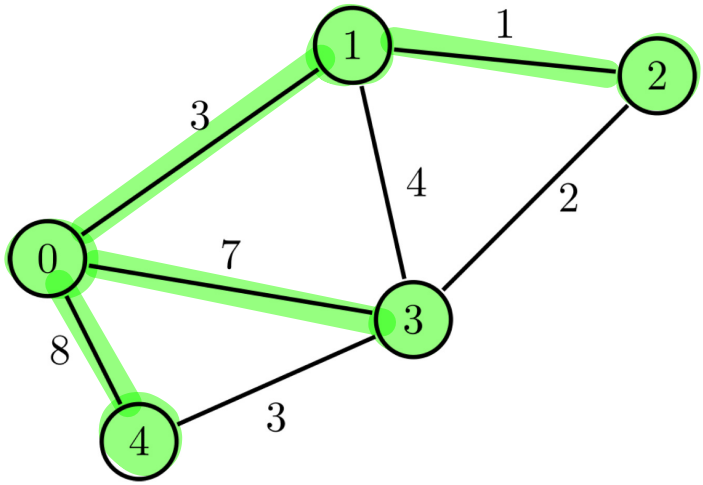
DFS: Stack; topological sorting

BFS: Queue; shortest path



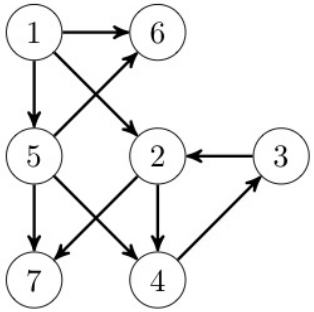
DFS

⟹ 0, 1, 2, 3, 4

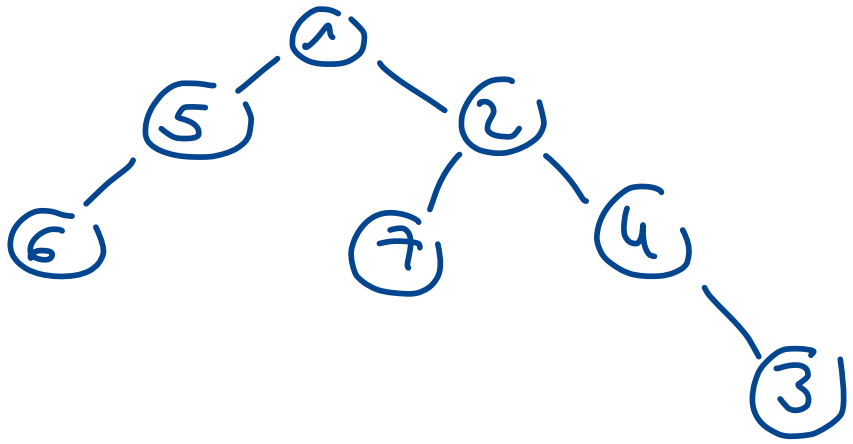


BFS

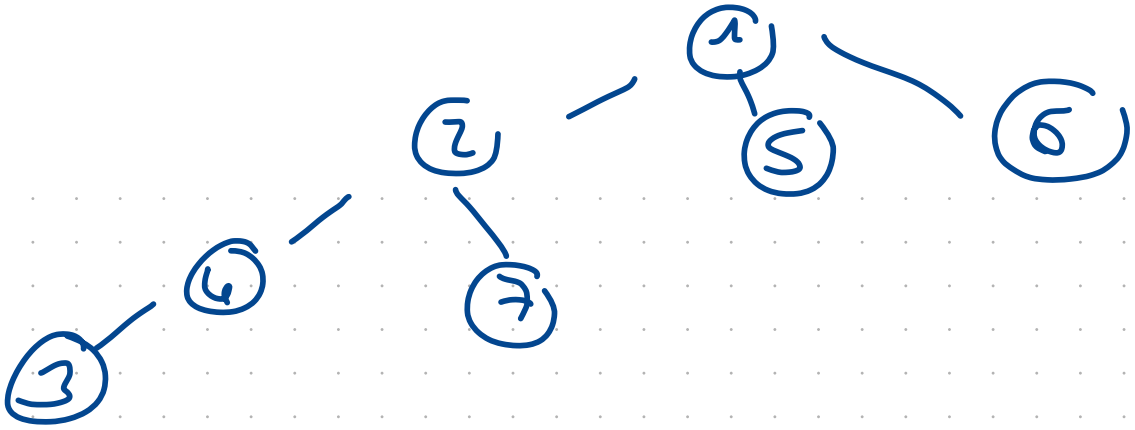
0, 1, 3, 4, 2 ⟸



i) Draw the depth-first tree resulting from a depth-first search starting from vertex 1. Process the neighbors of a vertex in increasing order.

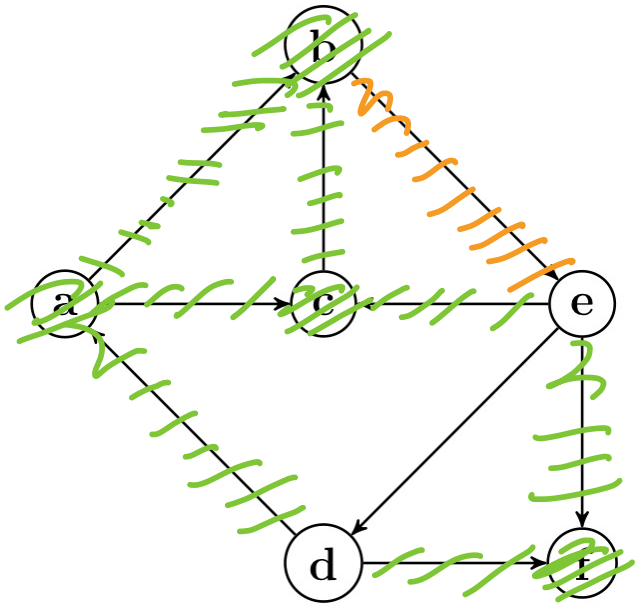


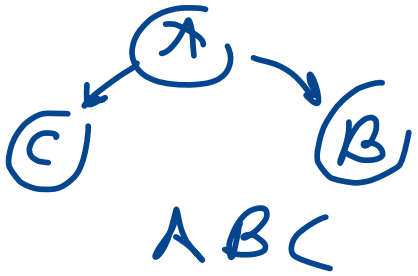
ii) Draw the breadth-first tree resulting from a breadth-first search starting from vertex 1. Process the neighbors of a vertex in increasing order.

