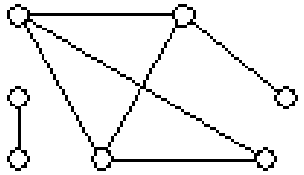
12 Graph algorithms

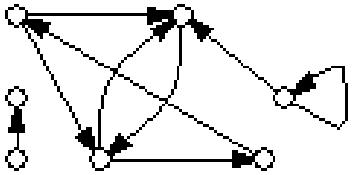
This chapter discusses the data structure that is a collection of points (called nodes or vertices) and connections between them (called edges or arcs) – a graph.

The common search algorithms related to graphs are discussed.

A graph is a data structure that consists of *node*s (or *vertex*), and *edge*s (or *arc*) that link the nodes to each other.

A graph can be *undirected* or *directed*.





Graphs play a very important role in computer science.

- Diagrams can often be seen as graphs
- relations between things can often be represented with graphs
- many problems can be turned into graph problems
  - search for direct and indirect prerequisites for a course
  - finding the shortest path on a map
  - determining the capacity of a road network when there are several alternate routes

12.1 Representation of graphs in a computer

In mathematics a graph G is a pair $G = (V, E)$.

- V = set of vertices
- E = set of edges
- thus there can be only one edge to each direction between vertices
  - this isn't always enough in a practical application
  - for example, there can be more than one train connection between two cities
  - this kind of graph is called a *multigraph*

- if only one edge to each direction between nodes is possible $\Rightarrow E \subseteq V^2$
  - for a directed graph $|E|$ can alter between $0, \ldots, |V|^2$
  - this is assummed when analyzing the efficiencies of graph algorithms

The efficiency of a graph algorithm is usually given as a function of both $|V|$ and $|E|$

- we're going to leave the absolute value signs out for simplicity inside the asymptotic notations, i.e. $O(VE) = O(|V| \cdot |E|)$

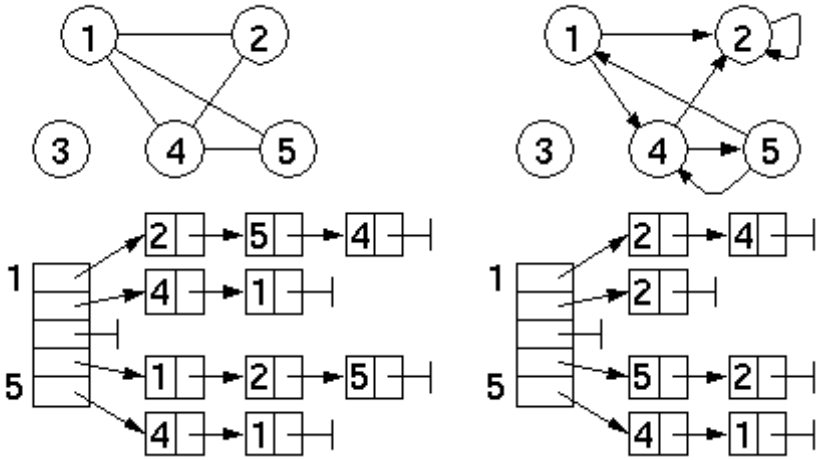
There are two standard ways to represent a graph in a computer *adjacency list* and *adjacency matrix*.

The adjacency list representation is more commonly used and we concentrate on it on this course.

- Nodes are stored in some data structures (depending on what other data nodes contain, how they have to be searched, inserted, etc.)
- In the simplest form every node has a data structures, which stores info on nodes that this node has an edge to.
  - The choice of data structure depends on estimated number of edges, whether they will be inserted/deleted often, if a certain edge has to be found quickly, etc.)
  - The edge's target node can be stored as a pointer, a node index (if nodes are in an indexable structure), etc.
  - The order of the nodes in the adjacency list is typically irrelevant

- The sum of the sizes of all adjacency lists of the graph is
  - $|E|$, if the graph is directed, $2 \cdot |E|$, if the graph is undirected

  $\Rightarrow$ whole memory consumption is $O(\max(V, E)) = O(V + E)$



- The search for "is there an edge from vertex v to u" requires a scan through one adjacency list which is $\Theta(V)$ in the worst case (unless edges are stored in a data structure which provides quick search based on the target node)
- If edges are weighted or have other extra data, that must be stored in the adjacenvy list also with the target node (for example a struct, with the target node and other data)
- Sometimes it's needed to store also info on *incoming* edges of a node (to help removing nodes and edges, for example). That can be done in the same manner as outgoing edges.

