CS 225 – Data Structures

ZJUI – Spring 2022

Lecture 9: Graph Data Structures

Klaus-Dieter Schewe

ZJU–UIUC Institute, Zhejiang University

International Campus, Haining, UIUC Building, B404

email: kd.schewe@intl.zju.edu.cn

9 Graph Data Structures

After linear sequence structures, hash tables and tree structures we now look at graphs in general

Different to all previous topics there is no canonical choice how to represent graphs

We will look into different options: edge sequences, adjacency arrays, adjacency lists, adjacency matrices

We will then explore graph traversal procedures enabling to retrieve data from graph data structures

In the following sections we look into graph algorithms, in particular concerning minimum spanning trees and shortest paths

The algorithms highlight, which graph data structures is suited best for a particular problem and algorithm

Directed Graphs

Recall that a **directed graph** consists of a set V of **vertices** (also called **nodes**) and a set E of **edges**

Each edge $e \in E$ has an **origin** (or **start node**) $o(e) \in V$ and a **destination** (or **end node**) $d(e) \in V$

A directed graph $G = (V, E)$ is **simple** iff for any pair $(v_1, v_2) \in V \times V$ there is at most one edge $e \in E$ with origin $o(e) = v_1$ and destination $d(e) = 2$

In the case of simple directed graphs we can identify edges e with pairs $(o(e), d(e)) \in V \times V$

In the following all directed graphs that we consider will be simple, unless we state explicitly that G is a **directed multigraph**

Undirected Graphs

An **undirected graph** also consists of a set V of **vertices** (also called **nodes**) and a set E of **edges**

Each edge $e \in E$ is associated with a set $\{e_1, e_2\}$ of **end nodes**)—if this set contains only one element, the edge is called a **loop**

An undirected graph $G = (V, E)$ is **simple** iff for any pair $(v_1, v_2) \in V \times V$ there is at most one edge $e \in E$ with end nodes $\{v_1, v_2\}$

In the following all undirected graphs that we consider will also be simple, unless we state explicitly that G is an **undirected multigraph**

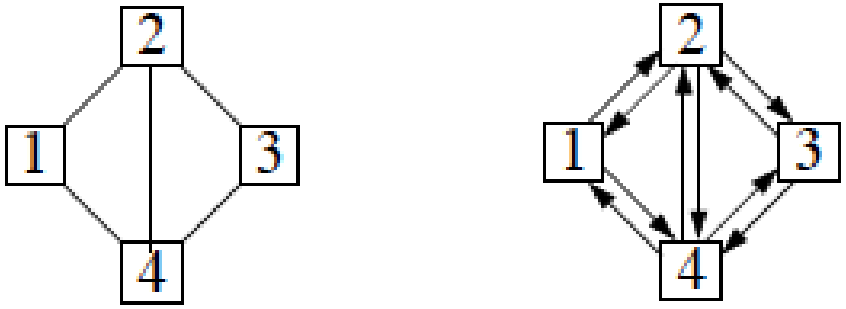
9.1 Graph Representations

Representing an edge (directed or not) by a pair (v_1, v_2) is an obvious choice

If there are no vertices that do not occur as end node of some edge, it suffices to store the edges

Then it is a matter of interpretation, if a pair (e_1, e_2) represengts a directed or an undirected edge

Undirected graphs are often identified with **bi-directional graphs**, where for each edge $(e_1, e_2) \in E$ there is also an edge $(e_2, e_1) \in E$



Unordered Edge Sequences

A straightforward representation of a graph $G = (V, E)$ is to use a sequence of edges in some arbitrary order (assuming that there are no "orphan" vertices)

Example: The undirected graph above could be represented by the list

$$[(1, 2), (2, 4), (3, 4), (2, 3), (1, 4)]$$

Then we can exploit any of the sequence data structures, in particular unbounded arrays or (singly or doubly) linked lists to store a graph

This can also be extended to weighted graphs, e.g. in the case of edge labels we could use triples instead of pairs

However, there are very few graph algorithms, for which a representation of a graph by an edge sequence would be efficient

Unordered Edge Sequences / cont.

In many algorithms we have to find **out-going** edges, i.e. given a node $v \in V$ find an edge $(v, w) \in E$, for which the whole sequence has to be searched

This becomes worse, if we have to follow paths, i.e. once $(v, w) \in E$ is found, search for $(w, u) \in E$, etc.

