# 1 Introduction (Harder)

There exist many different problem fields in computer science. For all problems ~~in computer science~~ it is not only necessary to solve them correctly but also to solve them as fast as possible respectively efficiently, even for very large instances of a problem.
In the past semester (WS 2017/18) we participated in the Lab "Efficient Algorithms for Selected Problems: Design, Analysis and Implementation". We had to solve different types of problems by implementing algorithms and strategies in Java. There have been tasks related to some problem fields in computer science, for example greedy algorithms, dynamic programming, geometric problems and graph algorithms. For some tasks we could use well-known algorithms and patterns to solve them in a straightforward way. In some cases it ~~was~~ has been necessary to modify well-known algorithms such that they could be applied on special problems or transform ~~tasks~~ problems in a way that they were solvable by using known strategies.
In this report we ~~want~~ would like to list and explain some of the problem fields we had to use. For each section we are to give some exemplary problems, point out interesting modifications ~~and~~, transformations and explain how we solved the tasks.

We have been confronted with

# 2 Tasks on number streams (Herschel)

**Input:** A Stream of numbers
**Output:** for example

- a certain element of the stream fulfilling a property
- a newly computed number fulfilling a property
- a classification of each element
- ...

When it comes to handling streams of numbers, there are two types of input. Either all numbers are known in advance or they do come in a stream one by one while running the algorithm.

During our LAB we had to ~~handle~~ solve a few ~~task~~ problems about numbers. Here we are to ~~will~~ point out a few of them to show where the difficulties lie and how to ~~handle~~ work with them.

→ handle und solve würden schon viel zu oft verwendet

## 2.1 Floating Median

(Task 2.4, Floating Median)

Given a stream of numbers one might want to know the median from the numbers seen so far (Task 2.4). Given a sorted set of numbers. $a_1, ...a_k$ following formula can be used to compute the median

$a_{(k+1)/2}$ if k is odd

# 6 Graph Algorithms (Harder)

Many problems in computer science can be solved by graph algorithms. A *graph* $G = (V, E)$ consists of a finite set $V$ of nodes and a finite set $E$ of edges. If the graph is *undirected* $E$ is a set of unordered pairs of nodes, if the graph is *directed* $E$ is a set ordered pairs of nodes. Each edge $e$ can have an additional value which is representative for the weight (length, capacity) of $e$. If a graph does not have any weights one can assume that each edge $e$ has a weight equal 1.

There are some commonly known and used problem fields on graphs.

- Graph exploration
- Shortest paths
- Minimal spanning trees
- Maximum flow problems

Depending on the graph (directed or undirected edges, positive or negative weights, different weights or all weights equal 1), there are different algorithms to solve these problems. In the Lab "Efficient Algorithms for Selected Problems: Design, Analysis and Implementation" we had to solve different types of graph problems. In some cases it was possible to solve tasks with well known algorithms. For more complicated tasks it was necessary to modify an algorithm or apply it multiple times.
In the following we will give a few example tasks for the different problem fields, explain how we solved the task and point out interesting properties and modifications to algorithms.

## 6.1 Graph exploration

A relative simple graph problem is to perform a graph exploration (or search). Possible tasks would be to find out the number of components in a graph, calculate the distances between a source and second node (this only works if all edge weights are 1, see the shortest paths section 6.2) or to check if a graph is bipartite. There exist two algorithms for graph explorations: Depth-First Search (DFS) and Breadth-First Search (BFS). Both algorithms traverse the graph in a tree based manner and visit all nodes in the component once.

As mentioned above it it possible to check if a graph is bipartite. Exercise 3.6 "Bipartite or not ?" is one example for this exercise type:

> Given was an undirected graph. The task was to decide if the given graph is bipartite or not.

