Introduction to Graphs

Alexander K. Hartmann
hartmann@theorie.physik.uni-goettingen.de
Institut für theoretische Physik, Bunsenstr. 9
37073 Göttingen, Germany
Tel. +49-551-395995, Fax. +49-551-399631

October 26, 1999

Contents

1 Graphs

Many optimization problems from physics or other areas can be mapped on optimization problems on graphs. Some of this transformations will be useful later on. For this reason a short introduction to graph theory is given here. Only the basic definitions and algorithms are presented. For more information, the reader should consult a specialized textbook on graph theory, e.g. [3, 4, 11]. We will sometimes just mention a real world application which can be treated with a given graph theoretical concept. The precise definitions of the applications will be given later on.
Consider a map of a country where several towns, villages or other places are connected by roads or railways. Mathematically this setting can be represented by a *graph*. A graph consists of *nodes* and *edges*. The nodes represent the towns,

villages or other places and the edges describe the roads or railways. Formally the definition of a graph is given by:

Definition: Graph A graph G is an ordered pair $G = (V, E)$ where V is a set and $E \subset V \times V$. An element of V is called *vertex* or *node*. An element $(i, j) \in E$ is called *edge*. $\square$

If the pairs $(i, j) \in E$ are ordered pairs, then G is called a *directed* graph. Otherwise G is called *undirected*, then (i, j) and (j, i) denote the same edge. A graph $G' = (V', E')$ is called *subgraph* of G if it has the properties $V' \subset V$ and $E' \subset E$ ($E' \subset V' \times V'$ by definition). The empty graph $(\emptyset, \emptyset)$ is a subgraph of all graphs.

First, some further notations are given which apply to both directed and undirected graphs. Usually we restrict ourselves to *finite* graphs, i.e. the set of nodes and edges are finite. In this case we denote by $n = |V|$ the number of vertices and by $m = |E|$ the number of edges. Let $i \in V$ be a vertex. If $(i, j) \in E$ we call j a *neighbor* of i (and vice versa). Both nodes are *adjacent* to each other. The set $N(i)$ of neighbors of i is given by $N(i) = \{j|(i, j) \in E \vee (j, i) \in E\}$. The degree $d(i)$ of node i is the cardinality of the set of neighbors: $d(i) = |N(i)|$. A vertex with degree 0 is called *isolated*.

A *path* from v_1 to v_k is a sequence of vertices $v_1, v_2, \ldots, v_k$ which are connected by edges: $(v_i, v_{i+1}) \in E \; \forall i = 1, 2, \ldots, k-1$. The *length* of the path is $k-1$. If $v_1 = v_k$ the path is called *closed*. If no node except the first and the last one appears twice in a closed path, it is called a cycle. A set of nodes is called a *connected component*, if it contains all nodes where from each node a path to each other node of the set exists. A graph is called *connected* if it has only one connected component.

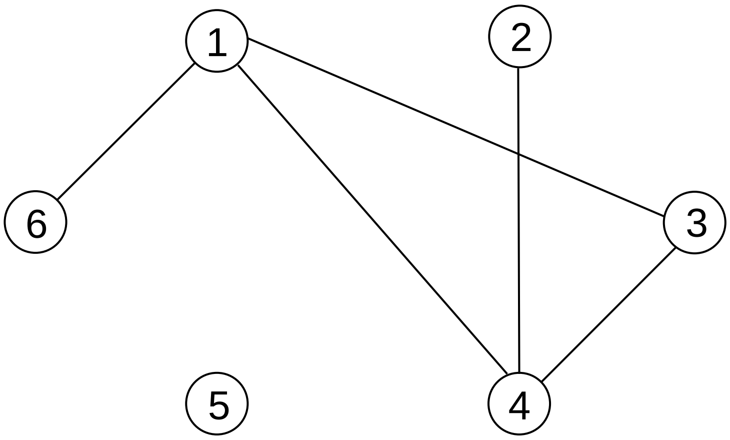

Figure 1: An undirected graph

Example: Graph

In Fig. 1 the graph $G = (\{1, 2, 3, 4, 5, 6\}, \{(1, 3), (3, 4)\ (4, 1), (4, 2),$

$(6,1)\}$) is shown. The nodes are represented by circles and the edges by lines connecting the circles. The graph has $n = 6$ vertices and $m = 5$ edges. E.g., nodes 3 and 4 are adjacent. The set of neighbors of vertex 1 is $N(1) = \{3, 4, 6\}$. Thus, node 1 has degree 3 while node 2 has only degree 1. Node 5 is isolated. The graph contains the path 6, 1, 4, 3 from node 6 to node 3 of length 3 and the cycle 1, 3, 4, 1. □

Now some definitions are given which apply only to directed graphs. For an edge $e = (i, j)$, i is the *head* and j the *tail* of e. The edge e is called *outgoing* from i and *incoming* to j. Please note that for a directed path it is important that all edges point into the direction of the path, formally the definition is the same as in the case of an undirected graph. A set of nodes is called a *strongly connected component*, if from each of its nodes a directed path to each other node of the set exists.

In a directed graph, the outgoing and incoming edges can be counted separately. The indegree is given by $id(i) = |\{j|(j,i) \in E\}|$ and the outdegree is $od(i) = |\{j|(i,j) \in E\}$. Obviously, for all vertices $d(i) = id(i) + od(i)$.

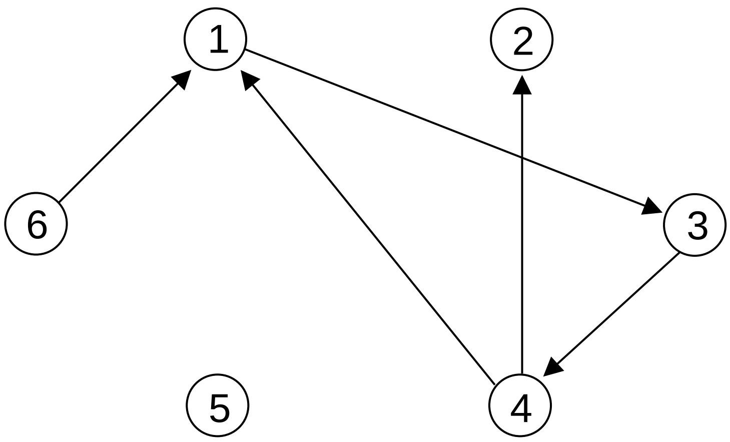

Figure 2: A directed graph

Example: Directed graph

When drawing a directed graph, the edges (i, j) are represented by arrows pointing from i to j. If one considers the example graph from above as directed, one obtains Fig. 2. Here, the sequence 6, 1, 4, 3 is not a path since the edges $(4, 1)$ and $(3, 4)$ point into the wrong direction. On the other hand, the graph contains the path 6, 1, 3, 4. Node 1 has outdegree $od(1) = 1$ and indegree $id(1) = 2$. The total degree is $d(1) = id(1) + od(1) = 3$ as in the case of the undirected graph. □