We already note the requirement that *a graph representation should permit easy access to edges that are incident to a given node*, i.e. the node is the origin or destination of the edge

Therefore, unordered edge sequences are mainly used for input and output of graphs

We will later look at the algorithms by Kruskal and Prim for the construction of minimum spanning trees; the former one works well also with unordered edge sequences

Adjacency Arrays

First concentrate on graphs that are *static* in the sense that there is no need to add or remove edges or nodes

In this case we can use for each node v an array representing (in some arbitrary order) all edges (v, w) outgoing from v

If edges are labelled with labels in an ordered set T, we may also order the array according to the labels

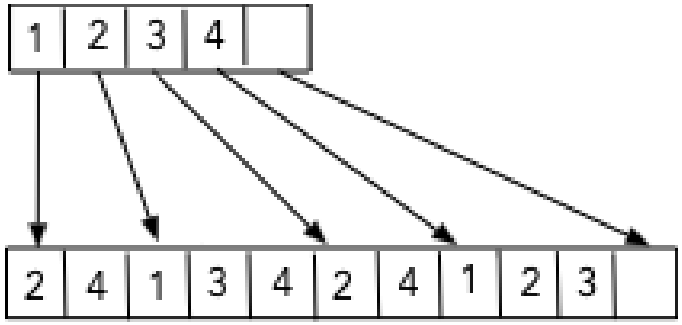
We may concatenate all these edge lists and use a single array E to store all edges

In addition, we can use an array V for storing the nodes, each with a pointer to its first outgoing edge in the edge array

If $|V| = n$ and $|E| = m$, then it is convenient to store a dummy node in the the $(n+1)$st field $V[n]$ of the array V pointing to the $(m+1)$st field $E[m]$ of the edge array E

Adjacency Arrays / cont.

If we simply number the nodes, i.e. $V = \{0, \ldots, n-1\}$ then the edges outgoing from v are found in $E[V[v]], \ldots, E[V[v+1]-1]$



The required space is $n + m + 2$, even less than for edge sequences

The time required to find an outgoing edge is linear in the outdegree of the given node

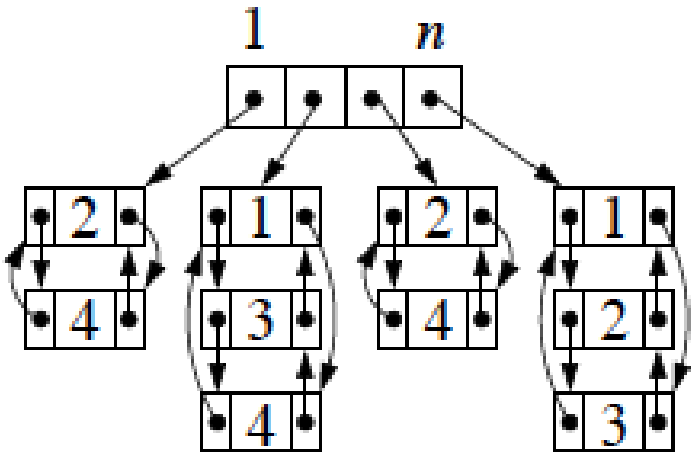
However, if also access to incoming edges is required, additional arrays E' and V' are required

Adjacency Lists / 1

Edge arrays are compact and efficient, but only suitable for static graphs

If we require the stored graph to be updated as well, we may use lists of arbitrary and varying length to store the outgoing edges of any node

Then we can use unbounded arrays or singly or doubly linked lists to represent these lists plus an unbounded array



We can dispense with dummy nodes in linked lists, as we have the extra array for storing the nodes

Adjacency Lists / 2

To find an outgoing edge we proceed in the same way as for adjacency arrays

If also incoming edges aee required, we have to duplicate the data structure as for adjacency arrays

When a new edge is to be inserted or deleted, we can use the corresponding list operation for the list of outgoing edges

In all nodes in the edge lists we may store additional data, e.g. edge labels

If also the vertex set is subject to updates, we may also a linked list or an unbounded array for the representation of V

Implementation

Define a class GRAPH capturing adjacency lists

Define a class VERTEXLIST analogous to the class DLIST

However, for vertices we provide an additional attribute *edges* linking to the list of outgoing edges

For the edge lists we proceed analogously using doubly linked lists without dummy node:

Define functions for insertion and deletion of a vertex, provided there are no edges incident to it

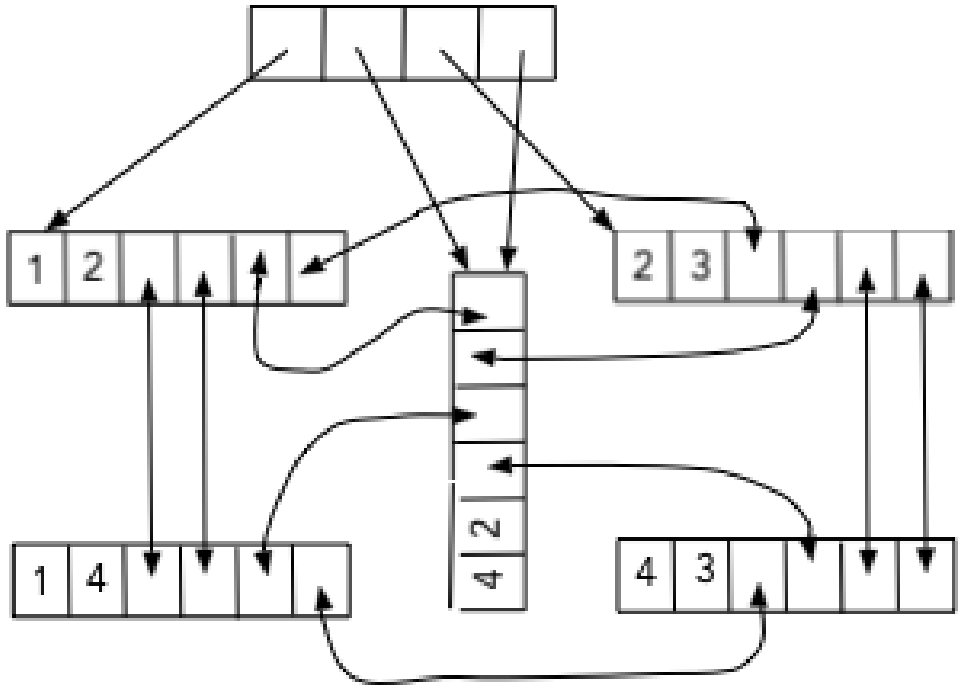
Define functions for insertion and deletion of edges