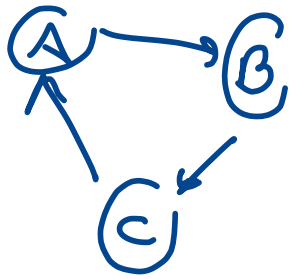


TOPOLOGICAL SORTING

f) *Topological sorting:* Consider the following directed graph G:



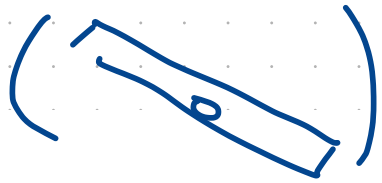




i) Remove the smallest possible number of edges from G such that a topological ordering of its vertices exists. (b, e)

ii) Compute a topological ordering of the vertices of your modified graph.

e, d, f, a, c, b



$= \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$

(a) True or false?

i. In an undirected graph, a tight bound for the number of edges is n^2. F

ii. In an undirected graph, a tight asymptotic bound for the number of edges is $\Theta\left(n^2\right)$. F

iii. If the maximum degree of any node in an undirected graph G is 1, then this graph is a tree. F

iv. The complexity of computing the out-degree of a vertex v in an adjacency matrix is $\Theta\left(\deg v\right)$. F

v. The complexity of computing the sum of all out-degrees of vertices in an adjacency matrix is $\Theta\left(n^2\right)$. T

vi. The list of vertices accessible from a vertex v in a graph G can be computed by using at most $n-1$ multiplications of $n \times n$ matrices. T

vii. A connected graph is Eulerian (i.e. contains a Eulerian circuit) iff all its vertices have even degree. T

viii. A graph contains a Eulerian walk iff all its vertices have even degree. F

ix. Testing if a graph is Eulerian is NP-complete. F

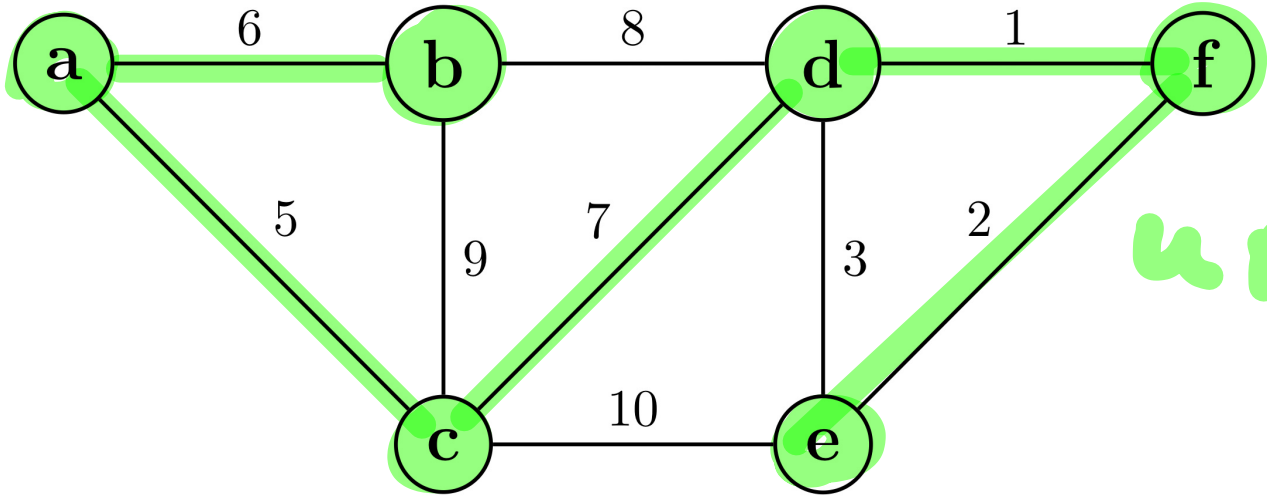
x. The post-order of BFS always gives a valid topological ordering. F

A

$(A^n)_{i:}$ are the nodes reachable from node i in n steps

Claim	**true**	**false**
The topological ordering of a directed acyclic graph is unique.	☐	☒
For all $n \in \mathbb{N}$, there exists a directed acyclic graph on n vertices with $\binom{n}{2}$ edges.	☒	☐
Let $v \in V$ be a vertex of an undirected graph $G = (V, E)$ with adjacency matrix A. It takes time $\Theta(1 + \deg(v))$ to compute $\deg(v)$ from A.	☐	☒
If every vertex of an undirected graph G has even degree, then G has an Eulerian walk.	☐	☒
In order to run Dijkstra's algorithm on a directed graph G, you first need to have a topological ordering of G.	☐	☐