To decide if a graph is bipartite it is sufficient to decide if a graph has an odd-length cycle. If there is an odd-length cycle, the graph is not bipartite. If no such cycle exists the graph is bipartite (see figure 4). To solve the odd-cycle-problem

we used a special version of BFS. In this version we defined two flags (colors): 1 and -1. The starting (root) node is initialized with the color 1. During BFS each new visited node gets the opposite color to its parent node. If two nodes with the same color are connected by an edge the graph has an odd-length cycle which means it is not bipartite. If no nodes with same colors are connected there is no odd-length cycle and the graph is bipartite.

## 6.2 Shortest paths

One common problem field on graphs are shortest path problems. In shortest paths problems one is looking for the smallest *distance* between two nodes in a graph or the shortest *path* between two nodes (a chain of nodes from one starting node (*source* or *root*) to an ending node (*target*)). There are two sub problems: Singe-Source Shortest Path Problem (SSSP) and All-Pairs Shortest Path Problems (APSP). SSSP calculates the shortest path distance from one source node to all other nodes in the graph. APSP calculates the distance between all pairs of nodes inside the graph.
In general APSP problems can be solved by using an algorithm which solves the SSSP problem for each node, but there are also algorithms which solve an APSP problem immediately.

Depending on the graph we have to use different algorithms. One has to distinguish between different cases with respect to the edge weights:

- all edges have the same weight
- edges have non-negative weights
- edges have positive and negative weights

All these cases can occur as SSSP problem or as an APSP problem. For graphs where all edges have the same length one can use the BFS Algorithms for the SSSP problem. In this case a graph has arbitrary non-negative edge weights the singe source shortest path problem can be solved by Dijkstra. For the third case in which one wants to calculate shortest paths in a graph with negative edge weights one has to guarantee that there is no negative cycle, because otherwise it is not possible to calculate distances. Some algorithms, for example Bellmann-Ford, can detect negative cycles and calculate the distances if no negative cycle exists (see below). As mentioned before it is possible to calculate a APSP problem by applying a SSSP algorithm for each node. Nevertheless there are also algorithms to solve a APSP problme, for example for the Floyd Warshall Algorithm.
In the following we want to give some examples where we had to solve different tasks for the different cases.

**SSSP problem with non-negative edge weights**

At first we have a look at a SSSP problem with non-negative edge weights: Dijkstra is an algorithm to solve the SSSP problem for graphs with positive edge weights. This means it calculates the distances from a source node to all other nodes in the graph with non-negative weights. For negative weights Dijkstra is not working and a different algorithm, for example Bellmann-Ford-Algorithm (see below), has to be used.
The Dijkstra-Algorithm generates a shortest path tree from a given root. A shortest path tree is a spanning tree such that the shortest paths from all nodes to the source lie on this tree and are minimal.
To generate the shortest path tree, the algorithm starts in the source and explores the graph always going to the edge with the smallest weight, which leads to a not (yet) visited node.

In Exercise 4.4 "Friendship" we used the Dijkstra algorithm with a few task related modifications:

> Given was a graph with different (non negative) edge weights. Four special nodes were defined: Node $A_1$, $A_2$, $B_1$ and $B_2$. The task was to calculate the length of the shortest path from $A_1$ to $A_2$ without using a node which lay on one of the shortest paths between $B_1$ and $B_2$. If there did not exist a path between $A_1$ to $A_2$ the result was -1.

To solve this problem two separate steps ~~where~~ necessary: In the first step, we calculated all shortest paths from $B_1$ to $B_2$ and removed all nodes which were part of one of these paths from the graph. To calculate all paths between $B_1$ and $B_2$ it was not possible to use the normal Dijkstra implementation, because only one of the shortest paths would be found. To find multiple paths we had to make a small modification: We changed the algorithm such that Dijkstra did not stop when the goal (here $B_2$) was reached. It continued until it was not possible anymore to reach the goal with the same distance compared to the first time (so we could not find paths which were not minimal). With this modification it was possible to find multiple paths from $B_1$ to $B_2$. But this modification was not sufficient, because the algorithms could not find paths were parts of the paths were equal (see figure 5). Even with the first modification these paths would not be found by the way Dijkstra works, because Dijkstra visits every node at most only once. To find all paths we modified the algorithm s.t. one node can be visited more then once, if the visits came from different nodes. Each node managed a list of nodes from which this node was visited with the same shortest distance to the source (see pseudo code in algorithm 5). With both modifications together we found all paths from $B_1$ to $B_2$ and removed them from the graph. In the last step we used the "normal" (not modified) Dijkstra algorithm to find the shortest path from $A_1$ to $A_2$ on the modified graph. Because we removed all paths in the first step we could guarantee that the shortest path between $A_1$ and $A_2$ did not use a node which lied on one of