Define retrieval functions returning all outgoing edges of as vertex

Adjacency Lists: Optimisation

If access to both outgoing and incoming edges of a node is required, it is natural to store this information only once

One may use just one item for an undirected edge and make this item a member of two adjacency lists



Adjacency Matrices

For completeness we also mention adjacency matrices

If $G = (V, E)$ is a graph with $|V| = n$, the the **adjacency matrix** of G is an $n \times n$ matrix $A = (a_{i,j})_{1 \leq i,j \leq n}$ with $a_{i,j} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{else} \end{cases}$

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

In most graphs the number of edges is far less than n^2, so an adjacency matrix will often be sparse

Finding outgoing edges requires to look at all n entries in a row of A, which requires time in $O(n)$ and is not efficient

However, adjacency matrices make sense as a tool to link graph theory with linear algebra

9.2 Graph Traversal

Algorithms on graphs often require to systematically explore the graph inspecting each vertex exactly once

This is realised by a procedure searching the graph from a start vertex until all reachable vertices have been explored

This applies to undirected and directed graphs—though we usually refer to the terminology of directed graphs such as *outgoing edge* instead of *incident edge*

We look into the two most common strategies for graph traversal: **breadth-first search** (**BFS**) and **depth-first search** (**DFS**)

We will also sketch some ideas for **heuristic search**

Examples

When we look at the plan of public transportation in a city we may be interested in finding out, which places can be reached from the current location

When roads in a city centre are organised primarily as one-way roads it will be important to find out, which places can be reached from a given one

Similar connectivity problems occur in all kinds of networks that are representable by graphs

When given a bipartite graph the question arises, if there is a complete or even perfect matching, and if yes how to determine it efficiently, which requires to traverse the graph

Graph traversal is also required, if we plan a complete tour visiting all vertices exactly once

General Terminology

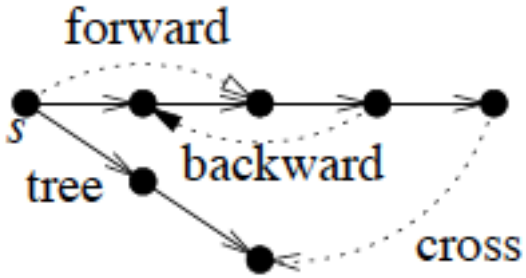
All graph traversal algorithms construct forests, i.e. sets of trees, and partition edges into four classes:

Tree edges. These are the edges (v, w) in the constructed forest

Forward edges. These edges (v, w) connect vertices that are already connected by a path in the constructed forest from v to w

Backward edges. These edges (v, w) connect vertices that are already connected by a path in the constructed forest from w to v

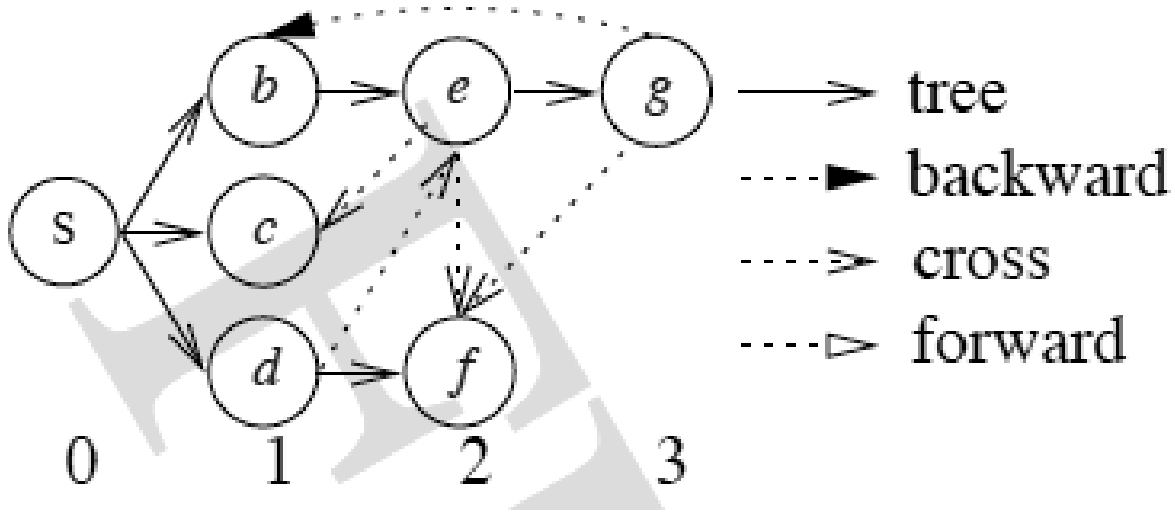
Cross edges. These edges connect vertices in different branches of the constructed forest