---

2 Trees and lists

A very important subclass of graphs are connected graphs without cycles. They are called *trees*. The name arises from the fact that it is possible to draw a tree in the following way:

- All nodes are arranged in several levels $l = 0, 1, \dots, h$.
- There are edges only between vertices of adjacent levels l and $l + 1$. A node on level l is called *father* and its neighbor on level $l + 1$ is called *son*.
- On level zero there is only one node, called the *root*.

Hence, trees are very often used to represent hierarchical structures. Let be v a node. The sons of v, their sons, and so on are called *descendants* of v. The father of v, its father, and so on are called *ancestors* of v. Thus, the root is ancestor of all other nodes. The nodes which have no descendants are called *leafs*. The index h of the largest level is called the *height* of a tree. The height is equal to the length of the longest path from the root to a leaf. Each node v can be regarded as a root of a *subtree*, which is given by v and its descendants.

Example: Tree
In Fig. 3 an undirected tree is shown. The root is usually displayed at the top. Here, node 7 is the root. Node 6 is descendant of the root and ancestor of node 5. Nodes 1, 2, 5, 9, 10, 11 are leafs. The tree has height 3. The subtree which has node 3 as a root contains the nodes 2, 3, 4, 11. □

For directed graphs, usually graphs without cycles are called trees only in the case if there are only edges from levels l to $l + 1$ and no edges in the other direction. Sometimes a directed graph is called a tree only if the undirected version of the graph contains no cycles.
An important application of trees in the field of computer science are *search trees*. They are used to store a collection of objects in an ordered way. Thus, an order relation "$\leq$" must be given for the objects. Search trees have the following properties:

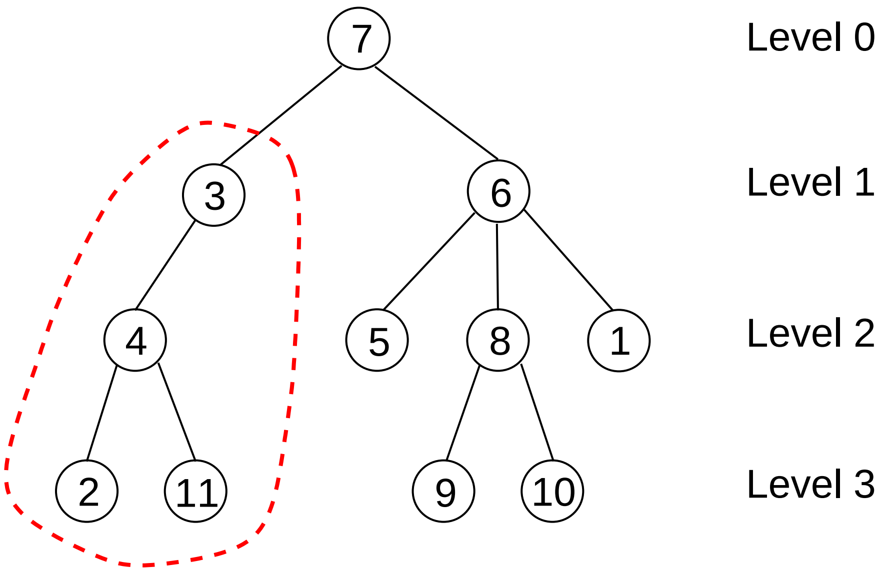

Figure 3: A tree containing 11 nodes.

- They are *binary* trees. This means that each node has at most two sons, called *left* and *right* son. The subtree which has the left (right) son as root is called *left (right) subtree*.
- In each node one object O_1 is stored.
- The following property is true for all objects O_1: In the left (right) subtree only objects are stored which are smaller (equal to or larger) than O_1 oocording the order relation "$\leq$".

This special organization allows to design very fast algorithms for finding,inserting and deleting elements within a search tree while always keeping the correct order. An example will be given in section 5. In Fig. 4 a search tree containing natural numbers sorted in ascending order is shown.
A binary tree is called *complete* if each node is either a leaf or has two sons. It can be shown easily by induction that a complete tree with height h has $n = 2^{h+1} - 1$ nodes and 2^h leafs.
A very simple kind of graphs are *lists*. They can be regarded as a special kind of trees which have exactly one root and exactly one leaf and every node has at most one son. The nodes of a list are called *elements*, the root is called *head* of the list and the leaf is called *tail*. For historic reasons lists are often drawn in a slightly different manner than other graphs. In Fig. 5 a list containing 5 elements is shown.
Several special types of lists are very useful in computer science. For *Queues* it is only possible to add new elements at the tail and to remove elements at the

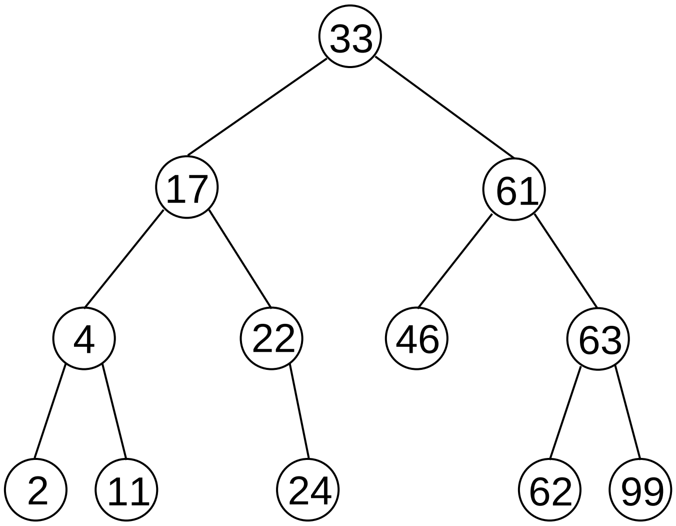

Figure 4: A binary tree containing the nodes 2, 4, 11, 17, 22, 24, 33, 46, 61, 62, 63, 99 sorted in ascending order.