the shortest paths between $B_1$ and $B_2$. The length of the calculated path was the result of the task, if no path existed the result was -1.

**Algorithm 5** modified RELAX procedure of the Dijkstra Algorithm

```
1: procedure RELAX(u, v)
2:     if d(v) > d(u) + w(u, v) then
3:         d(v) ← d(u) + w(u, v)
4:         π(v).clear()
5:         π(v).add(u)
6:     else if d(v) == d(u) + w(u, v) then
7:         π(v).add(u)
```

### SSSP with all edge weight equal 1

In the special case that all edges in a graph have weight 1 it is possible to use the Breadth-First Search (BFS) algorithm instead of Dijkstra. We saw BFS earlier in the graph exploration section (see section 6.1). One example to use BFS to solve a shortest path related problem is Exercise 4.3 "Collecting Eggs".

> Given was a graph with weight 1 on each edge, a start node ($s$) and a destination node ($t$). With an arbitrary number of paths from the start to the end node every node had to be visited at least once. The task was to minimize the length of the longest path, such that all nodes where visited at least by one path.

To solve this task we had to find distances from each node $v$ to the start point ($d(v, s)$) and to the end point ($d(v, t)$). The sum of both distances ($d(v, s) + d(v, t)$) is the minimum path length for a path from $s$ to $t$ which contains node $v$.

To calculate both distances for every node in the graph, we had to solve the SSSP problem twice: The first time from the start point $s$ to get the distance from $s$ to every node in the graph, and a second time from the end point $t$, to get the distance from $t$ to the other nodes. Because each edge had weight 1 it was possible to use BFS to solve both SSSP problems. After using BFS twice (first time with root $s$, the second time with root $t$) the distances to the start and to the end node were known for each node (see figure 6). To calculate the minimal length of the longest path, we had to add both distances in every node ($d(v, s) + d(v, t)$) and get the maximum of all these sums. The maximum sum over all nodes was the result of the problem.

### SSSP with positive and negative edge weights

Until now we assumed that all weights in the graph were non-negative, because neither Dijkstra nor BFS can handle negative edge weights. If a graph has some negative edge weights and we are looking for the shortest path between nodes, we have to use a different algorithm, for example the Bellman-Ford algorithm.

Bellmann-Ford can handle negative edge weights if there is no negative cycle (see figure 7). If there is a ~~negative cycle~~ one it is not possible at all to calculate distances because the ~~distances~~ become arbitrary small (by looping through the negative cycle arbitrary often). In this case Bellman-Ford detects the negative cycle and outputs that it is not possible to calculate distances because a negative cycle exists. The property that the algorithm detects a negative cycle is very useful. We implemented the algorithm to solve Exercise 4.6 "Time travel" by using this helpful property.

> Given was a graph with arbitrary edge weights (positive and negative weights). The task was to detect if a negative cycle existed in the graph or not.

To solve this task we had to use the Bellman-Ford algorithm once. If there ~~existed~~ exists a negative cycle in the graph Bellman-Ford would detect it and threw an error. So if there had been an error, we ~~knew~~ would have known there was a negative cycle. If the algorithm terminated successfully without error we knew there was no negative cycle.
The idea of the Bellman-Ford algorithm is to calculate at first distances with at most one edge in the path. Then it increases the number of edges in each step (at most two edges, at most three edges and so on). We know that (shortest) paths in a graph can have at most length of number of nodes - 1. This is why we continued this calculation until we reached this maximum length. If after number of nodes - 1 iterations the shortest path still can be improved the algorithm knows that there has to be a negative cycle and outputs this result.

