Applied Algorithm Design
Lecture 2

Pietro Michiardi

Eurecom

Part I: Basics of Algorithm Design

Introduction

Analyzing algorithms involves thinking about how their resource requirements, the *amount of time* and the *space* they use, will scale with increasing input size.

1. We will put this notion into practice
2. We will develop mathematical tools to talk about how different functions scale with increasing input size
3. We will develop *running-time bounds* for some basic algorithms, including the G-S algorithm

Computational tractability

Our goal in this course is to identify *efficient algorithms* for some problems. But how can we define efficiency?

Note:

A common property shared by most of the problems we will address is their fundmentally *discrete* nature. Like the Stable Matching problem, they will involve an implicit search over a large set of combinatorial possibilities

We will focus primarily on efficiency in running time: we want algorithms that run quickly. However, it is important that algorithms be efficient in their use of the amount of space (*memory*) they use. We will see techniques for reducing the amount of space needed to perform a computation.

Efficiency: informal definition

Efficiency:

An algorithm is efficient if, when implemented, it runs quickly on real input instances.

But what does “when implemented” mean? Well, it depends ...

- On *where* the algorithm is implemented: even bad algorithms can perform well on very fast processors;
- On *how well* they are implemented: even good algorithms can run slowly when they are sloppily implemented;

And what about the algorithm *scalability*? A common situation is that two very different algorithms will perform comparably on inputs of size 100; multiply the input size by 10^6 and one will still run quickly while the other consumes a large amount of time!!

Efficiency: example

What we need is a definition of efficiency that is *platform independent*, *instance independent* and predictive of what will happen when the input increases.

Example:

Recall the Stable Matching Problem and the G-S algorithm. The input to the problem has a natural size parameter N. We could take this to be for example the total size of the representation of all preference lists.

Otherwise, we could consider the number of men and women we try to match. Since there are n men and n women there will be $2n$ preference lists each of size n, hence we could view $N = n^2$.

Worst-case running times and brute force search

Worst-case analysis:

We will look for a bound on the largest possible running time an algorithm could have over all inputs of a given size *N*, and see how this scales with *N*.

But, isn't this a little bit too demanding? What if we have an algorithm that performs well "most of the time" and has just few pathological cases?

We will see, that worst-case analysis works well in most of the cases, and that is more robust than other kind of analysis. Think about an "average case" analysis. Then your definition would depend on how you "randomize" over the inputs of the algorithm and the attention would deviate from the algorithm analysis to the way random numbers are generated!!

Brute-force search

What is a reasonable analytical benchmark that can tell us whether a running-time bound is impressive or weak? A first possible comparison is with *brute force* search over the search space of possible solutions

Example:

Let’s go back to the SMP: even when the input size is small, the search space is huge in this problem! There are $n!$ possible perfect matchings between n men and n women. The natural brute-force approach would list all these possible matchings and then pick out the stable one.

Without implementing the algorithm we were able to characterize its running time at an *analytical* level and extract hints towards its implementation showing we could do better than brute-force search.

Polynomial Time as definition of efficiency

Although comparing against brute-force search can be useful sometimes, people in the past spent much effort in defining what is a *reasonable* running time.

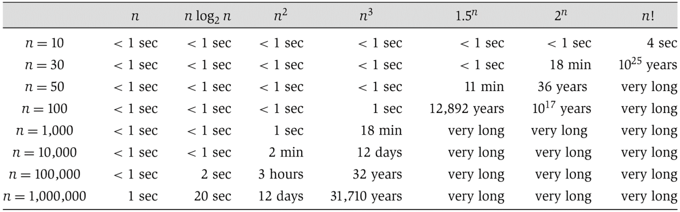
Polynomial time algorithms:

Suppose there are constants $c, d > 0$ so that on every input instance of size N to the algorithm, its running time is bounded by cN^d primitive computational steps. Then the algorithm is a polynomial time algorithm.

Efficiency: formal definition

An algorithm is efficient if it has a polynomial running time

Examples of running times:



	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	4 sec
$n = 30$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	18 min	10^{25} years
$n = 50$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	11 min	36 years	very long
$n = 100$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	$<$ 1 sec	$<$ 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	$<$ 1 sec	$<$ 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	$<$ 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Figure: Example of running times.

To sum up...

The first definition of efficiency was tied to the specific implementation of an algorithm: it turned efficiency into a moving target that depends on processor speeds.

The definition in terms of polynomial time is much more an absolute notion and, very importantly, it becomes negatable. It becomes possible to express the notion that *there is no efficient algorithm for a particular problem*.

Asymptotic order of growth

Our discussion of computational tractability has turned out to be based on our ability to express the notion that an algorithm's worst case running time on input of size n grows at a rate that is at most proportional to some function $f(n)$.

The function $f(n)$ becomes a bound on the running time of the algorithm.

Asymptotic order of growth

When we seek to say something about the running time of an algorithm on its input of size *n*, we could aim at a very precise statement such as:

Example:

On any input of size n, the algorithm runs for at most $1.6n^2 + 3.5n + 8$ steps.

It may be interesting in some context to be so precise but:

- Being so precise is generally an exhausting activity
- In many cases, such statements about running time are meaningless
- What is a *step*?

O, Ω and Θ

We want to express the growth rate of running times and other functions in a way that is insensitive to constant factors and low-order terms. We'd like to summarize the example before and say the algorithms running time grows like n^2.

Asymptotic upper bounds

$T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $\forall n > n_0$ we have $T(n) \leq cf(n)$

It is important to note that this definition requires a constant c to exist that works for all n; in particular c must not depend on n. Furthermore, $O(\cdot)$ expresses only an upper bound, not the exact growth rate of a function.

O, Ω and Θ

Asymptotic lower bounds

$T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $\forall n \geq n_0$ we have $T(n) \geq cf(n)$

This definition is just like $O(\cdot)$, except that we are bounding the function $T(n)$ from below, rather than from above.

Note:

Let $T(n) = pn^2 + qn + r$.