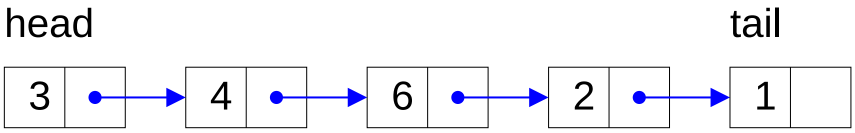

Figure 5: A list containing 5 elements.

head, see Fig. 6. They are called FIFO ("first in first out") lists as well. An application of queues are printer queues. The job which enters the queue at first will be printed at first. Other jobs which are added while a job is being printed have to wait.
Lists are called *stacks* if adding and removing elements takes place only at the head (see Fig. 7). They behave in a LIFO ("last in first out") manner. Stacks are used for example to realize recursion. Whenever a call to a procedure occurs the computer has to remember where to proceed when the procedure is finished. These so called *return addresses* are stored on a stack, because the procedure which was called at last will be finished at first.

3 Networks

Consider again the TSP (see first part). It can be represented by an undirected graph if one identifies the cities with nodes and the connections between cities with edges. In order to describe the TSP completely, the distances between different cities have to be represented as well. This can be done by introducing a

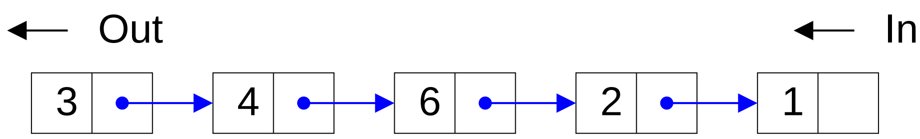

Figure 6: A queue.

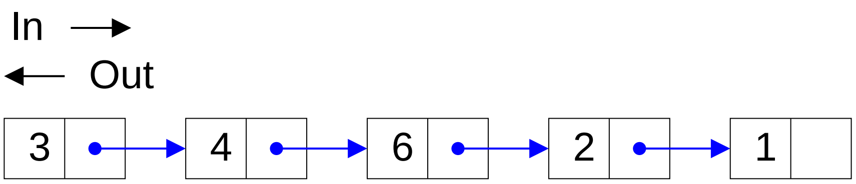

Figure 7: A stack.

function $d : E \to \mathcal{R}$ from edges to real numbers. For an edge $e = (i, j) \in E$ the distance between the cities i and j is given by $d(e)$.
Sometimes functions $f : V \to A$ (A an arbitrary set) on vertices are useful as well. An arbitrary function on vertices or on edges is called *labeling*. A graph together with a labeling is called a *labeled* graph. Typical examples for labelings are distances, costs or capacities.
The case where the edges are labeled with capacities is so important that it has its own name: a *network*. For example a system of pumps connected via pipes which can transport some fluid like water is a network. Our kind of networks contain two special pumps: the source where the fluid enters the network and the sink where the fluid leaves the network. The capacities of the pipes tell how much fluid can be transported through each pipe per time unit. The exact definition reads as follows:
Definition: network A network is a tuple $N = (G, c, s, t)$ with

- $G = (V, E)$ is a directed graph without edges of the form (i, i).
- $c : E \to \mathcal{R}_0^+$ is a positive labeling of the edges. Additionally, we assume $c((i, j)) = 0$ if $(i, j) \notin E$.
- $s \in V$ is a vertex called *source* with no incoming edges $di(s) = 0$.
- $t \in E$ is a vertex called *sink* with no outgoing edges $do(t) = 0$.

$\square$

In Fig. 8 an example of a network is shown. The capacities are the numbers written next to the edges. Please note that it is possible to define a network via an undirected graph as well. For the models we encounter here, directed networks are sufficient.

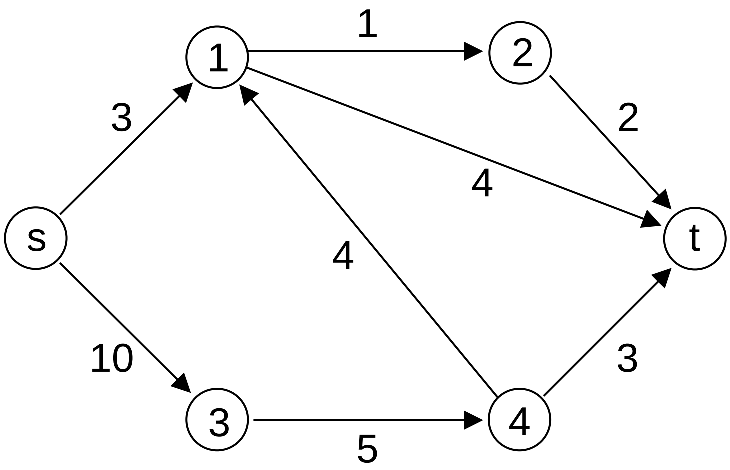

Figure 8: A network.

An actual flow of the fluid through the network can be described by introducing another labeling $f : V \times V \to \mathcal{R}$. A flow through an edge is always bounded by its capacity. For technical reasons, negative values of the flow are allowed as well. A negative value $f(i, j) < 0$ means a positive flow from node i to node j and vice versa. Furthermore, the flow is conserved in all nodes, except the source and the sink. In total we obtain

- Capacity constraint: $f(i, j) < c((i, j))$ for all $i, j \in V$
- Negative flows: $f(i, j) = -f(j, i)$ for all $i, j \in V$
- Flow conservation: $\sum_j f(i, j) = 0$ for all $i \in V$ with $i \neq s, t$

Please note that by combining the first and second properties $f(j, i) > -c((i, j))$ is obtained. The total flow through the graph is given by $f_0 = \sum_j f(s, j) = -\sum_j f(t, j)$. Determining the maximum possible amount which can flow from s to t through a network, called the *maximum flow*, is an optimization problem. E.g., the maximum flow for the network in Fig. 8 is eight. Later we will see that the determination of the maximum flow can be solved in polynomial time. In physics, we will apply these methods to calculate ground states of disordered magnetic materials like random field systems or diluted antiferromagnets.
The *minimum cost flow* problem is related to the maximum-flow problem. Here, additionally costs $c' : E \to \mathcal{R}$ and a total flow f_0 are given. $c'(e)$ specifies the costs of sending one unit of flow through edge e. More generaly, the costs may depend arbitrarily but in a convex way on the flow through an edge. The minimum cost flow problem is to find a flow of minimum total cost $\sum_e c'(e) f(e)$, which is compatible with the capacity/ conservation constraints and has the total value f_0. Later it will be shown how ground states of solid-on-solid systems or of vortex

glasses (from the theory of superconductivity) can be obtained by minimizing cost flows. For this problem polynomial time algorithms are known as well. The case $f_0 \equiv 0$ occurs very often. Please note that the optimum flow may not vanish inside the network in this case as well, since the costs can take negative values thus making circular flows favorable.