The question abowe can easily be answered with the adjacency matrix representation

- the adjacency matrix is a $|V| \times |V|$ -matrix A, where the element a_{ij} is
  - 0, if there is no edge from vertex i to j
  - 1, if there is an edge from i to j
- the adjacency matrices of the earlier example are

	1	2	3	4	5
1	0	1	0	1	1
2	1	0	0	1	0
3	0	0	0	0	0
4	1	1	0	0	1
5	1	0	0	1	0

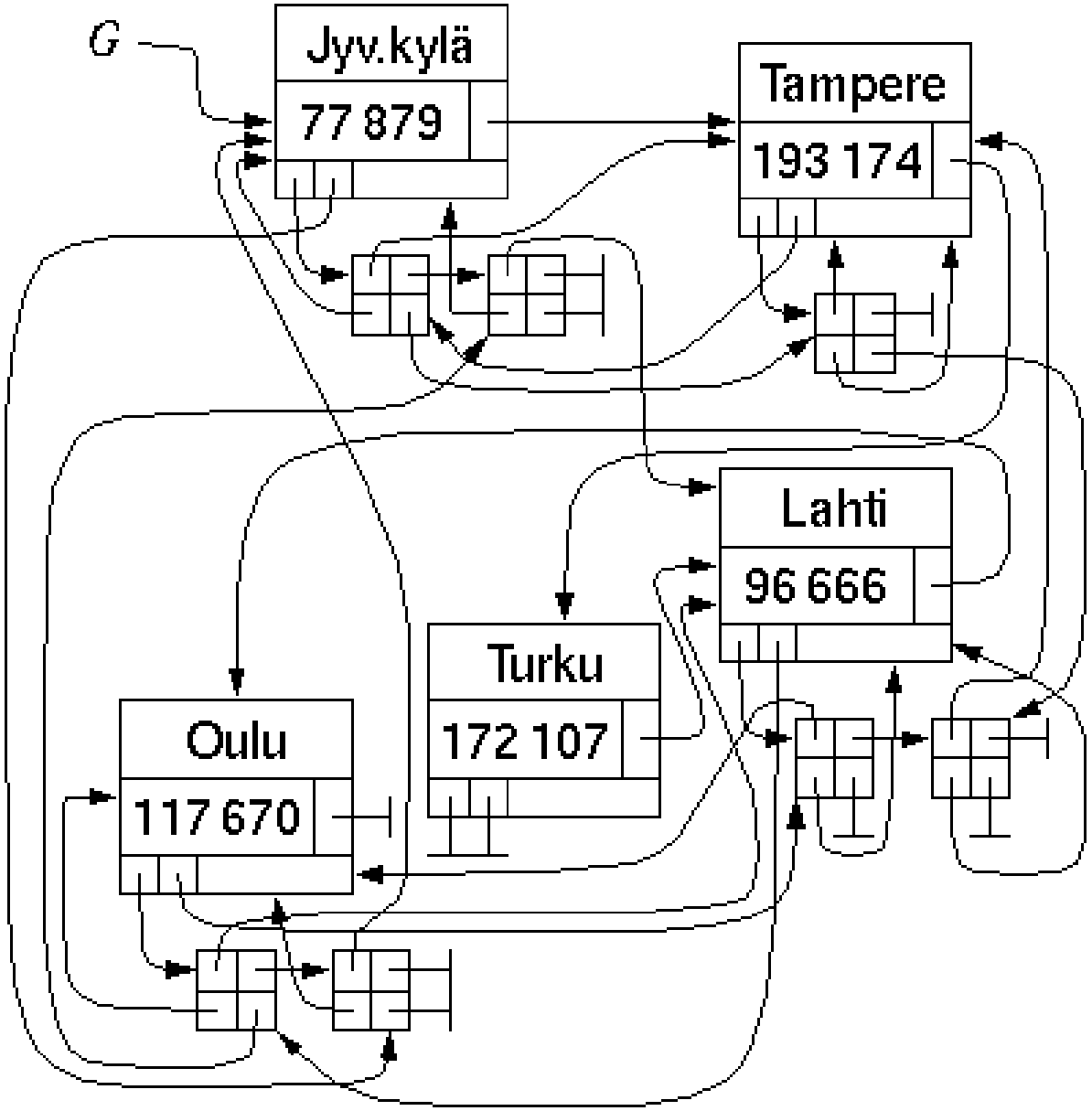
	1	2	3	4	5
1	0	1	0	1	0
2	0	1	0	0	0
3	0	0	0	0	0
4	0	1	0	0	1
5	1	0	0	1	0