- $T(n)$ is $O(n^2)$ but it is also true that $T(n)$ is $O(n^3)$
- $T(n)$ is $\Omega(n^2)$ but it is also true that $T(n)$ is $\Omega(n)$

O, Ω and Θ

We just saw that upper and lower bounds can be *tighter* or *weaker*.

Asymptotically tight bounds

$T(n)$ is $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$

In a natural sense, if we can prove that the above definition holds, then we have found the "right" bound. One can find a tight bound by closing the gap between upper and lower bounds.

Note:

Let f and g be two functions such that $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists and is equal to some number $c > 0$. Then $f(n)$ is $\Theta(g(n))$.

Properties of asymptotic growth rates

Transitivity

- If $f = O(g)$ and $g = O(h)$ then $f = O(h)$
- If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$
- If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$

Additivity

- If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$
- If $f = \Omega(h)$ and $g = \Omega(h)$ then $f + g = \Omega(h)$
- If $f = \Theta(h)$ and $g = \Theta(h)$ then $f + g = \Theta(h)$

Asymptotic growth of some common Functions

- **Polynomials**: Let $f(n) = a_0 + a_1 n + \cdots + a_d n^d$. Then $f(n) = O(n^d)$ if $a_d > 0$;

Polynomial running time:

An algorithm is polynomial if its running time $T(n)$ is $O(n^d)$ for some constant d.

- **Logarithms**: Let $f(n) = \log n$. Then $f(n) = O(n^x)$.
  This means that logarithms grow slower than polynomials.

- **Exponentials**: For every $r > 0$ and every $d > 0$ we have n^d is $O(r^n)$.
  This means that exponentials grow faster than polynomials.

All right... but how can we put all this into practice?

The G-S algorithm implementation

How to go for implementing it:

- Write a pseudo-code
- Decide which data structure to use
- Check it's running time with some toy examples

The implementation of basic algorithms using data structures is something that you probably have had some experience with before... As we will see, the choice of data structures to use is left to the designer of the algorithm, which has to chose the ones that make it efficient and easy to implement. In some cases, this may involve **preprocessing** the input.

Implementation of G-S algorithm

Data structures

- Arrays: let $A[i]$ be an array (or vector) of size n
  - What is the i^{th} element on the array? $\rightarrow O(1)$
  - Is $e \in A[i]$? $\rightarrow O(n)$
  - If we pre-process the array and *sort* it, then the question above can be answered in $O(\log n)$.
- Lists: simple linked lists or doubly linked list
  - They are better than arrays to maintain a dynamic set of elements
  - But, unlike arrays, we cannot find the i^{th} element of the list in $O(1)$, it takes $O(i)$

It is important to recall that we can manipulate the input of a problem before feeding it to the algorithm. In general it can help improving the efficiency of the algorithm... But it does not come for free: you have to count also the time you take to preprocess the input.

A survey of common running times

Linear Time:

An algorithm runs in $O(n)$, i.e. is linear, if its running time is at most a constant factor times the size of the input.

One basic way to get an algorithm with a linear running time is to process the input in a single pass, spending a constant amount of time on each item of input encountered.

- Example 1: Computing the Maximum
- Example 2: Merging two sorted lists

A note on Linear Time

Sometimes the constraints of an application force this kind of one-pass algorithm we have just seen.

Example:

An algorithm running on a high-speed switch on the Internet may see a stream of packets flying past it, and it can try computing anything it wants to as this stream passes by, but it can only perform a constant amount of computational work on each packet, and it can't save the stream so as to make subsequent scans through it.

Sub-Area of algorithm design:

The study of this model of computation (one-pass over inputs) is the subject of two trendy branches of algorithm design:

Online Algorithms and **Data Stream Algorithms**

A survey of common running times

$O(n \log(n))$ Time

This is very common in many algorithms: it is the running time of any algorithm that splits its input into two equal-sized pieces, solves each piece recursively, and then combines the two solutions in linear time.

It arises in **Divide and Conquer** algorithms

Example:

Sorting is perhaps the most well-known example of a problem that can be solved this way. For example the *Mergesort* algorithm divides the set of input numbers into two equal-sized pieces, sorts each half recursively, and then merges the two sorted halves into a single sorted output list.

A note on $O(n \log(n))$ running time

Note:

We also frequently encounter $O(n \log(n))$ as a running time simply because there are many algorithms whose most expensive step is to sort the input.

Example:

Given n time-stamps $x_1, \cdots, x_n$ on which copies of a file arrive at a server, what is the largest time interval when no copies of the file arrive?

$O(n \log(n))$ solution: sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.

A survey of common running times

Quadratic Time

This running time arises when you enumerate all pairs of elements.

Example: Closest pair of points in a plane

Given a list of n points in a plane $(x_1, y_1), \cdots, (x_n, y_n)$ find the pair that is closest.

- $O(n^2)$ solution: try all pairs of points (two `for` loops).
- It seems that $\Omega(n^2)$ seems inevitable
- People have shown instead that with the principle of Divide and Conquer you can design an algorithm that runs in $O(n \log(n))$; others have shown that with **Randomization** it is possible to design an algorithm that runs in $O(n)$ time.

A survey of common running times

$O(n^k)$ Time

In the same way we obtained a running time of $O(n^2)$ by performing brute-force search over all pairs formed by a set of n items, we obtain a running time of $O(n^k)$ for any constant k when we search over all subsets of size k.

Example: independent set of size k:

Given a graph $G = (V, E)$, are there k nodes such as no two nodes are joined by an edge?

- $O(n^k)$ solution: enumerate all subsets S of k elements and find those satisfying the condition
- Check whether S is an independent set = $O(k^2)$
- Number of subsets S with k elements = $\binom{n}{k} \leq \frac{n^k}{k!}$
- $O(k^2 n^k / k!) = O(n^k)$

A survey of common running times

Beyond Polynomial Time

The previous example leads us to the path of running times that grow faster than any polynomial. Two very frequent bounds are 2^n and $n!$.