4 Graph representations

If we want to apply graphs on a computer, we have to represent them somehow in the memory. Different ways of storing graphs are known. It depends on the kind of application which choice is the most efficient. We start with the storage of lists, since they are used for the representations of the other graph types. Then trees are discussed. At the end of the section two methods of representating arbitrary graphs are shown.
Since lists are just linear sequences of elements, the straight forward implementation uses arrays. A list of n elements can be stored in array $L[1 \ldots n]$ e.g. by storing the head in $L[1]$, its son in $L[2]$, etc, and the tail in $L[n]$. The advantage of this realization is its simplicity. Furthermore, it is possible to access all elements of the list in constant time. A minor problem is that one has to know the maximum size of the list when allocating the memory for the array. But, if the maximum size is not known, one can reallocate the memory if the list grows to long. If the reallocation occurs rarely, the additional computer time needed can be neglected.

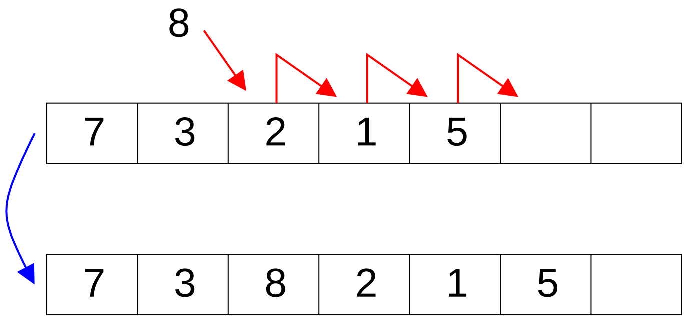

Figure 9: Element 8 is inserted into an array.

One major problem is the following: if elements are to be inserted or to be removed somewhere within the list, large chunks of memory have to be moved around each time the operation occurs (see Fig. 9). Thus, insert and delete operations take $\Theta(n)$ time. For stacks and queues this problem does not occur since elements are inserted and deleted only at the head or at the tail. Consequently, stacks and queues are usually realized by arrays. A stack is stored usually in reversed order in an array, adding and removing elements occurs at the end of

the stack which is given by a pointer variable. Please remeber that a pointer is just a memory address. For a queue two pointers are necessary: one which stores the head and one which holds the tail of the queue. If a pointer reaches the end of the array it is set to the beginning again.
An implementation of general lists which is more flexible uses pointers as well. For each element of a list, a small chunk of memory is allocated separately. In addition to the information associated with each element, a pointer which contains the address of the following element is stored. At the tail of the list the pointer has the special value NULL. This realization is reflected directly by the manner lists are drawn: the arrows represent the pointers. If the position where an operation is to be performed is given already, insert and delete operations can be performed in constant time just by (re-)assigning some pointers. One has to be a little bit careful to loose no parts of the chain, for details see e.g. [10]. The insert and delete operations become especially simple if along each element *two* pointers are stored: one to the next element and one to the preceding element. In this case the list is called *double linked*. A drawback of the realization with pointers is that it is not possible to access elements of the list (say: the 20th element) directly. One has to start at the head or the tail and go through the list until the element is reached. Furthermore, this type of realization consumes more memory, since not only the elements but the pointers have to be stored as well.
The situation which we encounter here is typical for computer science. *There are now perfect data structures*. On has to take the application into account to find the best realization.
This is true for trees as well. If the maximum number of sons for each node is known, usually the following type of realization is used, which is similar to the representation of lists via pointers. Along with the information which is to be stored in a node, an array containing pointers to the sons is kept. If a node has less than the maximum number of sons, some pointer values contain the value NULL. In Fig. 10 a realization of the tree from Fig. 3 is shown.
For binary trees there is a very simple realization using one array T. The basic idea is to start at the root an go through each level from left to right. This gives us the order in which nodes are stored in the array. One obtains:

- The root is stored in $T[1]$
- If a node is empty, a special value (say -1) is assigned.
- For each node $T[k]$ the left son is stored in $T[2k]$ and the right son is kept in $T[2k+1]$

In Fig. 11 the array representation of the search tree from Fig. 4 is shown. This kind of tree realization is comparable to the array representation of lists: It is very easy to built up the structure and to access nodes but insert and delete operations are time consuming.