- memory consumption is always $\Theta(V^2)$
  - Each element uses only a bit of memory, to several elements can be store in on word $\Rightarrow$ the constant coefficient can be made quite small
- the adjacency matrix representation should be used with very dense graphs

Let's analyze the implementation of the adjacency list representation a little closer:

- In practical solutions all kinds of information usefull to the problem or the algorithm used is collected to the vertices
  - name
  - a bit indicating whether the vertex has been visited
  - a pointer that indicates the vertex through which this vertex was entered

  $\Rightarrow$ the vertex should be implemented as a structure with the necessary member fields
- usually it's beneficial to make the vertices structures which have been linked to the vertices that lead to them
- the main principle:
  - store everything once
  - use pointers to be able to travel to the necessary directions


G
Jyv.kylä
77 879
Tampere
193 174
Lahti
96 666
Turku
172 107
Oulu
117 670

12.2 General information on graph algorithms

Terminology:

- step = moving from one vertex to another along an edge
  - the step needs to be taken to the direction of the edge in a directed graph
- the *distance* of the vertex v_2 from the vertex v_1 is the length of the shortest path from v_1 tov_2
  - the distance of each vertex from itself is 0
  - denoted by $\delta(v_1, v_2)$
  - it is possible (and common) in a directed graph that $\delta(v_1, v_2) \neq \delta(v_2, v_1)$
  - if there is no path from v_1 to v_2 then $\delta(v_1, v_2) = \infty$

To make understanding the algorithms easier, we'll color the vertices.

- white = the vertex hasn't been discovered
- grey = the vertex has been discovered but hasn't been completely processed
- black = the has been discovered and is completely processed
- the color of the node changes from white $\rightarrow$ grey $\rightarrow$ black
- the color coding is a tool for thinking and it doesn't need to be implemented fully. Usually it is sufficient to know whether the node has been discovered or not.
  - of this information can also be determined from other fields

Many graph algorithms go through the graph or a part of it in a certain order.

- there are two basic ways to go through the graph: breadth-first search and depth-first search
- an algorithm that
  - visits once all vertices in the graph or some part of it
  - travels through each edge in the graph or some part of it once

  is meant by "going through"

The search algorithms use some given vertex of the graph, the *source*, as the starting point of the search and search through all nodes that can be accessed through the source with 0 or more steps.

12.3 Breadth-first search

The breadth-first search can be used for example for:

- determining the distance of all the nodes from the source
- finindh (one) shortest path from the source to each node

The breadth-first is so named as it investigates the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.