Example: maximum independent set

Given a graph $G = (V, E)$, what is the maximum size of an independent set?

- $O(n^2 2^n)$ solution: enumerate all the independent sets and find the one with maximum size.

A note on beyond polynomial time

Note:

What we have seen in the previous example is nothing but a brute-force algorithm over the search space of the problem!

- 2^n arises in many problems: recall from Lecture 1, the interval scheduling problem. Trying out all possible subset of intervals would take $O(2^n)$ steps. However we will see in Lecture 3, that is actually feasible to design an algorithm that finds an optimal solution in $O(n \log(n))$ time.
- There are many problems that have a similar-looking search space, but in some cases it is possible to bypass the brute-force search, in others it is not.

Part II: Graph Theory

Introduction

The main focus of this course is on problems with a *discrete* flavor: discrete mathematics has developed basic combinatorial structures that lie at the heart of the subject of this second part of the Lesson.

One of the most fundamental and expressive combinatorial structure is the *graph*.

- We begin by giving some basic definitions that most of you will be familiar with
- List a spectrum of different algorithmic settings in which graphs arise
- Discuss on some basic algorithmic primitives for graphs

Basic definitions and applications

Definition: a Graph

A graph G is a way of encoding pairwise relationships among a set of objects: it consists of a collection V of *nodes* or *vertexes* and a collection E of *edges*, each of which joins two nodes. We thus represent an edge $e \in E$ as a two-element subset of V : $e = \{u, v\}$ for some $u, v \in V$, where we call u and v the *ends* of e.

Edges usually encode a symmetric relationship; in this case we call the graph an *undirected* graph.

Basic definitions and applications

Often we want to encode asymmetric relations, so we need to use the following definition.

Definition: Directed Graph

A directed graph G' constits of a set of nodes V and a set of *directed edges* E' such that each $e' \in E'$ is an *ordered pair* (u, v). We call u the *tail* and v the *head* of the edge. We will also say that e' *leaves node* u and *enters node* v.

Examples of graphs:

- Transportation networks: e.g. airline networks, indicating (symmetric) relations among airports and flights between them. It is interesting to notice a few things here: there are often a small number of “hubs” with a very large number of incident edges; furthermore it is often possible to get between any two nodes via a small number of intermediate hops.
- Communication networks: this is the most obvious example. Graphs can be used to model an enterprise network (nodes are real machines) as well as the whole Internet (nodes are ASs). Note that also wireless networks are just another kind of graphs where the nodes are situated at locations in physical space, and there is an edge from u to v if v is close enough to receive a signal from it. These graphs are usually called “radio graphs” or “line of sight” graphs.

Examples of graphs, continued...

- Information networks: the WWW can be viewed as a directed graph, in which nodes correspond to Web page and there is an edge from u to v if u has an hyperlink to v. Note that the directedness of the graph is crucial. The structure of the graph can be used by algorithms to try inferring the most important pages on the Web, a technique used by most current search engines (see Google!)
- Social networks: given a collection of people who interact we consider that nodes are people with an edge joining u and v if they are friends. Different types of edges (directed or not) may represent different kind of relations. These are really popular (see FaceBook!!) and can be used to understand the dynamics of rumor spreading (viral marketing) or the spreading of diseases (real or computer viruses).

Paths

A basic operation in a graph is that of traversing a sequence of nodes connected by edges.

Definition: Path

A path in an undirected graph $G = (V, E)$ is a sequence of nodes $v_1, v_2, \cdots, v_{k-1}, v_k$ with the property that each consecutive pair v_i, v_{i+1} is joined by an edge in G.

- *Simple path*: if all its vertices are distinct from one another
- *Cycle*: it is a path $v_1, v_2, \cdots, v_{k-1}, v_k$ in which $k > 2$, the first $k - 1$ nodes are all distinct and $v_1 = v_k$.

All these definitions carry over to directed graphs.

Connectivity

- Undirected graphs:

Definition: Connectivity

We say that an undirected graph is connected if, for every pair of nodes u, v, there is a path from u to v.

- Directed graphs:

Definition: Strong Connectivity

We say that an *directed* graph is strongly connected if, for every pair of nodes u, v, there is a path from u to v and a path from v to u.

Distance

In addition to simply knowing the existence of a path between some pair of nodes u, v, we may also want to know whether there is a *short* path.

Definition: Distance

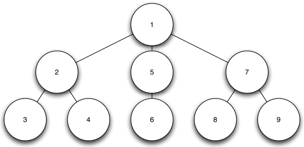
We define the distance between two nodes u and v to be the minimum number of edges in a $u - v$ path. When two nodes are not connected by a path it is common to assign an ∞ distance.

Trees

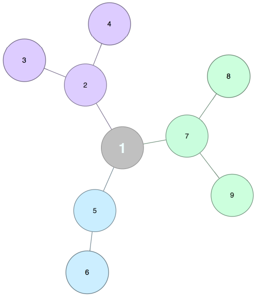
Definition: Tree

We say that an undirected graph is a tree if it is connected and does not contain any cycles.

Note: deleting any edge from a tree will disconnect it.



(a)



(b)

Figure: Two representations of the same tree

More about trees...

- A tree is generally represented starting from its *root*, r
- For each other node v we have:
  - a *parent* of v is the node u that directly precedes v on the path to r
  - a node w is *child* of v if v is the parent of w;
  - *anchestor* and *descendant*: w is a descendant of v if v lies on the path from w to r; in this case v is an anchestor of w

Proposition:

Every n-node tree has exactly $n - 1$ edges

Proof.

Each node other than the root has a single edge leading “upward” to its parent; conversely, each edge leads upward form precisely one non-root node. □

More about trees...

Here's a very important proposition that we will not prove here:

Proposition:

Let G be an undirected graph on n nodes. Any two of the following statements implies the third.