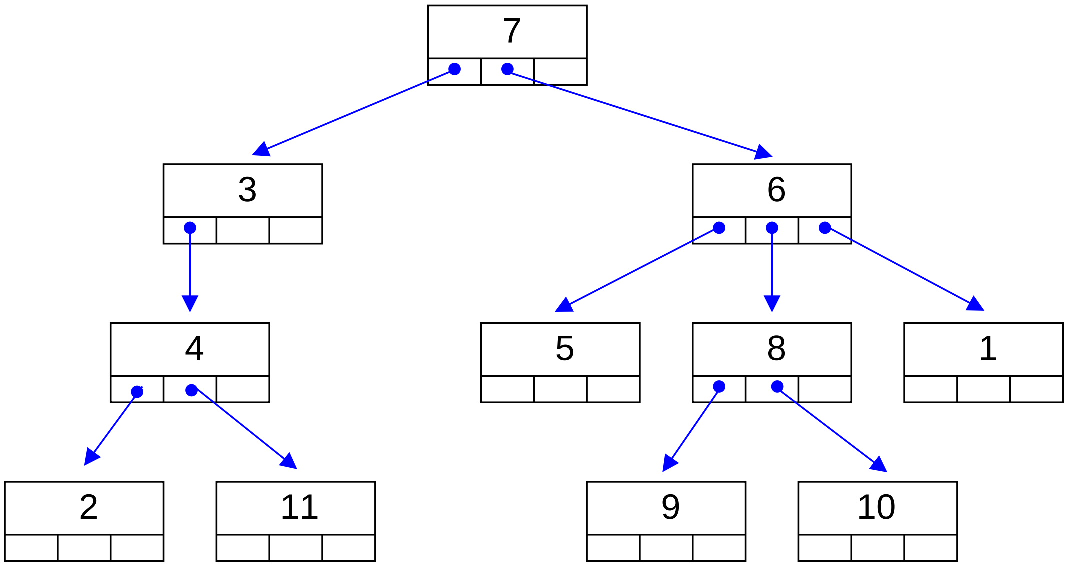

Figure 10: Representation of the tree from Fig. 3 via nodes containing arrays of pointers. Here, each array contains at most three entries per node.

We finish this section by explaining two methods to represent arbitrary graphs. Let $G = (V, E)$ be a graph. Without loss of generality we assume $V = \{1, 2, \ldots, n\}$. An *adjacency matrix* $\underline{\underline{A}} = (a_{ij})$ for G is a $n \times n$ matrix with entries $0, 1$, where n is the number of nodes. In the case G is directed we have:

$$a_{ij} = \begin{cases} 1 & \text{for} \quad (i,j) \in E \\ 0 & \text{else} \end{cases} \tag{1}$$

For an undirected graph, the adjacency matrix $\underline{\underline{A}}$ contains nonzero elements $a_{ij} = a_{ji} = 1$ for each edge $(i, j) \in E$, i.e. in this case the matrix is symmetric.
The adjacency matrices for the example graphs of section 1 are shown in Fig. 12. The advantage of the matrix representation is that one can access each edge within a constant amount of computer time. On the other hand, an adjacency matrix needs $\Theta(n^2)$ memory elements to be stored. Consequently, this choice is inefficient for storing *sparse* graphs, where the number of edges is $m \in \Theta(n)$.
For sparse graphs, it is more efficient to use *adjacency lists*: For each node i a list L_i is stored which contains all its neighbors $N(i)$. Hence, for directed graphs, L_i contains all vertices j with $(i, j) \in E$ while for an undirected graph it contains all vertices j with $(i, j) \in E$ or $(j, i) \in E$. Please note that the elements in the list can be in arbitrary order. The list representation uses only $O(m)$ memory elements. With this realization it is not possible to access an edge (i, j) directly, since one has to scan trough all elements of L_i to find it. But for most applications a direct access is not necessary, so this realization of graphs is widely used. A similar method is applied for the LEDA-library package [8].

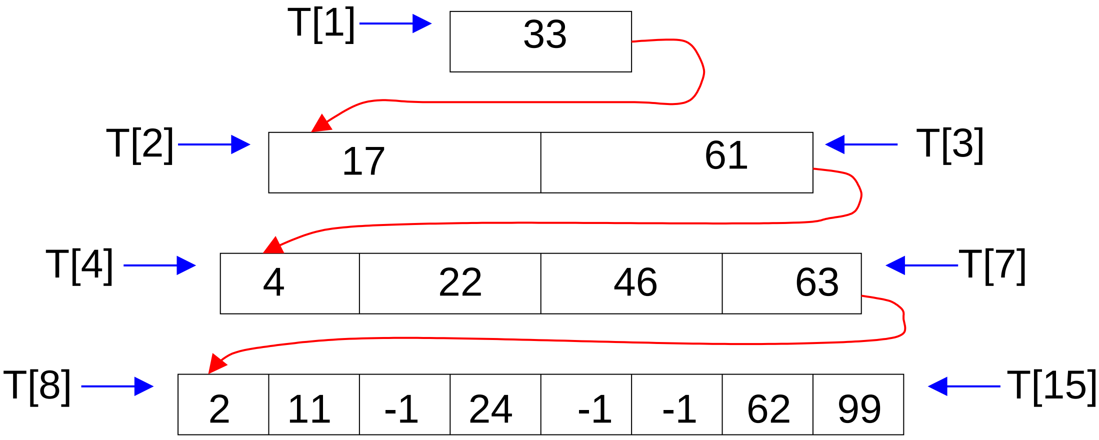

Figure 11: The representation of a binary tree via an array.

$$\begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

undirected directed

Figure 12: Adjacency matrices for graph $G = (\{1,2,3,4,5,6\}, \{(1,3), (3,4)\ (4,1), (4,2), (6,1)\})$. The left matrix shows the case when G is regarded as undirected graph, while the right one is for the case of a directed graph.

The adjacency lists for the sample graphs from above are shown in Fig. 13
For a directed graph, it is sometimes very convenient to have all incoming edges for a given vertex j available. Then one can store a second set of lists K_j containing all nodes i with $(i,j) \in E$. The Lists K_j for the directed version of the sample graph are shown in Fig. 14.
The methods presented above can be extended easily to implement labeled graphs. Labels of vertices can be represented by arrays. For labels of edges one can either use matrices or one can store the labels along with the list elements representing the edges.