The fields of the vertices:

- $v \rightarrow d$ = if the vertex v has been discovered its distance from s, else ∞
- $v \rightarrow \pi$ = a pointer to the vertex through which v was found the first time, NIL for undiscovered vertices
- $v \rightarrow colour$ = the color of vertex v
- $v \rightarrow Adj$ = the set of the neighbours of v

The data structure Q used by the algorithm is a queue (follows the FIFO policy).

```
BFS(s)                                      (the algorithm gets the source s as a parameter)
▷ in the beginning the fields of each vertex are colour = WHITE, d = ∞, π = NIL
s→colour := GRAY                            (mark the source as discovered)
s→d := 0                                    (the distance of the source from the source is 0)
PUSH(Q, s)                                  (push the source to the queue)
while Q ≠ ∅ do                              (repeat while there are vertices)
    u := POP(Q)                             (take the next vertex from the queue)
    for each v ∈ u→Adj do                   (go through the neighbours of u)
        if v→colour = WHITE then            (if the vertex hasn´t been discovered . . . )
            v→colour := GRAY                (. . . mark is as discovered)
            v→d := u→d + 1                  (increment the distance by one)
            v→π := u                        (vertex v was reached through u)
            PUSH(Q, v)                      (push the vertex into the queue to be processed)
    u→colour := BLACK                       (mark the vertex u as processed)
```

All fields of the vertex used by the algorithm are not necessarily needed in the practical implementaion. Some can be determined based on each other.

The running-time in relation to the amount of vertices (V) and edges (E):

- before calling the algorithm the nodes need to be initialized
  - this can be done in $O(V)$ in a sensible solution
- the algorithm scans the out edges of the vertex on line 7
  - can be done in linear time to the amount of the edges of the vertex with the adjacency list representation
- each queue operation is constant time
- the amount of loops in the while
  - only white vertices are pushed to the queue
  - the color of the vertex is changed into gray at the same time

    $\Rightarrow$ each vertex can be pushed into the queue atmost once

  $\Rightarrow$ the while-loop makes atmost $O(V)$ rounds
- the amount of loops in the for

  - the algorithm goes through each edge once into both directions

  $\Rightarrow$ for-loop is executed atmost $O(E)$ times in total

$\Rightarrow$ the running-time of the entire algorithm is thus $O(V + E)$

Once the algorithm has ended the π pointers define a tree that containes the discovered vertices with the source s as its root.

- *breadth-first tree*
- π pointers define the edges of the tree "backwards"
  - point towards the root
  - $v \rightarrow \pi$ = v's *predecessor*, i.e. *parent*
- all nodes reachable from the source belong into the tree
- the paths in the tree are the shortest possible paths from s to the discovered vertices

Printing the shortest path

- once BFS has set the π pointers the shortest path from the source s to the vertex v can be printed with:

```
PRINT-PATH(G, s, v)
1  if v = s then                    (base case of recursion)
2      print s
3  else if v→π = NIL then           (the search didn't reach the vertex v at all)
4      print "no path"
5  else
6      PRINT-PATH(G, s, v→π)        (recursive call . . . )
7      print v                      (. . . print afterwards)
```

- A non-recursive version can be implemented for example by
  - collecting the numbers of the vertices into an array by walking through the π pointers and printing the contents of the array backwards
  - walking through the path twice and turning the π pointers backwards each time (the latter turning is not necessary if the π pointers can be corrupted)

12.4 Depth-first search

Depth-first is the second of the two basic processing orders.

Where as the breadth-first search investigates the vertices across the entire breadth of the search frontier, the depth-first search travels one path forwards as long it's possible.

- only vertices that haven't been seen before are accepted into the path
- once the algorithm cannot go any further, it backtracks only as much as is needed in order to find a new route forwards
- the algorithm stops once it backtracks back to the source and there are no unexplored edges left there

The pseudocode for the algorithm resembles greatly the breadth-first search. There are only a few significant differences:

- instead of a queue the vertices waiting to be processed are put into a stack
- the algorithm doesn't find the shortest paths but a path

– for this reason the example pseudocode has been simplified by leaving out the π-fields