- G is connected
- G does not contain a cycle
- G has $n - 1$ edges

Graph connectivity and graph traversal

Suppose we are given a graph $G = (V, E)$ and two particular nodes s and t.

s-*t* Connectivity:

Is there a path from s to t in G? Is there an efficient algorithm that answers to this question?

In the following, we describe two natural algorithms for this problem at a high level:

- Breadth-first Search (BFS)
- Depth-first Search (DFS)
- Data structures to hold a graph G and use it as input to the algorithms

Breadth-first Search (1)

This is the simplest algorithm for determining *s*-*t* connectivity: we explore outward from *s* in all possible directions, adding nodes one "layer" at a time. Thus, we start with *s* and include all nodes that are joined by an edge to *s* (first layer). We then include all additional nodes that are joined by an edge to ay node in the first layer (second layer). And so on...

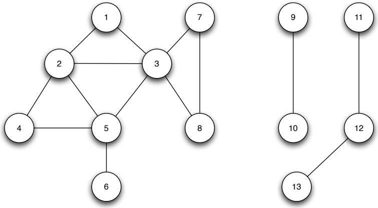


Figure: An example of a graph

Breadth-first Search (2)

As the previous example reinforces, there is a physical interpretation to the BFS algorithm. Essentially, we start at s and then *flood* the graph with an expanding wave that grows to visit all nodes it can reach. The layer L_i containing a node represents the point in time at which the node is reached.

Proposition:

For each $j \geq 1$, layer L_j produced by BFS consists of all nodes at distance exactly j from s. There is a path from s to t if and only if t appears in some layer.

Breadth-first Search (3)

Definition: the BFS Tree

BFS naturally produces a tree T rooted at s on the set of nodes reachable from s. Consider the moment when a node v is first “discovered”: this happens when some node u in layer L_j is being examined and we find it has an edge to a previously unexamined node v. At this moment, we add the edge (u, v) to the tree T (u becomes the parent of v).

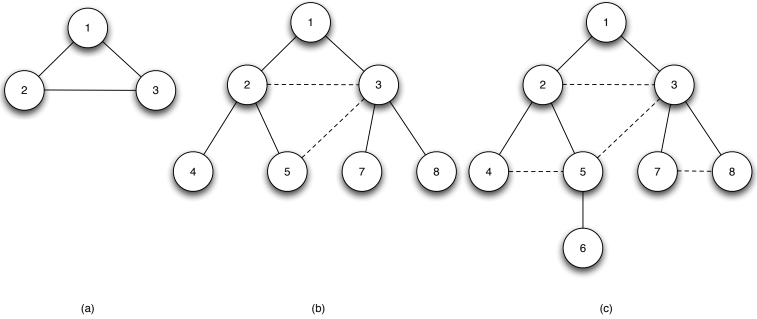


Figure: The BFS-tree when $s = 1$.

Breadth-first Search (4)

Proposition:

Let T be a BFS-tree, let x an y be nodes in T, with $x \in L_i$ and $y \in L_j$, and let (x, y) be an edge of G. Then i and j differ at most 1.

Proof.

- Suppose i, j differs by more than 1: e.g. $i < j - 1$
- Since $x \in L_i$, the only nodes discovered by x belong to L_{i+1} and earlier
- Hence if y is a neighbor of x in G, it should have been discovered by this time, that is $y \in L_{i+1}$ or earlier.

□

Exploring a connected component

Definition:

The set of nodes discovered by BFS is precisely those reachable from the starting node s. We will refer to this set R as the *connected component* of G containing s.

Algorithm 1: An algorithm to obtain the connected component.

```
R will consist of nodes to which s has a path
Initially $R = \{s\}$
while there is an edge $(u, v)$ where $u \in R$ and $v \notin R$ do
    Add $v$ to $R$
end
```

Exploring a connected component, continued ...

Proposition:

The set R produced at the end of the algorithm is precisely the connected component of $G = (V, E)$ containing s.

Proof.

- It's easy to show that $\forall v \in R$ there is a path from s to v.
- Consider $w \notin R$ and suppose (contradiction) there is a $s - w$ path P in G
- Since $s \in R$ but $w \notin R$ there must be $v \neq s \in P, \notin R$.
- Hence there must be $u \in P$ immediately before v, so that $(u, v) \in E$
- We must have $u \in R$ since v is the first node on P that is not in R
- It follows that: (u, v) is an edge where $u \in R$ and $v \notin R$, which contradicts the stopping rule of the algorithm

□

Depth-first Search (1)

Isn’t there another way to explore a graph instead of flooding it? We could take a node s then try the first edge leading out of it, to a node v. Then we could follow the first edge leading out of v and continue this way until we reach a “dead end”. We’d then backtrack until we get back to a node with an unexplored neighbor and resume from there.

We call this approach the Depth-first Search approach since it explores a graph G by going as deeply as possible and only retreating when necessary.

Depth-first Search (2)

Here's an example of recursive pseudo-code for DFS:

Algorithm 2: DFS(u)

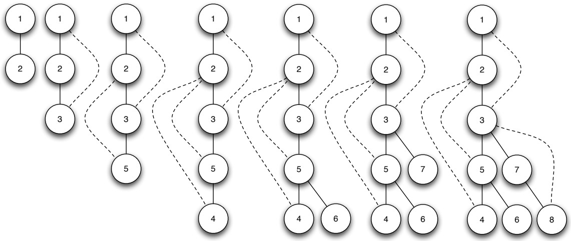
```
Mark $u$ as "Explored" and add $u$ to $R$
foreach edge $(u, v)$ incident to $u$ do
    if $v$ is not marked "Explored" then
        Recursively invoke DFS($v$)
    end
end
```

Depth-first Search (3)

While DFS visits the same set of nodes as in BFS, it does so in a very different order.

Definition: The DFS Tree

DFS produces a tree T rooted at s, where we make u the parent of v if u is responsible for the discovery of v. That is, whenever DFS(v) is invoked during the call to DFS(u), we add the edge (u, v) to T.