-7 1 und doppelte --

### APSP with all edge weights equal 1

Sometimes it is not sufficient to get all distances from a single source to all nodes in the graph. It can be necessary to calculate the distances from all possible node pairs in a graph. As mentioned above one can solve this by applying a SSSP algorithm (in this case e.g. BFS, see above) for each node in the graph. But there exist also some algorithms which solve this problem immediately. One of these algorithms is the Floyd Warshall Algorithm.
We used this algorithm in the Exercise 4.5 "Small-world experiment":

> Given was a graph with no specified weights (this means one can assume all weights equal 1). The task was to calculate the *diameter* of the graph.

The diameter is the longest shortest path in a graph, this is the maximum of all shortest paths between all nodes. If the graph has multiple components the result is infinite.
To calculate the diameter of a graph we had to calculate the shortest paths between all nodes first and get the maximum of all these shortest paths. Calculating the length of the shortest paths between all nodes is an APSP problem (all-pairs shortest paths problem). To solve this APSP problem we used the Floyd-Warshall-Algorithm.

The Floyd-Warshall algorithm uses the idea of dynamic programming (see section 5). It solves for each node pair the shortest paths with only a limited set of nodes. By increasing this limited set it solves different sub problems and puts all sub problems together to the shortest path problem over the complete set of nodes.
After we calculated the length of the shortest path between all pairs, we had to get the maximum over all shortest paths with a simple loop over all shortest paths. This maximum value was the result of the algorithm.

## 6.3 Minimal spanning trees

For some graph problems it is helpful to look at spanning trees. A spanning tree of a graph is a sub graph (with no cycles) which connects all nodes together. The minimum spanning tree (MST) is one of the spanning trees with weight (sum of the weights of all used edges) less than or equal to the weight of every other spanning tree for a weighted graph. Note: For a graph without specific weights, respectively each edge has weight 1 every spanning tree is a MST, because the number of edges in each spanning tree is the same.
The Kruskal-Algorithm generates one of the minimum spanning trees of the given graph with help of a special data structure: the *union-find-structure*. The union-find-structure manages a set of disjoint sets $S_1$, $S_2$, ... $S_n$. Each set $S_i$ has a representative element $e \in S_i$. The data structure has two important operations: *Find(x)* returns the representative element of the set $S$, where $x \in S$. *Union(x, y)* replaces the sets $S_x$ and $S_y$ with the union of both sets $S_x \cup S_y$. The new set $S_x \cup S_y$ has a new representative element $e \in S_x \cup S_y$.

We used the Kruskal-algorithm in Exercise 3.5 "Railway network":

> Given was a set of nodes in the euclidean plane and a set of edges. The length of an edge is the euclidean distance between both nodes connected by this edge. The task was to connect all nodes with a spanning tree such that the longest edge (highest length) which was used is minimal. This longest length was the result of the problem.

To calculate the minimal spanning tree for this task we used the Kruskal-algorithm: First we calculated the fully-connected graph. This means we calculated the euclidean length of all edges between all given nodes and added them in a list. Then we sorted the edge list with respect to the length in increasing order. For each edge (starting with the shortest edge) we used the *find* operation for both nodes $a$ and $b$ which were connected by this edge. If both nodes had the same representative element (same result of the *find* operation) they were in the same set. Being in the same set means that they were already connected within a spanning tree. In this case we did not insert this edge in our spanning tree but discard it. If both nodes had different representative elements (different results of the *find* operation) there was not (yet) a path between these two nodes. So we inserted this edge in our spanning tree with help of the *union* operation. Both sets $S_1$ and $S_2$ with $a \in S_1$ and $b \in S_2$ were merged to a new

set $S_1 \cup S_2$ with a new representative element $e \in S_1 \cup S_2$.