Breadth-First Search

The idea of BFS is to search the graph *layer-by-layer*:

- The start node is on layer 0
- If there is an edge (v, w) and v is on layer ℓ, then w is on layer $\ell + 1$ unless it has a neighbour on a layer $< \ell$



BFS Algorithm

Initialise two arrays *distance* and *parent*, where the entries correspond to the vertices: initially all distances and parents are undefined

Set the parent of the start node s to itself and its distance to 0

Use two queues Q and Q' for the nodes on the current and next layer: initially Q contains the start node s, and Q' is empty

Always choose the first vertex in Q and explore all its outgoing edges (v, w)

For a vertex w with undefined parent set the parent to the current node v, set the distance to the current distance +1, and push w to the queue Q'

When all nodes on a layer have been explored, switch to the next layer, i.e. Q becomes Q' and Q' is again empty

Properties

As we inspect in each step all outgoing edges, the algorithm explores all vertices that are reachable from the start node

That is, if the graph is connected, the whole graph will be explored

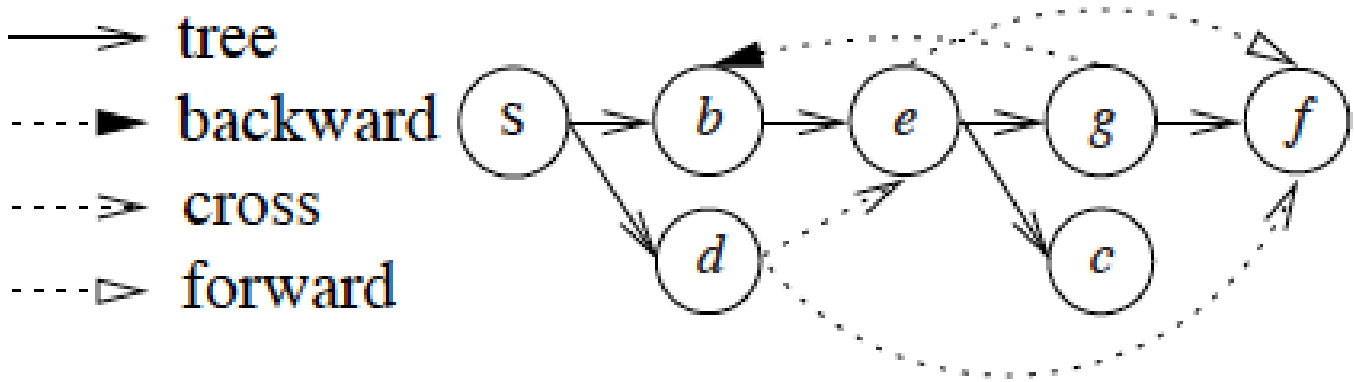
All nodes except the start node will have a unique parent, so the edges between nodes and their parents define a tree (with the start node as root)

As all edges in the connected component defined by the start node are inspected exactly once, the time complexity of the algorithm is in $O(|E|)$

Depth-First Search

While breadth-first search (BFS) proceeds carefully looking at all possible continuations at the same time, depth-first search does the extreme opposite

Whenever a new node is found, the search immediately continues from this node



An easy implementation would be to adapt the BFS algorithm using a stack instead of a queue—we will explore a similar approach, but make the handling of tree edges and others explicit

Depth-First Search Algorithm / 1

When given a start node s the DFS algorithm constructs a tree by exploring the tree along paths from s—for this we call $dfs(s, s)$

Initially, only the start node s is marked

In general, for an edge (u, v) all continuation edges (v, w), i.e. outgoing edges of v are explored:

- If w is unmarked, the edge becomes a **tree edge**, and the search continues with $dfs(v, w)$
- Otherwise the edge is considered a **non-tree edge** (i.e. a forward, backward or cross edge) that is handled by a *traverseNontreeEdge* procedure

Depth-First Search Algorithm / 2

In addition, the algorithm determines the order, in which nodes are reached, and the order, in which they are finished

For these two arrays *dfsNum* and *finishTime* are used, respectively

Two counters *dfsPos* and *finishingTime*, both initially 1, are used:

- Whenever a node is reached and marked, the correponding entry in the array *dfsNum* is set to *dfsPos*, and the counter is incremented

- Whenever a node has no more outgoing edges to be explored, the corresponding entry in the array *finishTime* is set to *finishingTime*, and the counter is incremented

  In this case a *backtrack* procedure is called

DFS Implementation

We can add some attributes to GRAPH objects in order to avoid having to pass a lot of parameters

The implementation follows the sketch above and creates a tree for a given starting node

The functions *__traverseTreeEdge*, *__traverseNontreeEdge* and *backtrack* can be tailored to specific needs for an algorithm traversing a tree

The initialisation can use arrays (or linked lists) for the vertices marked, the sequence numbers for the nodes reached by the search and the sequence numbers for the search being completed

The core of the DFS implementation is the *dfs* function, which classifies the edges

9.3 Graph Exploration

We will look briefly into several topics, where graph exploration is applied

- Decide, whether an undirected graph is acyclic or not
- Discover strongly connected components of an undirected graph
- Explore an implicitly defined graph using backtracking algorithms
- Alternatively, use the ***branch-and-bound*** technique
- Explore implicit graphs for games

Finally, look into alternatives to BFS and DFS: **heuristic search**

Directed Acyclic Graphs

Now concentrate on directed graphs, for which DFS can be applied just as described in the previous subsection—extend the search with a non-explored start node in case not all vertices have been reached

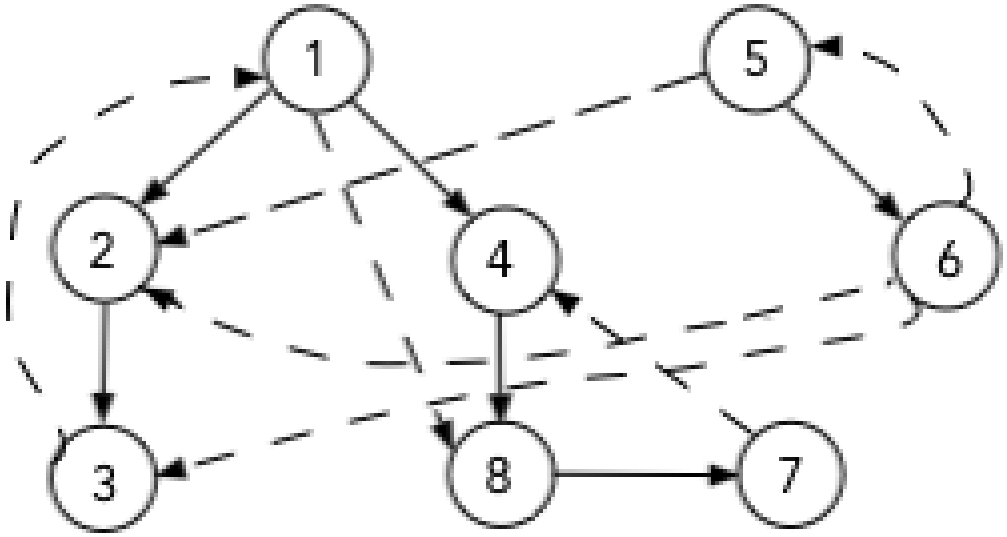
We can exploit DFS to determine, whether a directed graph is acyclic or not

Let $G = (V, E)$ be a directed graph and let $F = (V, E')$ be the forest generated by DFS with some arbitrary start node $s \in V$. Then G is acyclic iff $E - E'$ does not contain any backward edge

Clearly, if $E - E'$ contains a backward edge (v, w), then this edge plus the path from v to w in F defines a cycle

Conversely, if there is a cycle, we can first replace forward edges by the corresponding path, then see that there cannot be any cross edges due to the organisation of the search, so there must exist a backward edge in the cycle

Example



The graph contains the cycle $4 - 8 - 7 - 4$ with the backward edge $(7, 4)$

The cycle $1 - 2 - 3 - 1$ contains the backward edge $(3, 1)$

Topological Sorting

A **topological order** is a partial order $\leq$ on the set V of vertices of a directed acyclic graph $G = (V, E)$ such that $(v, w) \in E$ implies $v \leq w$