Depth-first Search (4)

Proposition:

For a call DFS(u) all nodes marked “Explored” between the invocation and end of the call are descendant of u in T.

Proposition:

Let T be a DFS tree, let $x, y \in T$ and let $(x, y) \in G$ that is **not** in T. Then one of x or y is an ancestor of the other.

Proof.

- Suppose $(x, y) \in G, \notin T$ and that x is reached first by DFS.
- When (x, y) is examined during DFS(x), it is not added to T because y is marked “Explored”
- Since y was not marked “Explored” when DFS(x) was first invoked, it is a node that was discovered between the invocation and the end of DFS(x)
- It follows from the previous proposition that y is a descendent of x

□

The set of ALL connected components

So far we have been talking about the connected component containing a particular node s. But there is a connected component associated with each node in the the graph. What is the relationship between these components?

Proposition:

For any two nodes s and t in a graph, their connected components are either identical or disjoint.

Proof.

- Consider s and t in a graph such that there exists an $s - t$ path
- We claim that the connected components containing s and t are the same set
- $\forall v \in R(s)$, v must be reachable from t by a path
- Then we can just walk from t to s and then from s to v
- The same reasoning works in the opposite way. Hence a node is in the component of one if and only if it is also in the component of the other
- If there is no $s - t$ path, then the above v cannot exists, otherwise we would have a $s - t$ path. Hence the connected components are disjoint.

□

Implementing Graph Traversal using Queues and Stacks

So far we have been discussing basic algorithmic primitives for working with graphs without mentioning any implementation details.

Here's what's next:

- How do we represent graphs?
- How do we implement BFS and DFS in practice?
- Brief overview on: Queues and Stacks

Representing Graphs

There are two basic ways of representing graphs:

- Adjacency matrix
- Adjacency list

A graph $G = (V, E)$ has two natural input parameters, the number of nodes $|V|$ and the number of edges $|E|$. In the following we will use the notation: $n = |V|$ and $m = |E|$.

- Running times of algorithms on graphs will be given as a function of these two input parameters, n and m
- We will aim for polynomial running times, and lower degree polynomials are better
- What can we do when we have two parameters for the running time?

Running times

- With at most one edge per any pair of nodes we will have at most $\binom{n}{2} \leq n^2$ edges
- As in many applications the graph is connected then we must have at least $m \geq n - 1$

We will use both n and m as measures of running time. When the running time is **linear** we will express it as $O(m + n)$.

Adjacency matrix

Consider a graph $G = (V, E)$ with n nodes, and assume the set of nodes is $V = \{1, \cdots, n\}$.

Adjacency matrix

An adjacency matrix A is an $n \times n$ matrix, where $A[u, v]$ is equal to 1 if the graph contains the edge (u, v) and 0 otherwise.

If the graph is undirected, then A is **symmetric**, with $A[u, v] = A[v, u]$ for all nodes $u, v \in V$.

Adjacency matrix: properties

The adjacency matrix representation:

+ allows to check in $O(1)$ time if a given edge (u, v) is present in the graph
- takes $\Theta(n^2)$ space. When a graph has many fewer edges than n^2, more compact representations are possible
- allows to check in $\Theta(n)$ the number of edges incident to a node v; for all other nodes w it requires to check if $A[v, w] = 1$. Since many graphs have fewer edges incident to most nodes, it would be great to find them more efficiently

Adjacency list

This representation works very well for **sparse graphs**, i.e. those with many fewer than n^2 edges.

Adjacency list

In an adjacency list there is a record for each node v, containing a list of the nodes to which v has edges.

We have an array `Adj`, where `Adj`[v] is a record containing a list of all nodes adjacent to node v.

For an undirected graph $G = (V, E)$, each edge $e = (v, w) \in E$ occurs on two adjacency lists: node w appears on the list for node v, and node v appear on the list for node w.

Adjacency list: properties

Adjacency lists require $O(m + n)$ space

Define the degree n_v of a node v to be the number of incident edges it has.

The length of the list at `Adj`$[v]$ is n_v, so the total length over all nodes is $O(\sum_{v \in V} n_v)$.

Sum of node degrees:

The sum of degrees in a graph is: $\sum_{v \in V} n_v = 2m$

Proof.

Each edge $e = (v, w)$ contributes exactly twice to this sum: once in the quantity n_v and once in the quantity n_w. Since the sum is the total of the contributions of each edge is $2m$. □

Adjacency list: properties, continued...

- Checking the existence of an edge (u, v) takes a time proportional to $O(n_v)$
- It requires constant time to read the list of neighbors of a given node u

The adjacency list is a natural representation for exploring graphs: as we have seen, we can explore the list of neighbors of a node u in constant time; moving to a neighbor v once we have seen it on the list can be done in constant time, to be then ready to read the list of neighbors of v again in constant time.

Exploring adjacency lists:

The adjacency list corresponds to the physical notion of “exploring” a graph in which you learn the neighbors of a node u once you arrive at u.

Queues and Stacks

Many algorithms have an inner step in which they need to process a set of elements: e.g. the set of all edges adjacent to a node, the set of all visited nodes in BFS, the set of all free men in the SMP, ...

Elements of a linked list:

One important issue that arises is the order in which to consider the elements in a list.

In some cases (SMP), the order is not important, while in other contexts (DFS, BFS) the order in which elements are considered is crucial.

- **Queue**: FIFO ordering
- **Stack**: LIFO ordering

Implementing BSF

```
BFS(s)
Set Discovered[$s$] = true and Discovered[$v$] = false for all other $v$
Initialize $L[0]$ to the single element $s$
Set the layer counter $i = 0$
Set the current BFS tree $T = \emptyset$
while $L[i]$ is not empty do
    Initialize an empty list $L[i+1]$
    foreach node $u \in L[i]$ do
        Consider each edge $(u, v)$ incident to $u$
        if Discovered[$v$] = false then
            Set Discovered[$v$] = true
            Add edge $(u, v)$ to the tree $T$
            Add $v$ to the list $L[i+1]$
        end
    end
    Increment the layer counter $i$ by one
end
```

Running time of BSF

Proposition:

The previous implementation of BFS runs in $O(m + n)$ if the graph is in the adjacency list representation.