The data structure S used by the algorithm is a stack (follows the LIFO-policy).

```
 DFS(s)                                          (the algorithm gets the source s as a parameter)
1   ▷ in the beginning the color field of eack (unprocessed) vertex is colour = WHITE
2   PUSH(S, s)                                  (push the source into the stack)
3   while S ≠ ∅ do                              (continue until the stack is empty)
4       u := POP(S)                             (pop the latest vertex added to the stack)
5       if u→colour = WHITE then                (if the vertex hasn't been processed . . . )
6           u→colour := GRAY                    (mark the state as discovered)
7           PUSH(S, u)                          (push again into stack (for marking black))
8           for each v ∈ u→Adj do               (go through the neighbours of u)
9               if v→colour = WHITE then        (if the vertex hasn't been processed . . . )
10                  PUSH(S, v)                  (. . . push it into the stack for prosessing)
11              else if v→colour = GRAY then    (gray node! Cycle found! . . . )
12                  ????                        (handle the cycle, if you are interested in it)
13      else
14          u→colour := BLACK                   (all neigbours handled, node is finished)
```

If the entire graph needs to be investigated, the depth-first search can be called for all nodes still unprocessed.

- this time the nodes are not colored white between the calls

An operation that needs to be done to all vertices in the graph could be added after line 5. We can for example

- investigate if the vertex is a goal vertex and quit if so
- store satellite data associated with the vertex
- modify the satellita data

The efficiency can be analyzed as with breadth-first search:

- before calling the algorithm the nodes need to be initialized
  - this can be done in $O(V)$ in a sensible solution
- the algorithm scans the out edges of the vertex on line 6
  - can be done in linear time to the amount of the edges of the vertex with the adjacency list representation
- each stack operation is constant time

- the amount of loops in the while
  - only white vertices are pushed into the stack
  - the color of the vertex is changed into gray at the same time
    $\Rightarrow$ each vertex can be pushed into the stack atmost once

  $\Rightarrow$ the while-loop makes atmost $O(V)$ rounds
- the amount of loops in the for
  - the algorithm goes through each edge once into both directions

  $\Rightarrow$ for-loop is executed atmost $O(E)$ times in total

$\Rightarrow$ the running-time of the entire algorithm is thus $O(V + E)$

The algorithm can also be implemented recursively, in which case the function call stack takes the role of the stack in the algorithm.

- The recursive version is actually somewhat simpler than the iterative version!
- (Can you notice why?)

Note! before calling the algorithm all vertices must be initialized white!

```
 DFS(u)
1   u→colour := GRAY                    (mark vertex as found)
2   for each v ∈ u→Adj do               (go through the neighbours of u)
3       if v→colour = WHITE then        (if v hasn't been visited ...)
4           DFS(v)                      (... continue the search recursively from v)
5       else if v→colour = GRAY then    (if has been visited but not fully processed ...)
6           ▷ a loop is found           (... a loop is found)
7   u→colour := BLACK                   (mark node as fully processed)
```

Running-time:

- the recursive call is done only with white vertices
- a vertex is colored grey at the beginning of the function
  $\Rightarrow$ DFS is called atmost $O(V)$
- like in the earlier version the for loop makes atmost two rounds per each edge of the graph during the execution of the entire algorithm
  $\Rightarrow$ there are atmost $O(E)$ rounds of the for loop
- other operations are constant time

$\Rightarrow$ the running time of the entire algorithm is still $O(V + E)$

Breadth-first search vs. depth-first search:

- the breadth-first search should be used for finding the shortest path
- if the state space of the graph is very large, the breadth-first search uses a significantly larger amount of memory
  - the size of the stack in depth-first search is usually kept smaller than the size of the queue is breadth-first search

  - in most applications for example in artificial intelligence the size of the queue makes using breadth-first search impossible

- if the size of the graph can be infinite, the problem arises that the depth-first search doesn't necessarily ever find a goal state and doesn't finish until it runs out of memory
  - this occurs if the algorithm starts investigating a futile, infinite branch
  - this is not a problem with finite graphs
- some more complicated problems like searching for loops in the graph can be solved with depth-first search
  - the grey nodes from a path from the source to the current vertex
  - only black or grey vertices can be accessed through a black vertex

    $\Rightarrow$ if a grey vertex can be reached from the current vertex, there is a loop in the graph