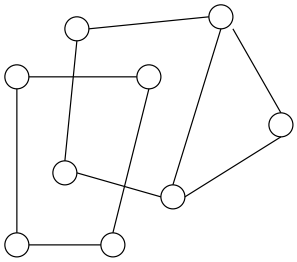
1 Content and motivation of the practical course

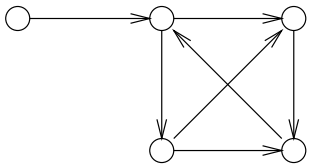
In this course "Algorithmen-Entwurf" we will implement several efficient algorithms in C++ and visualize their behavior. We will have a look at basic algorithms like Dijkstra's algorithm for the shortest path problem, as well as some more specialized algorithms for more complex problems. In terms of implementation purposes, we focus on efficiency on one hand, i.e. that the best possible worst-case bounds are met. On the other hand, the focus is on a clear visualization, helping to understand the methodology of the algorithm. The participants of this course will hereby see the benefits of using higher data structures for better performance and will develop a better understanding of the methodology of some typical algorithms. Better C++ programming skills will surely be gained as well. For some exercises, the algorithm will not be given explicitly, but instead we ask the participants to independently create an algorithm which should be as efficient as possible.

2 Graphs

A graph $G = (V, E)$ consists of a set of vertices V and a set of edges/arcs E. Each edge/arc $e \in E$ connects two vertices of V. We distinguish between directed and undirected graphs. In a directed graph, the arcs (directed edges) are given as ordered tuples (u, v) of vertices; u is the start vertex of the arc, v its end point. Arcs are usually drawn by arrows. Undirected edges have no orientation and are represented as two-element subsets $\{u, v\}$ of V.



undirected graph



directed graph

The number of vertices is denoted by $|V| = n$, the number of edges by $|E| = m$. The running time of the presented algorithms will be calculated in relation to n and m. Algorithms with a worst-case running time of $O(n + m)$ are called *linear*.

In an undirected graph, a connected component is a maximal subset of vertices, where all vertices within that component are pairwise reachable by a path. The left graph in the example above has two connected components.

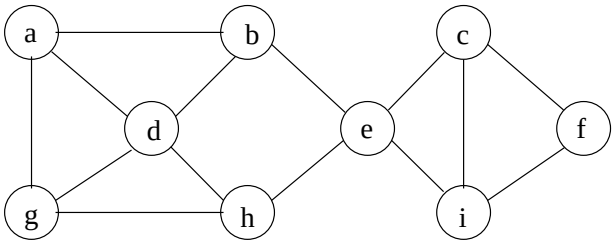
The data structure which is usually used to represent a graph in a program consists, roughly speaking, of a list of all vertices, a list of all edges, and for every vertex a list of its neighboring (incident) edges. For directed graph, two separate lists for outgoing and incoming edges are used. In general, vertices and edges are indexed. An alternative way to represent a graph uses an adjacency matrix, but a graph with n vertices needs space $\Omega(n^2)$ for that.

Many practical problems can be modeled naturally by graphs and can be represented as graph problems, e.g. path finding problems, mapping problems, and games.

3 Depth-first-search and Breadth-first-search

One basic problem is graph searching. The most common types used are the depth-first-search (DFS) and the breadth-first-search (BFS). Both types start with an unvisited vertex in the graph and visit all vertices which are connected to this vertex by edges (in the directed case, arcs are only traversed in forward direction). Then, if there are still some unvisited vertices, the DFS or BFS will be continued at an arbitrary unvisited vertex.

At the DFS or BFS the neighboring edges, or the neighboring vertices resp., of the current vertex are examined. By examining an unvisited neighboring vertex w of the current vertex v, the DFS searches on from w and afterwards returns to v, after the subgraph reachable from w is processed completely; only then the other neighboring edges of v are examined. The BFS, on the other hand, examines all the neighboring edges and remembers which vertices are examined for the first time; they will be inserted into a queue. The vertex that is visited next by the BFS will come from the queue. The BFS first visits all vertices which are reachable from the start vertex via one edge, then those which are reachable via two edges, and so on.



If, in this example, we choose e as the start vertex, a DFS could visit the vertices in the order $e, h, g, a, d, b, i, f, c$, whereas a BFS could visit the vertices in the order $e, i, c, b, h, f, d, a, g$. Generally, for the implementation of the DFS a recursive function is used, for BFS a queue (FIFO) is used. Both DFS and BFS have a linear running time, because each vertex and each edge is only visited a constant number of times.

If we index the vertices in the order they are visited by a DFS (or BFS, resp.) we get a so-called DFS (or BFS, resp.) numbering, which is generelly not unique. A DFS (or BFS, resp.) in an undirected connected graph yields a so-called *DFS tree* (or *BFS tree*, resp.), namely that subset of edges which are traversed from a visited vertex to find an unvisited one.