Proof.

We first prove a weaker bound on the running time.

- There are at most n lists $L[i]$ to set up, this requires $O(n)$
- When considering node u we explore all its edges (in $O(1)$ time since we have an adjacency list)
- There can be at most n edges incident to node u so the total time spent in the `For` loop is $O(n)$

Hence we have that BFS runs in $O(n^2)$ □

Running time of BSF: a tighter bound

Proof.

- The `For` loop can take less than $O(n)$ time if u has only few neighbors!
- Let n_u be the node degree for node u. Then the time spent for node u in the `For` loop is $O(n_u)$
- Summing over all neighbors we have $O(\sum_{u \in V} n_u)$
- Since $\sum_{u \in V} n_u = 2m$ we get an $O(m)$ time
- We need an $O(n)$ time to set up lists and manage the array `Discovered`
- The total time is $O(m + n)$

$\square$

Implementing DFS

```
DFS(s)
Initialize $S$ to be a stack with one element $s$
while $S$ is not empty do
    Take a node $u$ from $S$
    if Explored[$u$] = false then
        Set Explored[$u$] = true
        foreach  edge $(u, v)$ incident to $u$ do
            Add $v$ to the stack $S$
        end
    end
end
```

Running time of DFS

Proposition:

The previous implementation of DFS runs in $O(m+n)$.

Proof.

- The main step in the algorithm is to add and delete nodes to and from the stack S, which takes $O(1)$ time.
- How many elements ever get added to S?
- Let n_v denote the degree of node v. Node v will be added to the stack every time one of its n_v neighbors is explored, hence the total number of nodes added to S is $\sum_u n_v = 2m$

$\square$

Further notes on DFS

We have seen previously that there is an alternative way of implementing DFS through recursion. Here we have seen an implementation that is similar to that of BFS, but instead of using a queue we used a stack.

Note that DFS is under-specified!! The adjacency list of a node being explored can be processed in any order. The previous algorithm, since it pushes all adjacent nodes onto the stack before considering any of them processes each list in reverse order w.r.t. the recursive version of DFS

Question:

How would you build the DFS tree exploiting the previous algorithm?

Homework 1:

The set of **all** connected components:

Briefly explain how to use BFS or DFS to find the set of all connected components and provide a bound on the running time of such an algorithm.

Testing Bipartiteness:

Recall the def. of a bipartite graph: it is one where the node set V can be partitioned into sets X and Y in such a way that every edge has one end in X and the other in Y.

We could imagine of nodes in X to be colored in red and nodes in Y to be colored in blue. With this imagery, we can say a graph is bipartite if it is possible to color its nodes red and blue so that every edge has one red end and one blue end.

The problem:

Suppose we are given a graph G with no particular annotation: can we design an algorithm to determine if G is bipartite?

Testing Bipartiteness, continued

How difficult is to test bipartiteness? What obstacles can we meet?

- Using the coloring imagery discussed before, it is easy to see that a triangle cannot be bipartite
- More generally, consider a cycle C of odd length with nodes numbered $1, 2, ..., 2k, 2k + 1$. It is easy to see that alternatively coloring nodes lead to the conclusion that the last and first node will be of the same color

Observation:

If a graph G is bipartite, then it cannot contain an odd cycle.

Testing Bipartiteness: designing the algorithm

Assumption:

Assumption: $G = (V, E)$ is connected, for otherwise we can compute its connected components and process them separately

Procedure:

- Pick a random node $s \in V$ and color it red
- All neighbors of s will be colored blue
- All neighbors of these nodes will be colored red
- and so on...

Does this procedure look familiar to you?

Testing Bipartiteness: designing the algorithm, continued ...

The previous procedure is essentially identical to BFS: we moved outward from s, coloring nodes as soon as we first encounter them.

- We perform BFS coloring s in red
- Then all nodes in layer L_1 in blue
- Then all nodes in layer L_2 in red
- and so on

We just need to add an extra array `Color` to the implementation of BFS we saw before. Whenever we get to a step in BFS where we add a node v to a list $L[i+1]$ we assign `Color`$[v]$ = red if $i+1$ is an even number and blue otherwise.

Testing Bipartiteness: analyzing the algorithm

Proposition:

Let G be a connected graph and let $L_1, L_2, ...$ be the layers produced by BFS starting at node s. Then exactly one of the following two things must hold.

1. There is no edge of G joining two nodes of the same layer. In this case G is a bipartite graph in which the nodes in even-numbered layers can be colored red, and the nodes in odd-numbered layers can be colored blue
2. There is an edge of G joining two nodes of the same layer. In this case, G contains an odd-length cycle, hence it cannot be bipartite

Testing Bipartiteness: analyzing the algorithm, continued...

Proof.

First part:

- By a previous proposition we know that every edge of G joins nodes either in the same layer or in adjacent layers. In 1) we are in the second case.
- Our coloring procedure gives nodes in adjacent layers opposite colors
- Every edge will have ends of opposite color, hence G is bipartite

$\square$

Testing Bipartiteness: analyzing the algorithm, continued...

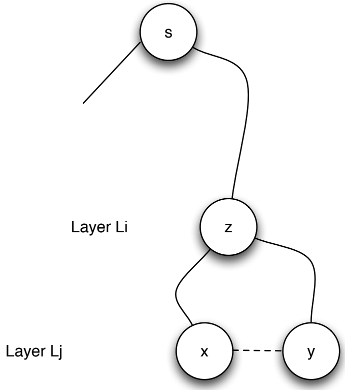
Proof.

Second part:

- We are told G contains an edge joining two nodes of the same layer
- Consider the path (a cycle!) following the $z - x$ path in T, then the edge e linking $x - y$ and then the $y - z$ path in T
- The length of this cycle is $(j - i) + 1 + (j - i) = 2(j - 1) + 1$ which is odd
- By our previous observation, if a graph contains an odd cycle then it cannot be bipartite

□

Testing Bipartiteness: analyzing the algorithm, continued...