Sometimes it is useful to just use the union-find-structure instead of the complete Kruskal-Algorithm. This case occured in Exercise 2.7. "An online graph problem".

> Given was a set of nodes and an empty set of edges. Then a series of two event types occurred: The add event adds an edge between two node in the graph (if there has not been already a connection between the two nodes) and the query event queries if there is a connection between two nodes the task was to find the number of successful queries (both nodes were connected) and the number of queries which were not successful (both nodes were not yet connected).

To execute both events as fast as possible we used the union-find-structure of the Kruskal algorithm. If an add event occurred we tested with the *find* operation if both nodes were in the same set (have the same representative element). If both representative elements were different, we added the edge to the graph by using the *union* operation with both nodes. If both nodes were inside the same set (have the same representative element) we did nothing, because this means that both nodes were already connected (within the spanning tree). If a query event occurred, we used the *find* operation. If both nodes were in the same set (had the same representative element), they were connected and we increased the success counter. If both nodes were in different sets (had different representative elements) they were not (yet) connected and we increased the counter for not successful queries. These two counters were the result of the problem.

## 6.4 Maximum flow problems

The maximum flow problem on a graph is a problem to get a maximum flow from a starting node (often called *source* $s$) to a destination node (*sink* $t$). Each edge $e$ can have a flow between 0 and the *capacity* (weight) of the edge. Each node but $s$ and $t$ has the same amount of incoming flow as of exiting flow.

One of the algorithms which can solve the maximum flow problem is the Ford-Fulkerson algorithm: This algorithm starts with an initial total flow of 0. While there exists a path from $s$ to $t$ with which the total flow can be increased, it adds the flow of this path to the total flow. For each used path the adjacency matrix is updated such that all used edges decrease their value by the path flow and the inverse edges increase their value by the path flow. When no flow increasing path is left the algorithm terminates and outputs the total flow.
Edmonds-Karp algorithm is a improvement of Ford-Fulkerson. It follows the same algorithmic idea but instead of using an arbitrary path from $s$ to $t$ it uses the shortest possible path. This small modification improves the efficiency for worst case configurations dramatically.

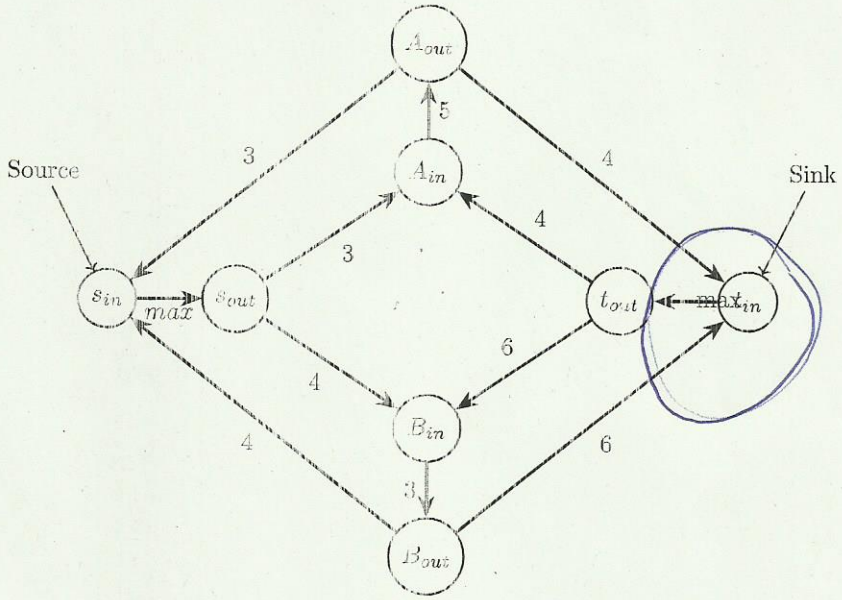


Figure 11: Modified maximum flow problem with directed edges. Capacities of Node $N$ were transformed in edges between $N_{in}$ and $N_{out}$.