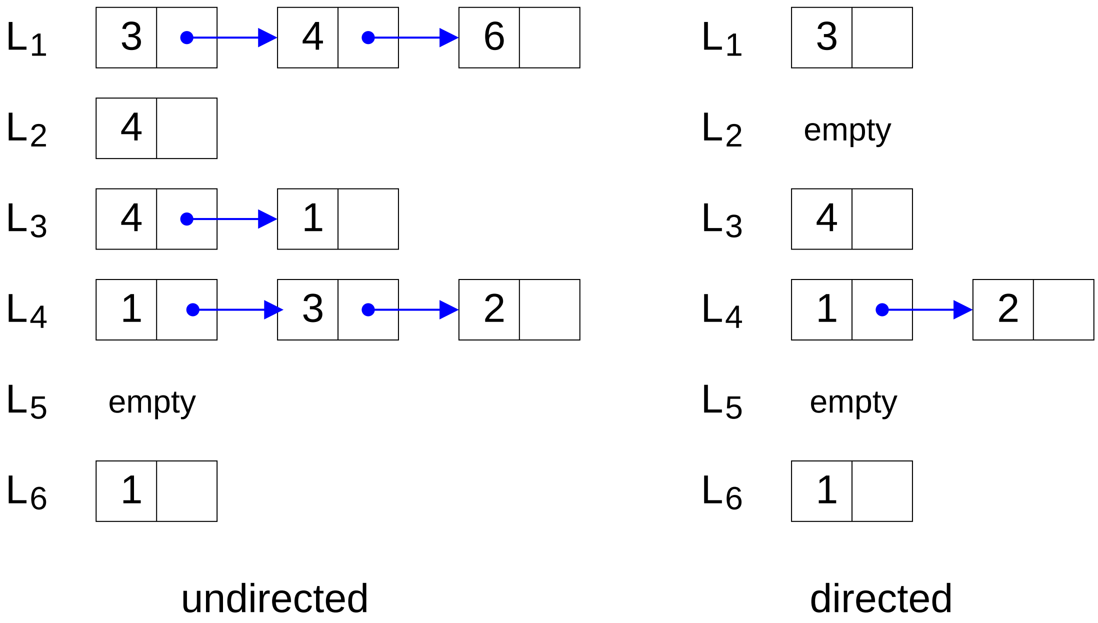

Figure 13: Adjacency lists for the example graphs. The left lists represents the case when G is regarded as undirected graph, while the right are for the case of a directed graph.

5 Simple graph algorithms

Most basic algorithms on graphs, trees and lists can be found in [1, 10]. Here we present just one algorithm for searching in a tree and one for finding the connected components of a graph. This should be sufficient to make the reader familiar with the subject.
Consider a search tree, as introduced in section 2. One basic operation is the *find* operation. It tests whether an object O_1 is contained in the tree or not. The algorithm starts at the root of the tree. If the tree is empty, then the object is not contained in the tree. If O_1 is stored at the root it is found. In both cases the algorithm terminates. Otherwise, if O_1 is smaller than the object at the root, the search continues in the left subtree. If O_1 is larger, the search continues in the right subtree. The recursive formulation of the algorithm reads as follows:

algorithm find(tree, object)
begin
    if tree is empty **then**
        return(not found)
    if root contains object **then**
        return(tree)
    if object at root-node $\geq$ object **then**

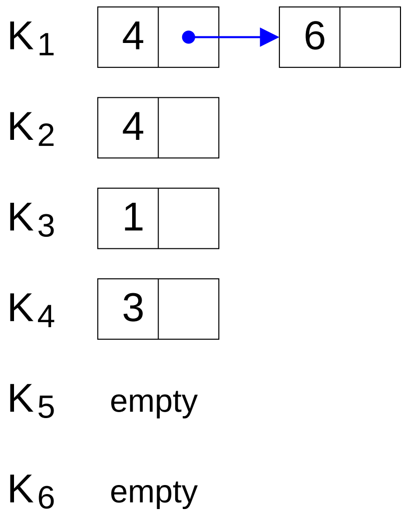

Figure 14: List of incoming edges for directed example graph.

```
        find(left subtree, object)
    else
        find(right subtree, object)
end
```

Example: Search in tree
We trace the search for element 24 in the tree shown in Fig. 15. The algorithm is called with the root of the tree and number 24. Since the tree is not empty and 24 is smaller than the object at the root (33), a recursive call with the left subtree occurs. Here, number 17 is stored at the root. Thus, the procedure is called with the right subtree of the first subtree. In the next step again the search continues in the right subtree where finally the object is found. □

The algorithm performs one descent into the tree. Hence, its time-complexity is $\Theta(h)$ if h is the height of the search tree. If the search tree is complete, the height is $h \in \Theta(\log n)$, where n is the number of elements in the tree.
The find algorithm can be directly extended to insert elements into the tree: if the algorithm reaches an empty subtree, then the element is not stored in the tree. Thus, a node containing the object can added at the position where the search has stopped. Consider as example that number 21 is to be inserted into the tree from Fig. 15. At node 22 the algorithm branches to the left subtree, but

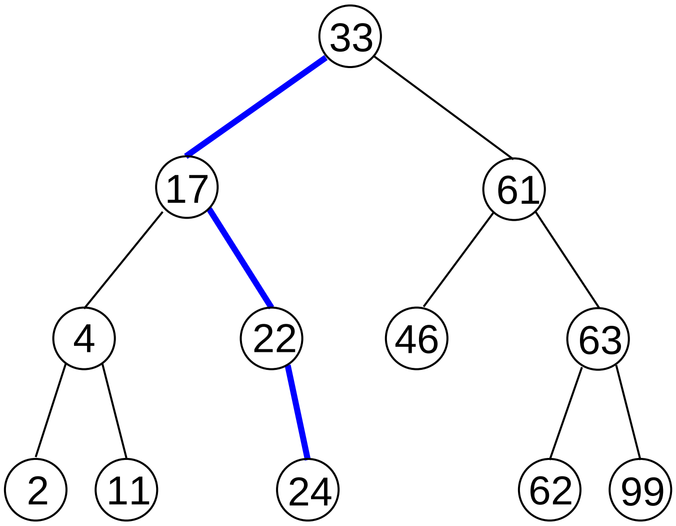

Figure 15: Search for element 24 in a sorted binary tree.

this tree is empty. Therefore, a node with the number 21 will be added as left son of the node 22.

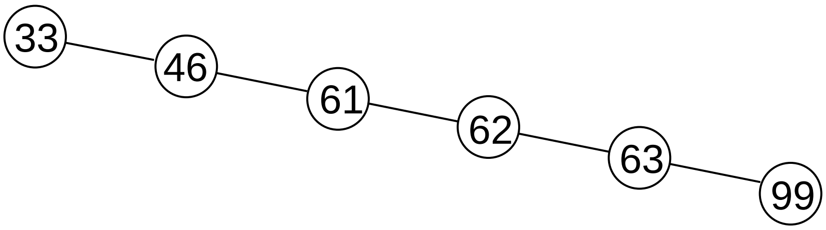

Figure 16: A binary search tree built by inserting numbers 33, 46, 61, 62, 63, 99 in the given order using a simple insert operation.