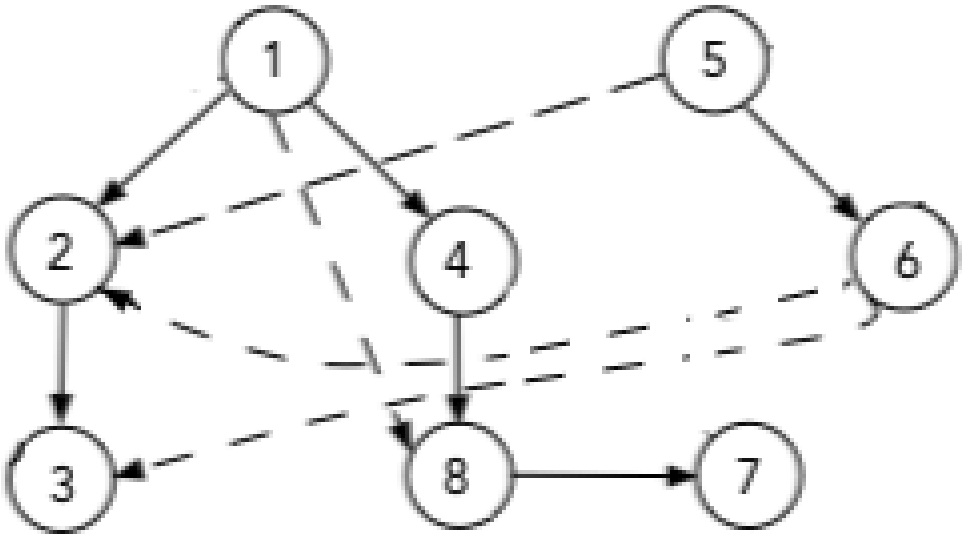
Topological sorting returns a list of the vertices in V in some topological order $\leq$

We can modify DFS to become a topological sorting algorithm: we start with a node without incoming edges

Whenever we need to *backtrack* from a vertex v, push v onto a stack

Finally, the elements on the stack (from top to bottom) provide a sequence of the vertices that is topologically sorted

Example



In this example (starting from 1 and continuing from 5) the resulting topological order will be $[5, 6, 1, 4, 8, 7, 2, 3]$

If we start with 5 (and continue with 1), the resulting topological order will be $[1, 4, 8, 7, 5, 6, 2, 3]$

Correctness

The algorithm defines a partial order: we have $v < w$, if v appears before w in the sequence produced by the algorithm, equivalently if w is pushed onto the stack before v

The partial order $\leq$ is the reflexive, transitive closure of $<$

The partial order $\leq$ defined by the topological sorting algorithm is a topological order.

Proof. A vertex v is pushed onto the stack, when the algorithm backtracks from it

As the graph is acyclic, backtracking from v occurs, when all outgoing edges from v have been explored before

In particular, if $(v, w) \in E$ is such an outgoing edge, then at the time v is pushed onto the stack w is already on the stack

According to the definition of $\leq$ this implies $v \leq w$, i.e. $\leq$ is a topological order

Strongly Connected Components

A directed graph $G = (V, E)$ is ***strongly connected*** iff for all $u, v \in V$ there exists a path from u to v

A ***strongly connected component*** of a directed graph $G = (V, E)$ is maximum subgraph that is strongly connected

In our example (slide 439) we have three strongly connected components $(\{1, 2, 3\}, E_1)$, $(\{5, 6\}, E_2)$ and $(\{4, 7, 8\}, E_3)$

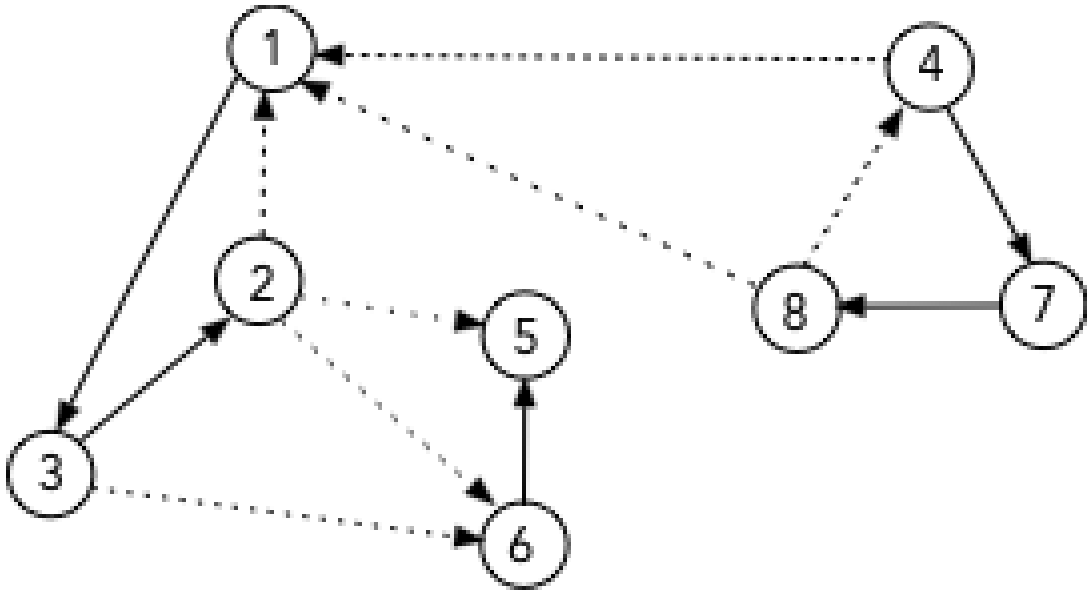
Here the edge sets E_i contain all edges that have origin and destination vertices in the corresponding set V_i, i.e. $V_1 = \{1, 2, 3\}$, $V_2 = \{5, 6\}$, and $V_3 = \{4, 7, 8\}$