MST



KRUSKAL

Prim

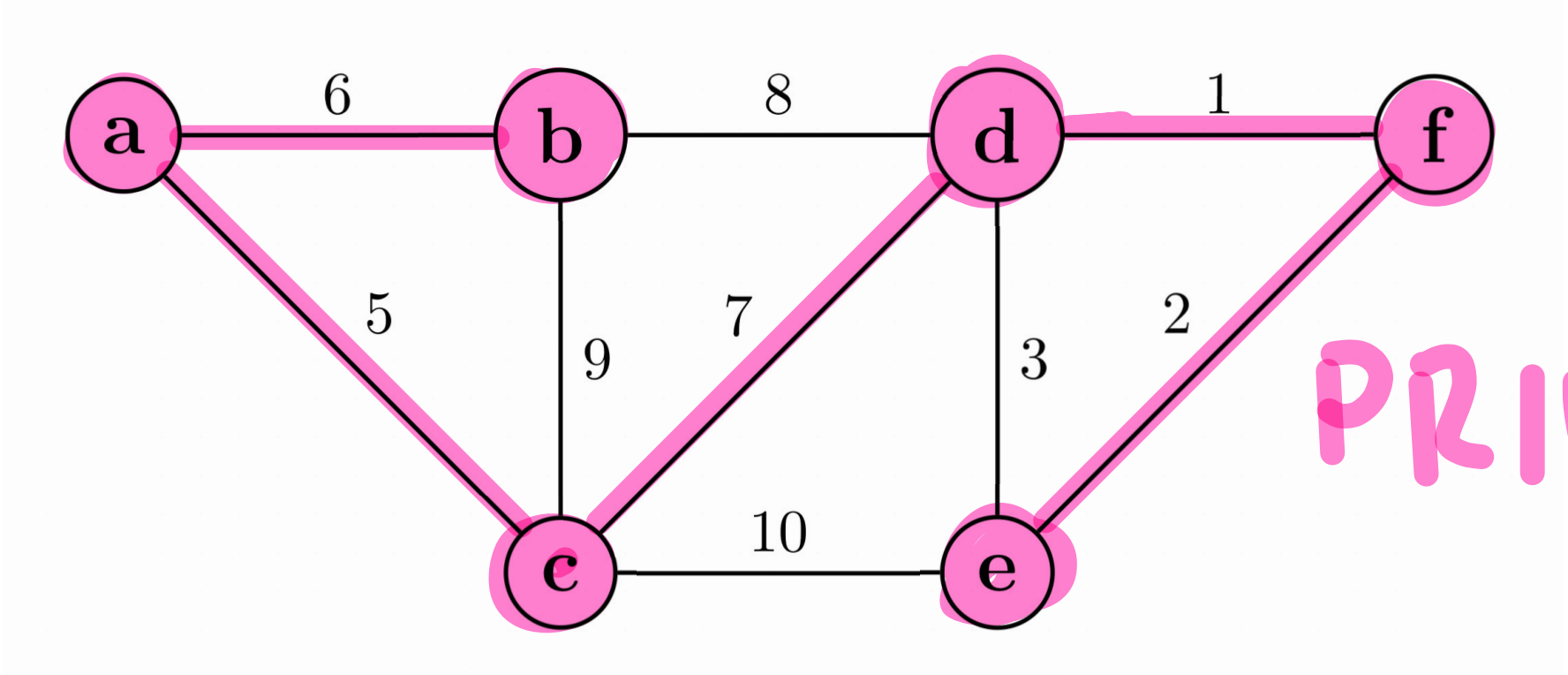
Kruskal ⇒ Union-Find

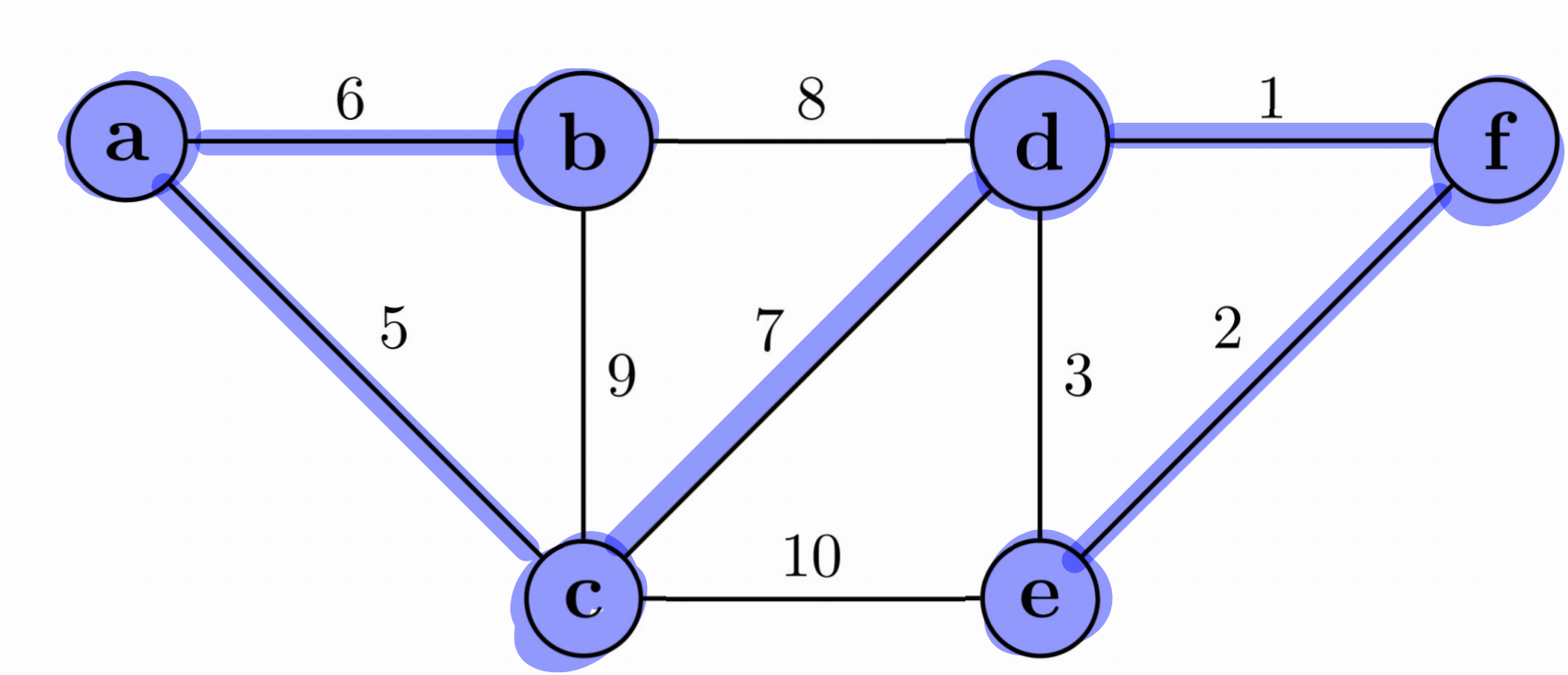
Boruvka

MAKE-SET

UNION

≈FIND-SET


a
b
c
d
e
f
6
8
1
5
9
7
3
2
10
PRIM




```
1) Input is a connected, weighted and un-directed graph.
2) Initialize all vertices as individual components (or sets).
3) Initialize MST as empty.
4) While there are more than one components, do following
   for each component.
      a)  Find the closest weight edge that connects this
          component to any other component.
      b)  Add this closest edge to MST if not already added.
5) Return MST.
```

Theory Task T4.