This simple algorithm for inserting new objects has a major drawback. Assume that numbers sorted in increasing order are inserted into a tree which is empty at the beginning. During the execution of the algorithm always the right subtree is taken. Consequently, the tree which is built in this way is in fact a list (see Fig. 16). In this case the height of the tree is equal to the number of elements, i.e. the search for an object takes $\Theta(n)$ time instead of logarithmic time.
In order to avoid this trouble one can keep the tree always *balanced*. This means that for every node the heights of the left and right subtree differ at most by one. There are several clever algorithms which assure that a search tree remains balanced. The main idea is to reorganize the tree in case of imbalance while keeping the sorted order. More details can be found in [1, 10]. For balanced trees, it is guaranteed that the height of the tree grows only logarithmically with

the number of nodes n. Thus, all operations like insert, search and delete can be realized in $\Theta(\log n)$.
Next, an algorithm for calculating the connected components of an undirected graph $G = (V, E)$ is presented. This algorithm has many applications in physics, as we will see later. E.g., the connected component corresponds to a cluster in percolation theory or to a domain in a magnetic material.
The main work of the algorithm is done within the procedure depth-first(), which starts at a given vertex i and visits all vertices which are connected to i. The main idea is to perform recursive calls of depth-first() for all neighbors of i which have not been visited at the moment. The array comp[] is used to keep track of the process. If comp[i] = 0 vertex i has not been visited yet. Otherwise it contains the number t of the component. Please note that the array comp[] is passed as reference, i.e. it behaves like a global variable.

procedure depth-first(G, i, **comp**, t)
begin
  comp[i]=t
  for all neighbors j of i **do**
    if comp[j] = 0 **then**
      depth-first(G, j, comp, t)
end

algorithm components(G)
begin
  initialize comp[i] := 0 for all $i \in V$
  $t := 1$
  while there is a node i with comp[i]=0 **do**
    depth-first(G, i, comp, t); t:=t+1;
end

In Fig. 17 an example graph is shown. The numbers next to the nodes indicate a possible order in which the nodes are visited. The edges which are used by the algorithm are indicated by thick lines. Since each node is visited not more than once, there are no cycles in the subgraph which contains only these thick edges. Thus, they constitute a tree for each component. Since all nodes are visited, the tree *spans* the whole component. For this reason the tree is called *spanning tree*. The collection of all spanning trees of a graph is called its spanning *forest*.
Please note that the algorithm tries to go as far as possible within a graph, before visiting other neighbors of the node where the search started. For this reason the procedure has its name. The spanning trees constructed by this procedure are also called *depth-first spanning trees*. A similar algorithm, which instead first

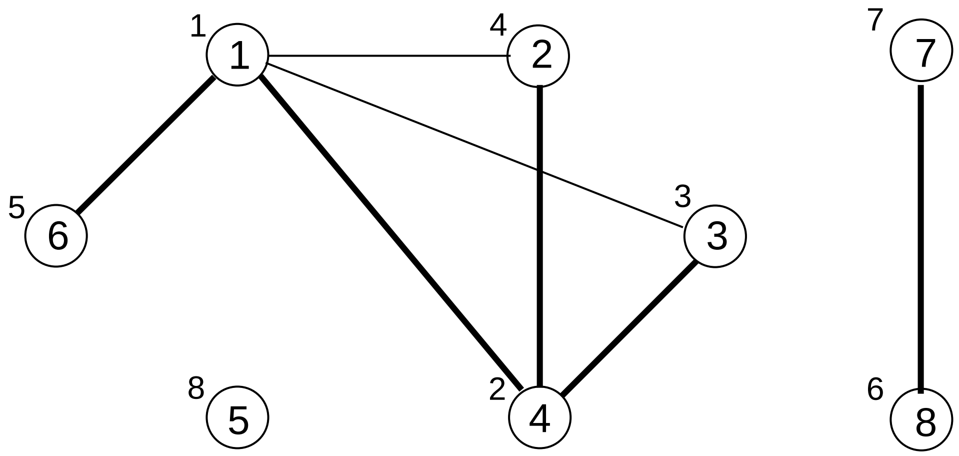

Figure 17: Depth-first spanning forest for a graph with three connected components.

visits all neighbors of a node before proceeding with nodes farer away, is called "breadth-first". In the example, the edge $(1, 2)$ would be included in the spanning tree, if it was constructed using a breadth-first search.

6 NP-complete graph problems

The time complexities of the algorithms presented in the last chapter are all bounded by a polynomial. But there are many hard graph problems as well. Interestingly, there are many pairs of similar problems where one problem is in P while the other in NP. Some of them will be presented now.
The algorithms presented in the last chapter are very simple. Usually the algorithms for solving graph problems, even for the problems in P, are very complicated. Thus, in this section no algorithms will be presented, only some basic graph problems are explained. For the proofs that certain problems are NP-complete the reader is referred to the literature as well. But later on, for the optimization problems we encounter here, all necessary algorithms will be discussed in detail.
The first example is about cycles. Let $G = (V, E)$ be an undirected graph. An *Euler cycle* is a cycle which contains all *edges* of the graph exactly once. It was shown by Euler in 1736 that an undirected graph contains an Euler cycle if and only if each vertex has an even degree. Thus, the recognition problem, whether an undirected graph has an Euler cycle or not (EC), can be decided in polynomial time. The first algorithm for constructing an Euler cycle was presented 1873 in [6]. An *Hamilton cycle* is a cycle which contains each *vertex* exactly once. The problem whether an undirected graph contains a Hamilton cycle or not (HC) is NP-complete. A proof can be found in [9], the main idea is to show 3-SAT$\leq_p$HC.