Part III: Graph generation, a.k.a. Network models

Introduction (1)

- Finding suitable models for the real world is the primary goal here; another goal is to design algorithms to build such networks
- First we will start with some slightly unrealistic models and then approach real-world models
- But what are real-world networks? We will deal with networks that mostly fall into three categories: the Internet, biological networks and social networks

In this course we will mostly deal with Internet-like networks and social networks.

Introduction (2)

- Many - but not all - of these examples from different areas have some characteristics in common: For example metabolics, the WWW, and co-authorship often form networks that have very few vertices with very high degree, some of considerable degree and a huge number of vertices with very low degree
- Unfortunately, the data is sometimes forced to fit into that shape, or even mischievously interpreted to show a so called power law
- Often deeper results are not only presented without proof, but also only based on so called experimental observations

Introduction (3)

- An important observations is: most of the real-world networks are intrinsically historical; they did not come into being as a complete and fixed structure at one single moment in time, but they have developed step by step; They emerged
- Therefore, on the one hand, it makes sense to understand the current structure as the result of a **growth process**
- On the other hand, one is often more interested in the network's future than in one of its single states. Therefore several models have been developed that define a graph, or a family of graphs, via a process in the course of which they emerge

Fundamental models

Random Graphs (1)

We now define the graph model $G_{n,p}$.

- A graph model is a set of graphs endowed with a probability distribution
- In this case the graphs under consideration are undirected.

The following three graph models stochastically converge to each other as $n \to \infty$.

Random Graphs (2)

1. **Input: n nodes and average degree z:**
   The first way to generate a random graph is to choose a graph uniformly at random among all graphs of given vertex number n and average vertex degree z.
2. **Input: n nodes and probability distribution of node degree:**
   Alternatively, choose every edge in a complete graph of n vertices with probability p to be part of $E(G)$, where $\frac{2p\binom{n}{2}}{n}$ is the expected average degree. This model is denoted by $G_{n,p}$.
3. **Input: growth model with n nodes and probability distribution of node degree:**
   In the third method, n vertices v_i are added successively, deciding for each v_i and for each $j < i$ whether to put $\{v_i, v_j\}$ in the edge set or not with probability p.

Random graphs: discussion

- There is a myriad of literature and highly developed theory on the $G_{n,p}$ and related models
- It turns out that a graph chosen according to that distribution, a graph “generated” by that model, shows a number of interesting characteristics with high probability
- On the other hand, this graph model has, precisely because of these characteristics, often been disqualified as a model for real-world networks that usually do not show these characteristics
- For example, without deep mathematical consideration one can see that the majority of the vertices will have almost or exactly the same average degree. For many networks in the real world this is not the case

Social Networks: a.k.a. small world networks

One of the starting points of network analysis is a sociological experiment conducted to verify the urban legend that anyone indirectly knows each other by just a few other mediators (Milgram experiment). The notion of “Small World” has become technical since, usually encompassing two characteristics:

- the **average shortest path** distances over all vertices in a small world network has to be small, that is it grows at most logarithmically with the number of vertices
- Mathematically speaking a network shows the worldly aspect of a small world if it has a high clustering coefficient[1]

[1] The clustering coefficient gives the fraction of pairs of neighbors of a vertex that are adjacent, averaged over all vertices of the graph

Small world networks (1)

A very popular abstract model of small world networks, i.e., a graph with clustering coefficient bounded from below by a constant and logarithmically growing average path distance, is obtained by a simple **rewiring procedure.**

Definition:

The kth power of a cycle is a graph where each vertex is not only adjacent to its direct neighbors but also to its k neighbors to the right and k neighbors to the left.

Small world networks (2)

```
Start with the $k$th power of an $n$-cycle, denoted by $C_n^k$.
foreach edge $\{a, b\} \in C_n^k$ do
    Decide independently by a given probability $p$ whether to keep it in
    place or to replace the edge $\{a, b\}$ by an edge $\{a, c\}$ where $c$ is
    chosen uniformly at random from the vertex set
end
```

Hey, but there's some *ambiguity* here!!!

Small world networks (3)

- Viewing the rewiring process as iteratively passing through all vertices, one may choose an edge to be rewired from both of its vertices
- The natural way to straighten this out is the following:
  Visit each vertex iteratively in some order, and make the rewiring decisions for each of the currently incident edges

Therefore, strictly speaking, the model depends on the order in which the vertex set is traversed. Anyway, you should be confident that this does not affect the outcome we are interested in, namely the average shortest path distance and the clustering coefficient C.

Part IV: Network Analysis

Fundamentals

We still need to overview a couple of details on graphs.

- Weighted graphs
- Single source shortest path: Dijkstra algorithm, Bellman-Ford algorithm
- Network flows
- k-connectivity
- Graph algebra

Introduction

Network analysis is carried out in areas such as project planning, complex systems, electrical circuits, social networks, transportation systems, communication networks, epidemiology, bioinformatics, hypertext systems, text analysis, bibliometrics, organization theory, genealogical research and event analysis.

Question:

What is network analysis?

Outlook:

Element Analysis:

Study characteristics of elements of a network (or graph), i.e. nodes and edges.

Group Analysis:

Study the characteristics of groups, or clusters that emerge from a network.

Network Statistics and Comparison:

Globally characterize a network, hence comparison between graphs become possible.

Element analysis: introduction (1)

Let's begin with a practical example.

Search engines on the Web index large numbers of documents to answer keyword queries by returning documents that appear **relevant** to the query.

The success of a search engine is thus crucially dependent on its definition of relevance.

Contemporary search engines use a weighted combination of several criteria: the number, position, and markup of keyword occurrences, their distance and order in the text, or the creation date of the document, a **structural measure of relevance** employed by market leader Google turned out to be most successful.