/ 12 P

/ 3 P

a) Consider the following problem. The Swiss government is negotiating a deal with Elon Musk to build a tunnel system between all major Swiss cities. They put their faith into you and consult you. They present you with a map of Switzerland. For each pair of cities it depicts the cost of building a bidirectional tunnel between them. The Swiss government asks you to determine the cheapest possible tunnel system such that every city is reachable from every other city using the tunnel network (possibly by a tour that visits other cities on the way).

i) Model the problem as a graph problem. Describe the set of vertices, the set of edges and the weights in words. What is the corresponding graph problem?

✓ cities

E (src, dest, w)

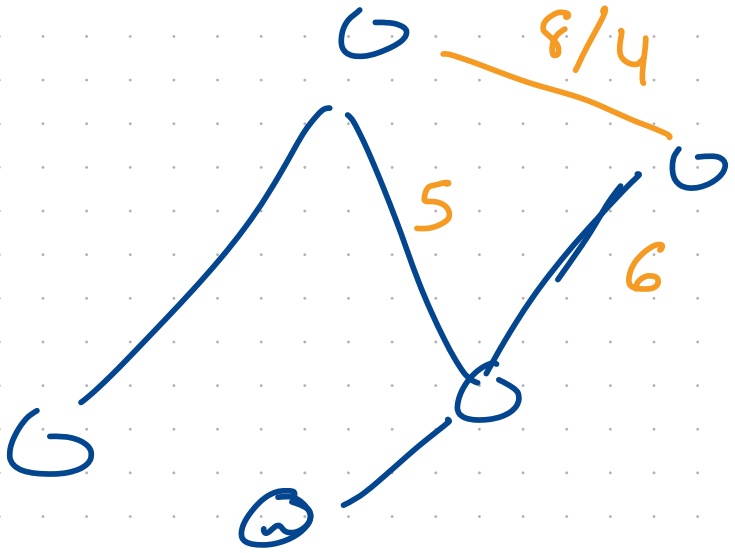
ii) Use an algorithm from the lecture to solve the graph problem. State the name of the algorithm and its running time in terms of $|V|$ and $|E|$ in Θ-notation.

kruskal (...)

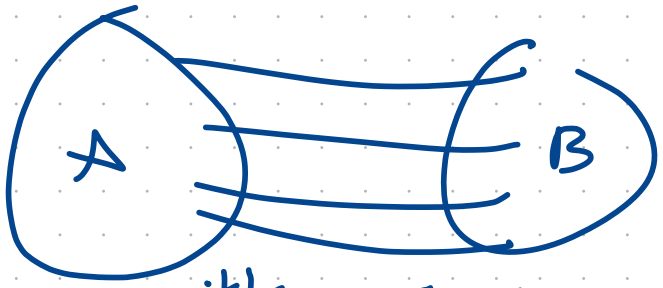
b) Now, the Swiss tunneling society contacts the government and proposes to build the tunnel between Basel and Geneva for half of Musk's cost. Thus, the government contacts you again. They want you to solve the following problem: Given the solution of the old problem in a) and an edge for which the cost is divided by two, design an algorithm that updates the solution such that the new edge cost is taken into account. *In order to achieve full points, your algorithm must run in time* $\mathcal{O}(|V|)$.

Hint: You are only allowed to use the *solution* from a), i.e. the set of tunnels in the chosen tunnel system. You are not allowed to use any intermediate computation results from your algorithm in a).

i) Describe your algorithm (for example, via pseudocode). A high-level description is enough.



BLUE RULE:

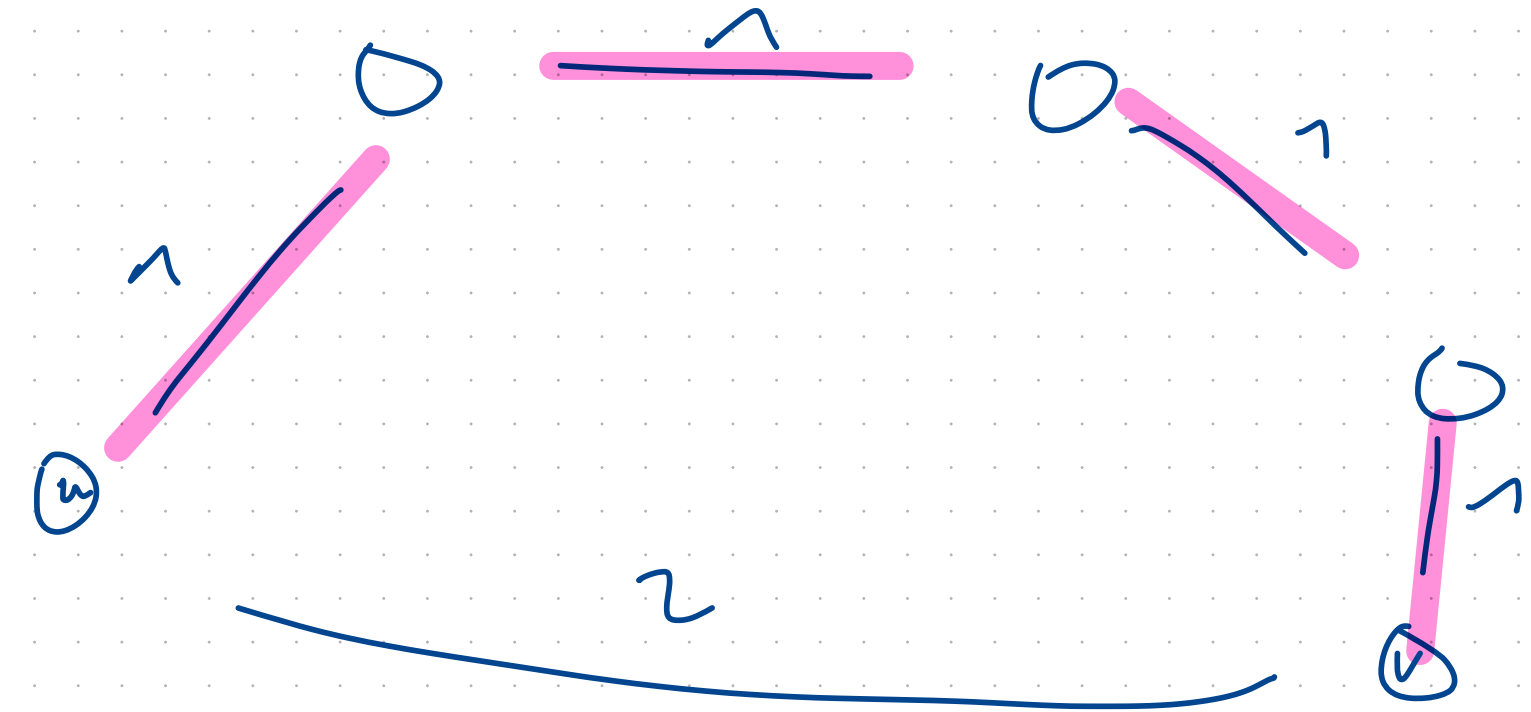


it's safe to add the edge with minimum weight (the edge goes from A to B)

RED RULE:

given a cycle, remove the edge with maximum weight

Prove or disprove: For all vertices u, v of a graph G, the only path between u and v in an MST T of G is a shortest path between u and v in G.



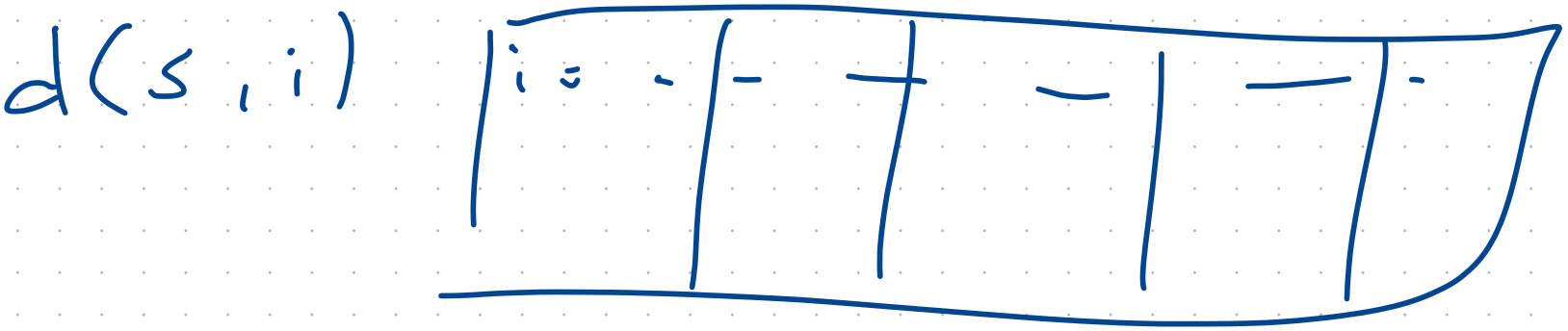
SHORTEST PATHS

	Situation	Algorithm	Runtime
1	One-to-all, all edges with the same weight	Modified BFS	$\mathcal{O}(\|V\| + \|E\|)$
2	One-to-all, directed acyclic graph with arbitrary weights	DP and TopoSort	$\mathcal{O}(\|V\| + \|E\|)$
3	One-to-all, only with non-negative weights	Dijkstra	$\mathcal{O}(\|V\| \log \|V\| + \|E\|)$ Fibonacci $O((\|V\| + \|E\|) \log \|V\|)$ Binary
4	One-to-all, also with negative weights	Bellman-Ford	$\mathcal{O}(\|V\|\|E\|)$
5	All-to-all	Floyd-Warshall	$\mathcal{O}(\|V\|^3)$
6	All-to-all	Johnson	$\mathcal{O}(\|V\|\|E\| + \|V\|^2 \log \|V\|)$

At this point we mention a useful property of the shortest path problem that one can easily show by contradiction. Shortest paths satisfy the following optimality principle: if $(v_0, v_1, \ldots, v_{l-1}, v_l)$ is a shortest path in G, then also $(v_0, v_1, \ldots, v_{l-1})$ is a shortest path in G. We know that if a problem exhibits an optimality principle, then it can be solved with dynamic programming. In facts one can write the following recurrence:

$$d(s,v) = \min_{(u,v)\in E} d(s,u) + c(u,v)$$

ORDER?



1

Algorithm 24: SP-BFS(G, s)

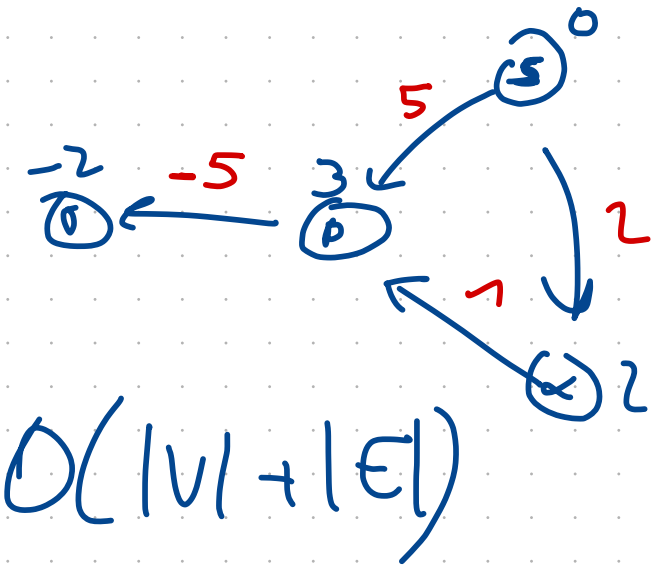
```
d[s] ← 0;
d[v] ← ∞ for v ∈ V \ {s};
Q ← ∅;
ENQUEUE(s,Q);
while Q ≠ ∅ do
    u ←DEQUEUE(Q);
    for each (u, v) ∈ E do
    if d[v] = ∞ then
        d[v] ← d[u] + c(u, v);
        ENQUEUE(v,Q);
```

s, α, β, γ

2

Algorithm 25: SP-DAG(G, s)

```
d[s] ← 0;
d[v] ← ∞ for v ∈ V \ {s};
for v ∈ V in topological order do
    for all (v, w) ∈ E if d[w] > d[v] + c(v, w) then
        d[w] = d[v] + c(v, w);
```