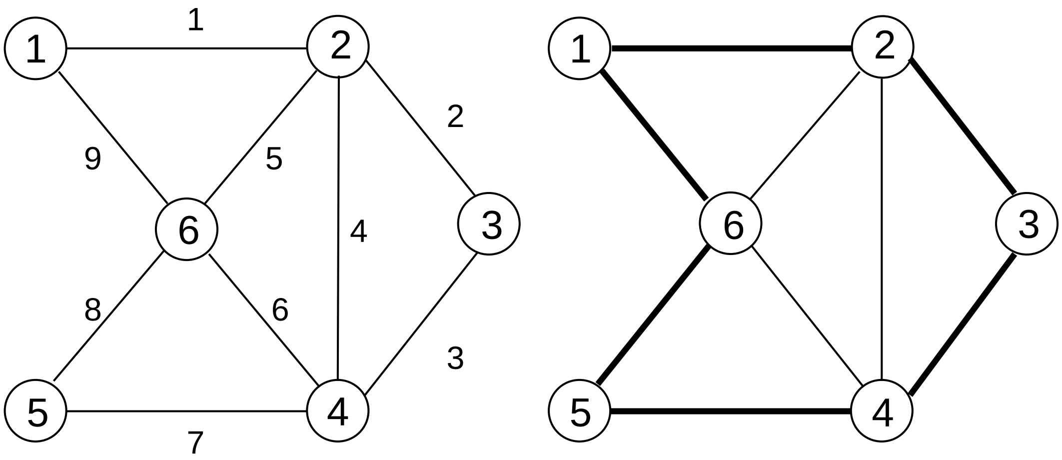

Figure 18: An undirected graph containing a Euler cycle (left) and a Hamilton cycle (right). The numbers next to the edges give one possible order of the edges within an Euler cycle.

In Fig. 18 a graph containing both an Euler Cycle and a Hamilton cycle is shown.

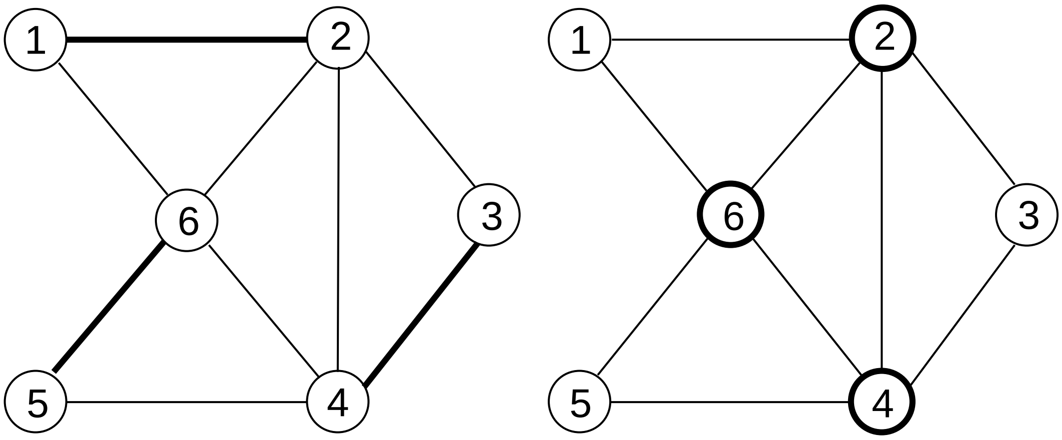

Figure 19: A graph and a minimum edge cover (left) and a minimum vertex cover (right). The edges/vertices belonging to the cover are shown in bold.

The next pair of graph problems is defined via *covers* of undirected graphs. An *edge cover* is a subset $E' \subset E$ of edges such that each vertex is contained in at least one edge $e \in E'$. Each graph which has no isolated vertices has an edge cover, since in that case E itself is an edge cover. A *minimum edge cover* is an edge cover where $|E'|$ is minimal. In Fig. 19 a graph and a minimum edge cover are shown. An algorithm which constructs a minimum edge cover in polynomial time can be found in [7]. Consequently, the corresponding recognition problem

"does graph G have an edge cover with $|E'| \leq K$" is in P. Related to the edge cover is the *vertex cover*. It is a subset $V' \subset V$ of vertices such that each edge contains at least one node from V'. In Fig. 19 a graph together with is minimum vertex cover is displayed. It has been shown in [5] that the problem "does graph G have an vertex cover with $|V'| \leq K$" (VC) belongs to the class of NP-complete problem. The proof by transforming 3-SAT$\leq_p$VC. Thus, there is no algorithm known which finds a minimum vertex cover in polynomial time.
We have already seen that the TSP is a hard optimization problem. The corresponding decision problem is NP-complete [5]. The proof works by demonstrating HC$\leq_p$TSP. HC has already been recognized as being NP-complete (see above). The proof is short, which is quite unusual for this kind of problems.
Proof: Let $G = (V, E)$ be a graph, $n = |V|$. Then we built a second graph $G' = (V, V \times V)$ with the following distance labeling

$$d(i,j) = \begin{cases} 1 & \text{if}(i,j) \in E \\ 2 & \text{else} \end{cases} \tag{2}$$

The transformation works in polynomial time since it contains just two nested loops over all nodes. Each tour in G' contains exactly n edges and visits all n nodes. Thus, a tour must have the length n at least. If it has exactly length n all distances along the tour must have value 1, i.e. all edges from the tour are in G as well, and vice versa. Consequently, the question "does G' have a shortest round tour with length $\leq n$" has the answer "yes" if and only if G has a Hamilton cycle. Hence, HC$\leq_p$TSP. □
On the other hand, if one is interested just in the shortest distance between two given cites, this problem can be solved in polynomial time. Several algorithms can be found in [2]. Some of them will be presented later on. A recent application in physics is the study of so called *small world networks*.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman: The design and analysis of computer algorithms, Addison-Wesley, Readin (MA) 1974

[2] D.P. Bertsekas: Network Optimization, Continuous and Discrete Models, Athena Scientific, Belmont (MA) 1998

[3] B. Bolobas: Modern Graph Theory, Springer, New York 1998

[4] J.D. Claiborne, Mathematical Preliminaries for Computer Networking, Wiley, New York, 1990.

[5] M.R. Garey and D.S. Johnson: Computer and Intractability, A Guide to the Theory of NP-Completeness, W.H. Freeman & Company 1979

[6] C. Hierholzer, Math. Ann. **6**, 30-32 (1873)

[7] E.L. Lawler: Combinatorial optimization: Networks and Matroids, Holt, Rinehard and Winston, New York 1976

[8] K. Mehlhorn, S. Näher and C. Uhrig: LEDA, http://www.mpi-sb.mpg.de/LEDA/leda.html

[9] C.H. Papadimitriou and K. Steiglitz: Combinatorial Optimization, Dover Publications, Mineola (NY) 1998

[10] R. Sedgewick: Algorithms in C, Addison-Wesley, Readin (MA) 1990

[11] M.N.S. Swamy and K. Thulasiraman, Graphs, Networks and Algorithms, Wiley, New York 1991.