DFS for Strongly Connected Components

Again, we can adapt DFS to determine all strongly connected components

- First run DFS on the graph G computing a forest and the array *finishTime*, which provides the order, in which the nodes have been finished by DFS
- Then construct the **inverted graph** $G' = (V, E')$ with $E' = \{(w, v) \mid (v, w) \in E\}$
- Run DFS on G' starting with the node i with highest value of $\mathit{finishTime}(i)$—also choose a node with highest value of $\mathit{finishTime}(j)$, when another start of DFS is required
- Each tree in the resulting forest defines a strongly connected component of G

Example



In the inverted graph G' for our previous example we start with 6, then choose 1 and finally 4

The nodes in each tree constructed by DFS on G' are the nodes generating the connected components of G

Correctness and Complexity

For the correctness of the above algorithm we have to show the following:

Two vertices $v, w \in V$ of a directed graph $G = (V, E)$ are in the same strongly connected component iff the above algorithm constructs a forest, where v and w belong to the same tree in the inverted graph $G' = (V, E')$.

Proof. First assume that v, w are in the same stongly connected component of G

At some point the DFS algorithm on G will reach a vertex u in this strongly connected component, thus producing a subtree with root u that contains both v and w

Among all vertices in this strongly connected component $\mathit{finishTime}(u)$ will be maximal, so the DFS algorithm on G' will at some point start with u

As there are paths from v and w to u in G, there are also paths from u to v and w in G'

Hence the DFS algorithm on G' will produce a tree with root u that contains both v and w

Proof / cont.

Conversely, assume that v, w are in the same tree in the inverted graph G'

Let u be the root of this tree, i.e. in the DFS algorithm on G' this vertex u is considered first (among those vertices in this tree)

Assume $v \neq u$, so we have $\mathit{finishTime}(u) > \mathit{finishTime}(v)$, and there is a path from u to v in G'

Hence there is also a path from v to u in G

When DFS was carried out on G we finished v before u, so there are just two possibilities:

- There is a path from u to v
- u is neither ancestor nor descendant of v

Proof / cont.

In the second case u and v must have a common least ancestor u', and the path from u' to v is explored before the path from u' to u

As there is also a path from v to u, the vertices on this path have either been finished before v or they are ancestors of v

This is only possible, if the path from v to u passes through u', which implies $\mathit{finishTime}(u') > \mathit{finishTime}(u)$ contradicting our assumption on u

This leaves only the first possibility, i.e. there exists a path from u to v

The same arguments apply to u and w, and this still holds, if u is one of v or w

Hence, v and w are in the same strongly connected component, which completes our proof

Complexity. The algorithm above basically runs DFS twice, each time requiring time in $O(|E| \cdot |V|)$, because we have to check, whether a vertex is marked

So the time complexity for the determination of strongly connected components using DFS is in $O(|E| \cdot |V|)$

Implicit Graphs: Backtracking

Sometimes graphs are only implicitly defined, e.g. using set comprehensions for the vertex or edge sets

For instance, consider a two-player game like Chess or Go, where nodes are legal configurations of the game, and edges correspond to possible moves by a player

To execute a playing strategy requires to explore the graph at least to some depth from a given start node

This can be done by DFS with **backtracking**, by means of which we look into a different possible continuation of the game

If the problem is to find an exit out of a maze, a backtracking algorithm corresponds to a strategy of using an **Ariadne thread**

Example

Consider the problem how to place n queens on an $n \times n$ chessboard, such that these queens do not threaten each other—in real Chess we have $n = 8$

It is not difficult to see that we can formalise this as follows: Find a permutation $\sigma \in S_n$ such that the two sets $\{\sigma(i)-i \mid 1 \leq i \leq n\}$ and $\{\sigma(i)+i \mid 1 \leq i \leq n\}$ each have n elements—the permutation property covers threats in rows and columns, and the two sets cover threats in diagonals

A node in the directed graph corresponds to some choice $\sigma(i) = j$, and successor nodes correspond to $\sigma(i+1) = k$ for $i < n$

Add an empty root with n successors $\sigma(1) = j$ for $1 \leq j \leq n$

In each step choose the successor $\sigma(i+1) = j$ with smallest unexplored j such that the constraints are all satisfied

Example / cont.