$O(|V| + |E|)$

(3) DIJKSTRA

S (where $d[s,i]$ is correct $\forall i \in S$)

At each iteration, we increase the cardinality of S by one

$d[s] = 0$, $d[v] = \infty \ \forall v \in V \setminus \{s\}$

$P \leftarrow \emptyset$, P.insert(s, 0)

while (P is not empty)

$u \leftarrow$ P.extract_min, $S + \{u\}$

for all $(u,v) \in E$

$d[v] = \min(d[v], d[u] + c(u,v))$ (*)

(4) if we changed $d[v]$ in (*), either insert in P or update the priority

Algorithm 27: Bellman-Ford(G, s)

```
for each v ∈ V \ {s} do d[v] = ∞;
d[s] = 0;
for i ← 1, 2, ..., |V| − 1 do
    for each (u, v) ∈ E do
        if d[v] > d[u] + c(u, v) then
            d[v] ← d[u] + c(u, v)
for each (u, v) ∈ E do
if d[v] > d[u] + c(u, v) then
    return Negative cycle;
```

$O(|V||E|)$

5

Algorithm 28: Floyd Warshall(G)

```
for 1 ≤ u ≤ |V| and 1 ≤ v ≤ |V| do
    d(u,v,0) = ∞;
for u ∈ V do
    d(u,u,0) = 0 ;
for (u,v) ∈ E do
    d(u,v,0) = c(u,v);
for i = 1,...,|V| do
    for u = 1,...,|V| do
        for v = 1,...,|V| do
            d(u,v,i) = min(d(u,v,i−1), d(u,i,i−1) + d(i,v,i−1));
return d;
```

d(u,v,i): SP from u to v using nodes 1...i as intermediate nodes

$O(|V|^3)$

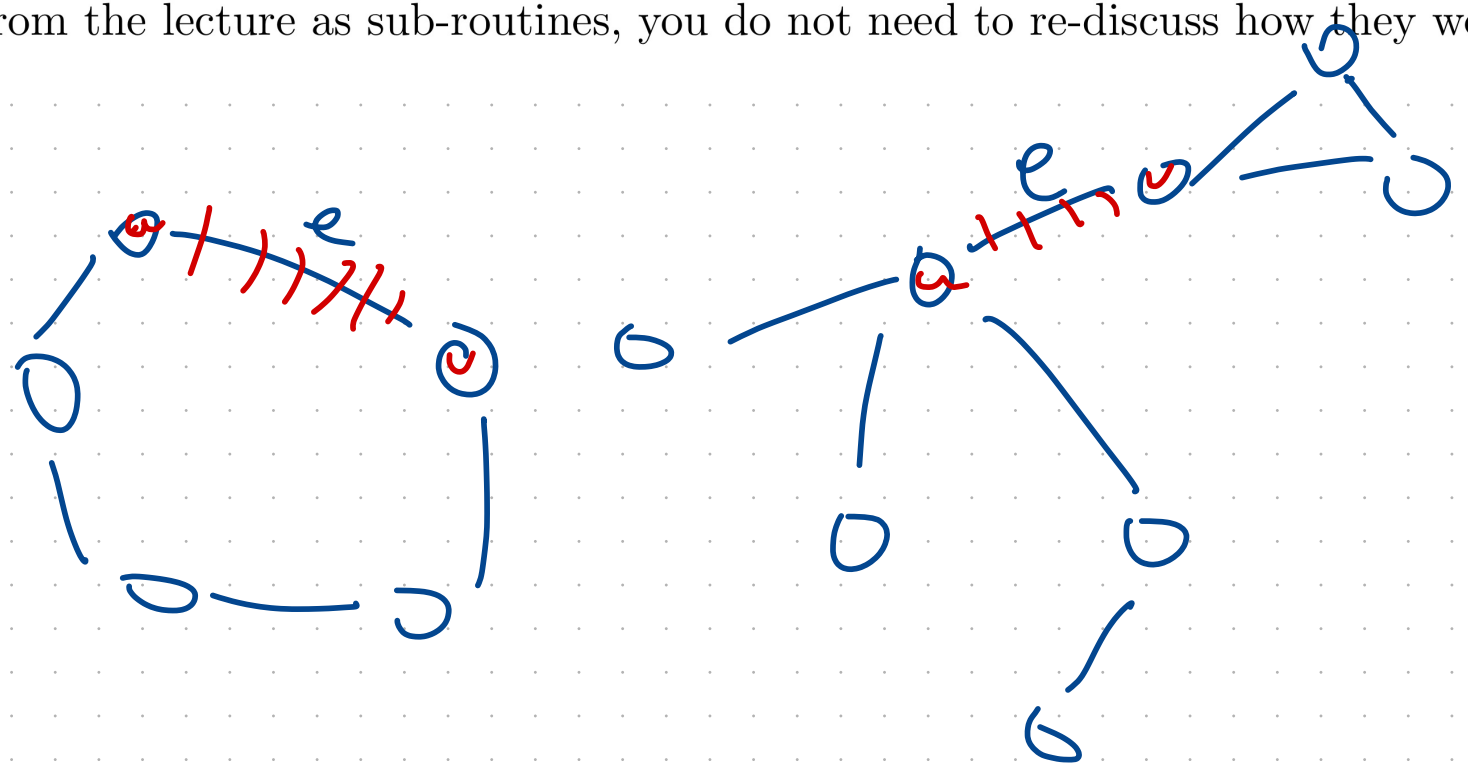
6

1. Introduce v_0 in the graph and an edge with weight zero to all other nodes.
2. Use Bellman-Ford from v_0 to calculate $h(v)$ for all $v \in V$. $w'(u,v) = w(u,v) + h(u) - h(v)$
3. Calculate the new weights using the function h.
4. Run $|V|$ times Dijkstra.
5. Use h to modify the distances found with Dijkstra.

Finding a cheap cycle

Let $G = (V, E)$ be a weighted undirected graph, where all edge weights are positive. Provide an efficient algorithm that, given an edge $e \in E$, outputs the weight of the cheapest cycle (that is, the cycle of smallest total weight) that contains e, and outputs ∞ if e is not contained in any cycle. Give the running time of your algorithm in terms of $|V|$ and $|E|$. In order to get full points, your algorithm should run in time $\mathcal{O}((|V| + |E|) \log |V|)$

You do not need to write a proof of correctness or a runtime analysis. If you use algorithms known from the lecture as sub-routines, you do not need to re-discuss how they work.

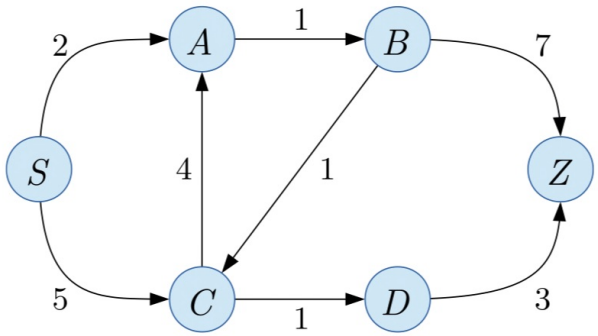


Remove e

Run Dijkstra

d(u,v) + c(e)

Sie haben ein Elektroauto und wollen von S nach Z fahren. Das Navigationssystem Ihres Autos hat eine Strassenkarte, in der alle Orte verzeichnet sind. Wenn es eine direkte Strassenverbindung von einem Ort i zu einem Ort j gibt, auf der keine weiteren Orte liegen, dann nennen wir j einen *direkt benachbarten Ort von* i (in der Abbildung rechts ist etwa B direkt benachbart von A, aber nicht von S). Von einem Ort i gelangt man zu einem direkt benachbarten Ort j in Zeit $l_{ij} \geq 0$.



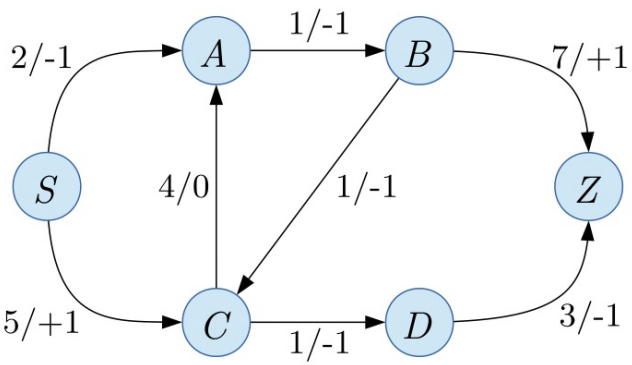
Modellieren Sie die nachfolgenden drei Fragestellungen jeweils als graphentheoretisches Problem. Geben Sie dazu an, welche Knoten und Kanten definiert werden und welche Gewichte den Kanten zugeordnet werden. Erläutern Sie in Teil b) und c), wie die Lösung der Fragestellung von einem geeigneten Weg abgelesen werden kann. Nennen Sie einen effizienten Algorithmus zur Lösung des graphentheoretischen Problems und bestimmen Sie seine Laufzeit in Abhängigkeit von der Anzahl der Orte n und der Anzahl direkter Strassenverbindungen m.

—

Gesucht wird ein schnellster Weg von S nach Z. In dem oben dargestellten Strassennetz ist dies der Weg (S, A, B, C, D, Z) mit Reisezeit 8. Vereinfachend nehmen wir für diese Teilaufgabe an, dass die Batterie des Elektroautos hinreichend gross dimensioniert ist, sodass Sie sich über das Nachladen keine Gedanken machen müssen.

Dijkstra

) Ab sofort nehmen wir an, dass die Batterie Ihres Elektroautos fünf Ladezustände $0, \ldots, 4$ annehmen kann (wobei 0 einer leeren und 4 einer komplett geladenen Batterie entspricht). Auf dem Weg von einem Ort i zu einem direkt benachbarten Ort j verändert sich der Ladezustand der Batterie um $e_{ij} \in \{-1, 0, 1\}$. Ein Wert $e_{ij} = +1$ bedeutet, dass die Batterie auf diesem Strassensegment aufgeladen wird (weil die Strasse abschüssig ist). Die Batterie kann aber niemals mehr als den Ladezustand 4 erreichen.



Befindet sich das Fahrzeug bei einem Ort i und hat die Batterie Ladezustand 0, dann darf ein Strassensegment zu einem benachbarten Ort j nur dann befahren werden, wenn auf dem Weg dorthin keine Energie benötigt wird. Zu Beginn der Fahrt ist die Batterie voll geladen.

Gesucht wird nun ein schnellster Weg von S nach Z, auf dem der Ladezustand der Batterie niemals negativ wird. In dem in der vorigen Abbildung dargestellten Strassennetz ist dies der Weg (S, C, D, Z) mit Reisezeit 9. Der Weg (S, A, B, C, D, Z) mit Reisezeit 8 ist keine gültige Lösung, da die Batterie bei Ort D Ladezustand 0 besitzt und somit zur Fahrt nach Z keine Energie mehr übrig ist.

we build a Graph $G' = (V_0 \cup V_1 \cup V_2 \cup V_3 \cup V_4, E')$
where V_i are copies of the nodes in G and E' are defined as follows:

for $(v, w) \in E$

if $e_{ij} = 0$: I insert an edge in E' with the cost of (v, w) from the copy of v to the copy of w in $V_0 \ldots V_4$

if $e_{ij} = 1$ I insert an edge in E' with the cost of (v,w) from the copy of v in V_i to the copy of w in V_{i+1} for $i = 0 \dots 3$ and from the copy of v in V_4 to the copy of w in V_4

if $e_{ij} = -1$ I insert an edge in E' with the cost of (v,w) from the copy of v in V_i to the copy of w in V_{i-1} for $i = 4 \dots 1$

Then I run Dijkstra from the copy of s in V_4 and return the minimum between the distances to z in the corresponding nodes in $V_0 \dots V_4$.

Runtime: In G' we have $5|V|$ nodes and $\leq 5|E|$ edges, so the runtime of building G' + Dijkstra in G' is upper bounded by $O((|V|+|E|)\log(|V|))$ with binary heaps or $O(|V| + |E|\log|V|)$ using Fib-heaps.