So let $n = 8$, so we first choose $\sigma(1) = 1$

Then we successively have to choose $\sigma(2) = 3$, $\sigma(3) = 5$, $\sigma(4) = 7$, $\sigma(5) = 2$, $\sigma(6) = 4$ and $\sigma(7) = 6$

Unfortunately, this leaves only the option $\sigma(8) = 8$, which violates the constraints

We have to backtrack to get the next possible choice with $\sigma(5) = 4$, which requires $\sigma(4) = 2$, and then forces us to backtrack again, this time to $\sigma(4) = 8$

We continue until we reach a permutation that satisfies all constraints—the number of choices explored is fairly small

Branch and Bound

Branch and Bound defines another method for the exploration of implicit directed graphs

The method is used for problems, where an optimal solution is sought, e.g. a complete cycle with minimal costs for the travelling salesman problem (TSP)

In the simplest version it calculates a lower bound for a solution that can be reached from a given node

This lower bound is used to discard certain continuations for a breadth-first or depth-first search

It is also used as a heuristics, which continuation to explore first

We will look at branch and bound on an example for TSP

Example: TSP

In case of the **travelling salesman problem** (**TSP**) we are given a directed graph $G = (V, E)$, where the edges e are labelled with costs $c(e) > 0$

Without loss of generality we can assume that the graph is complete, i.e. for any $v, w \in V$, $v \neq w$ we have an edge $e = (v, w) \in E$—if not, add it with $c(v, w) = \infty$

We can represent the graph by an $n \times n$ matrix, where $n = |V|$ and $c(v, w)$ is the entry in the row corresponding to v and the column corresponding to w

The problem is to find a cycle, in which each vertex is visited exactly once, with minimum costs

Example / Search Graph

Assume $n = 5$ with the following cost matrix:

$$\begin{pmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{pmatrix}$$

Create an (implicit) search graph, where the nodes are sequences $v_1, \ldots, v_k$ of distinct vertices ($k \leq n$) that correspond to a prefix of a tour (with a fixed start v_1)

We have an edge from $v_1, \ldots, v_k$ to $v_1, \ldots, v_k, v_{k+1}$, i.e. we extend the tour by exactly one vertex

Example / Calculation of Bounds

The cost $c(v, w)$ can be divided up into the cost $c(v, w)/2$ to *leave* v *in direction* w and the cost $c(v, w)/2$ to *reach* w *from direction* v

Then for each vertex there is a ***minimum cost for leaving*** as well as a ***minimum cost for arriving***

$$min_{\text{leave}}(v) = \frac{1}{2}\min\{c(v, w) \mid w \in W\} \qquad min_{\text{arrive}}(w) = \frac{1}{2}\min\{c(v, w) \mid v \in V\}$$

The ***cost for visiting*** a node v is the sum $min_{\text{leave}}(v) + min_{\text{arrive}}(v)$

For each vertex $v_1, \ldots, v_k$ of the search graph we can then calculate a lower bound for a tour with this prefix: add the costs for the path, the costs for visiting each of the remaining nodes (excluding to be reached from $v_1, \ldots v_{k-1}$ or leaving to $v_2, \ldots, v_k$) and the cost for returning to v_1 from one of the remaining nodes

Example / Branch and Bound Search

Starting from 1 we compute the lower bound of any tour adding 2 (costs for leaving 1), 6, 4, 3, 3 (costs for visiting the other nodes), and 2 (costs for arriving at 1)—this gives the bound 20

For the four successors $1-2$, $1-3$, $1-4$ and $1-5$ we compute the lower bounds 31, 24, 29, 41—e.g. for $1-2$ add the cost 14 for the path from 1 to 2, 7/2 for the departure from 2 to one of the remaining nodes, 11/2, 3 and 3 for the costs of visiting nodes 3, 4 and 5 (not coming from 1 nor leaving to 2), and a cost 2 for the return to node 1 from 3, 4 or 5

Therefore, continue with $1-3$ and explore the successors $1-3-2$, $1-3-4$ and $1-3-5$

For these we calculate the lower bounds 24, $30+1/2$ and $40+1/2$

Example / cont.

For instance, for $1-3-2$ we calculate 9 for the path from 1 to 2, 7/2 for the departure from 2 towards 4 or 5, costs 3 each for the visits of 4 and 5 and 11/2 for the return to 1 from 4 or 5

Finally, the successors $1-3-2-4$ and $1-3-2-5$ permit only one possibility to complete the tour, so we get the real costs for $1-3-2-4-5-1$ as 37, and the real cost for $1-3-2-5-4-1$ as 31

Then a continuation from $1-3-4$ or $1-3-5$ can be ignored, as the costs will always be higher than 31; the same holds for $1-2$ and $1-5$

So we continue from $1-4$, and compute the lower bounds 40, $41+1/2$ and 29 for the successors $1-4-2$, $1-4-3$ and $1-4-5$, so continue with $1-4-5$

Finally, we determine the cycle $1-4-5-2-3-1$ with the lowest cost 30

Heuristic Search: A* Algorithm

The A* algorithm is the best known **heuristic search** algorithm

In general, heuristic search expands in each step a node by looking at the successor and providing an estimate on the success by proceeding with this successor

A* keeps a list of all the expanded and not yet further explored nodes, and chooses the most promising successor

The more the search progresses the more the real costs instead of the estimates are used

In this way paths that have been explored for a while may be given up at some point, when the continuation does not look promising anymore