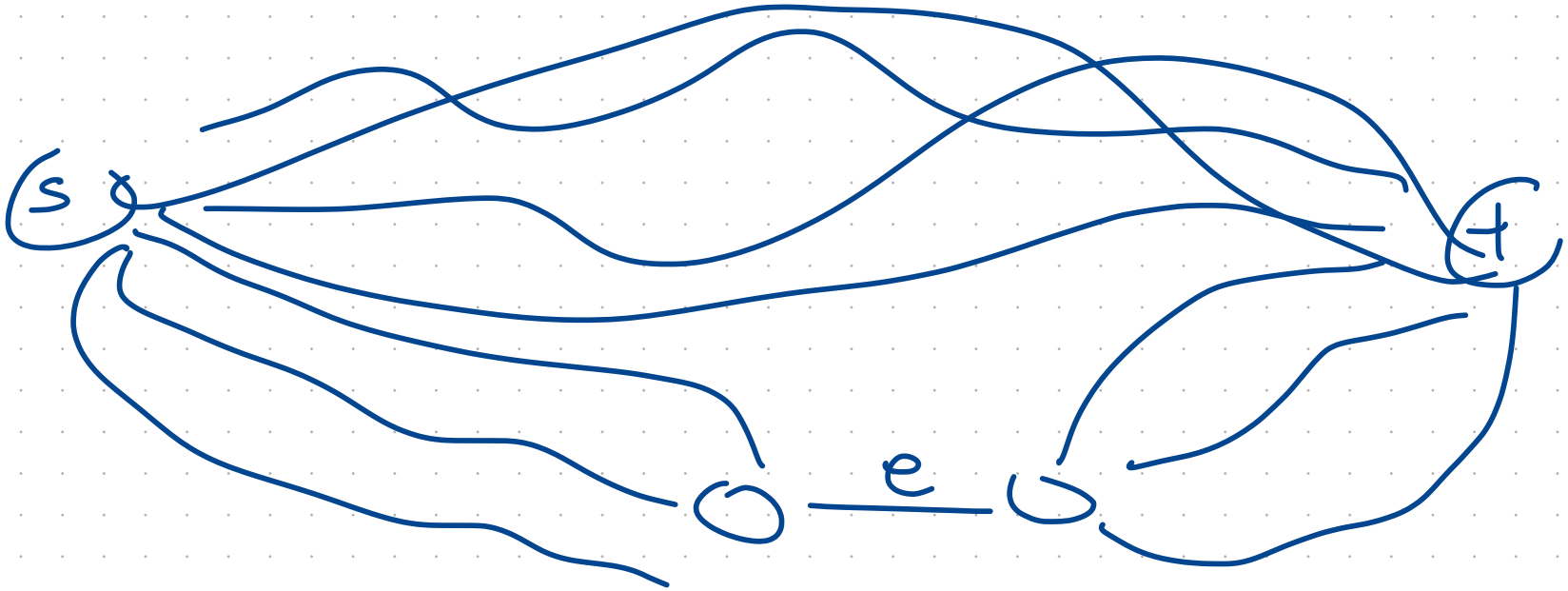
Wie in b) nehmen wir an, dass die Batterie fünf Ladezustände besitzt und zu Fahrtbeginn voll geladen ist. Gesucht wird nun ein (nicht notwendigerweise schnellster) Weg von S nach Z, auf dem der Ladezustand der Batterie wie zuvor niemals negativ wird, und bei dessen Ankunft bei Z die Batterie so voll wie möglich ist. In der unteren Abbildung auf der Seite vorher ist dies (S, C, A, B, Z), bei dessen Ankunft die Batterie voll geladen ist (bei S, C und A ist der Ladezustand 4, bei B dann 3 und bei Z wieder 4).

Same G' as before + DFS/BFS

TRICK

Sometimes you need to run an algorithm "twice" and combine their result

Example Given an edge $e = (u, v)$, is there a SP from s to t ($s \neq u, v$; $t \neq u, v$) that uses edge e?



Additional DP Trick Consider using 2 DP Tables

LIGHT COFFEE

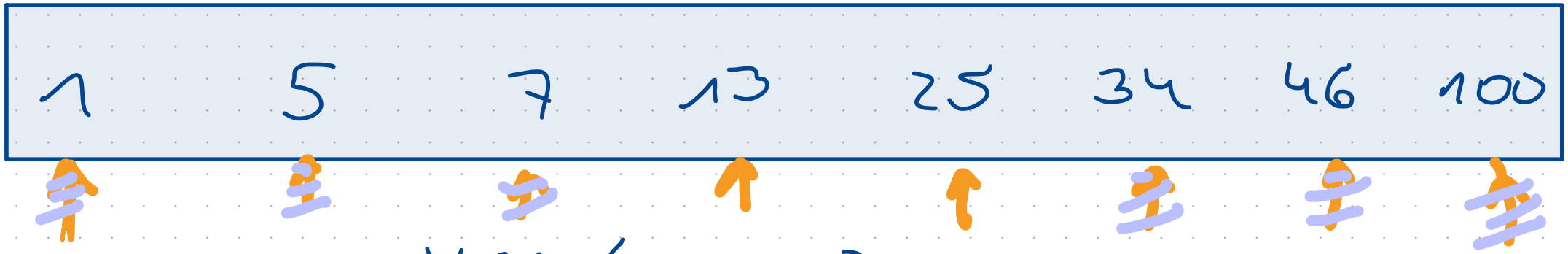
EHF (exactly) C ; considering coins 1…i ; 0: even #coins, 1: odd #coins

ALGO(C, n, 0)

ME ← new int[C](n)
MO ← new int[C](n)

```
ALGO( C , i , flag ) {
    if ( i = 0 ∧ C > 0 ) return -∞
    if ( i = 0 ∧ C = 0 ∧ flag = even ) return 0
    if ( i = 0 ∧ C = 0 ∧ flag = odd ) return -∞
    if ( flag = 0 ∧ ME[C][i] != -1 ) return ...
      |--------| ∧ |-- MO ________|  |______|
    choice1 ← wi + ALGO( C - vi , i - 1 , !flag )
    choice2 ← ALGO( C , i - 1 , flag )
    update ME/MO (based on flag) with max(choice1, choice2)
    return max( choice1 , choice2 )
}
```

Sorted Array. Are there two numbers in this array that sum up to a value K?



K = 38 YES (13+25)

K = 37 NO

Trick Sliding window