Element analysis: introduction (2)

- **The Web Graph**: consider the graph consisting of a vertex for each indexed document, and a directed edge from a vertex to another vertex, if the corresponding document contains a hyperlink to the other one
- Since a link corresponds to a referral from one document to another, it embodies the idea that the second document contains relevant information
- It is thus reasonable to assume that a document that is often referred to is a relevant document, and even more so, if the referring documents are relevant themselves
- Technically, this (structural) relevance of a document is expressed by a positive real number, and the particular definition used by Google is called the **PageRank** of the document.

Element analysis: introduction (3)

As for Page Rank, similar valuations of vertices and also of edges of a graph have been proposed in many application domains:

- Which is the most important element of a graph?
- How important is this element?

These questions are typically addressed using concepts of structural centrality, but while a plethora of definitions have been proposed, no general, comprehensive, and accepted theory is available. In the following we will see some centrality indexes in more details...

Element analysis: centrality indexes

Centrality indices are to quantify an intuitive feeling that in most networks some vertices or edges are more central than others.

- First we'll need to define which properties a centrality index must satisfy
- Then we'll focus on some examples of **vertex** and **edges** centralities
- There are many families of centralities: we will see those based on distances, on paths, and finally on feedback

Centrality indexes: an introductory example

What is it that makes a vertex central and another vertex peripheral? Centrality can be interpreted as “influence”, as “prestige” or as “control”. For example, a vertex can be regarded as central if it is heavily required for the transport of information within the network or if it is connected to other important vertices.

Example: election leader

A school class of 30 students has to elect a class representative and every student is allowed to vote for one other student.

Example: election leader (1)

We can derive different graph abstractions from this situation that can later be analyzed with different centrality indices.

Election leader: a first interpretation.

We will first look at a network that represents the voting results directly. In this network vertices represent students and an edge from student *A* to student *B* is established if *A* has voted for *B*.

In such a situation, a student could be said to be the more "central" the more people have voted for him or her. This kind of centrality is directly represented by the number of edges pointing to the corresponding vertex. The so called **in-degree centrality** will be discussed in more detail in the following.

Example: election leader (2)

Election leader: second interpretation.

We now look at a network in which an edge between *A* and *B* represents that student *A* has convinced student *B* to vote for his or her favorite candidate. We will call this network an “influence network”.

Assume the class split into two groups *X* and *Y*. Let some person have a relation to members from both groups but has a favorite candidate from group *X* and convinces a big part of group *Y* to vote for this candidate. With this argument we can say that a vertex in the given influence network is the more central the more it is needed to transport the opinion of others. A family of centrality indices that captures this intuition is the family of **betweenness centrality indices**.

Example: election leader (3)

Election leader: third interpretation.

In yet another perspective we could view the social network of the class: Who is friends with whom?

Someone who is a friend of an important person could be regarded as more important than someone having friends with low social prestige. The centrality of a vertex in this kind of network is therefore given by the centrality of adjacent vertices. This kind of **feedback centrality** is captured by many centrality indices that will be discussed in details.

Centrality indexes: what about edges?

So far we've been discussing about vertex centrality, but what about edges? There are mainly two different approaches to measure the centrality of an edge in a network:

- The first counts the number of substructures like traversal sets or the set of shortest paths in the graph on which an edge participates.
- The second approach is based on the idea of measuring how much a certain network parameter is changed if the edge is removed

Although edge centrality is very important (e.g. allows studying the **robustness** of a network), we don't have time to go into details.

Centrality indexes: a loose definition

Recall that two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic ($G_1 \simeq G_2$) if there exists a one-to-one mapping $\phi : V_1 \rightarrow V_2$ such that (u, v) is an edge of E_1 iff $(\phi(u), \phi(v))$ is an edge of G_2.

Definition: structural index

Let $G = (V, E)$ be a weighted, directed or undirected multigraph and let X represent the set of vertexes or edges of G, respectively. A real-valued function s is called a structural index iff the following condition is satisfied: $\forall x \in X : G \simeq H \Rightarrow s_G(x) = s_H(\phi(x))$, where $s_G(x)$ denotes the value of $s(x)$ in G.

Centrality indexes: distance and neighborhoods

In this section we will present centrality indices that evaluate the "reachability" of a vertex. Given any network these measures rank the vertices according to the number of neighbors or to the cost it takes to reach all other vertices from it. These centralities are directly based on the notion of distances within a graph, or on the notion of neighborhood, as in the case of the degree centrality.

Degree centrality

The most simple centrality is the degree centrality $c_D(v)$ of a vertex v that is simply defined as the degree $d(v)$ of v if the considered graph is undirected. The degree centrality is a local measure, because the centrality value of a vertex is only determined by the number of its neighbors.

Centrality indexes: a first application

Facility location analysis deals with the problem of finding optimal locations for one or more facilities in a given environment. Location problems are classical optimization problems with many applications in industry and economy.

As compared to what we have seen in Lecture 1, there exist several ways to classify location problems.

Our goal here is not to study facility location problems but to introduce three important vertex centralities by examining location problems.

Centrality indexes: facility location

- **minimax criterion**: e.g. consider the problem of determining the location for an hospital. The main objective of such a facility location problem is to find a site that minimizes the maximum response time between the facility and the site of a possible emergency
- **minisum criterion**: e.g. consider the problem of determining the location for a shopping mall. The aim here is to minimize the total travel time
- **competitive criterion**: e.g. this deals with the location of commercial facilities which operate in a competitive environment. The goal of a competitive location problem is to estimate the market share captured by each competing facility in order to optimize its location

Centrality indexes: facility location - Assumptions

The definition of different objectives leads to different centrality measures. A common feature, however, is that each objective function depends on the distance between the vertices of a graph.

Assumptions:

1. $G = (V, E)$ is connected
2. G is undirected
3. G is unweighted
4. $d(u, v)$ defined the distance between two vertices u, v as the length of the shortest path from u to v.

Centrality indexes: eccentricity

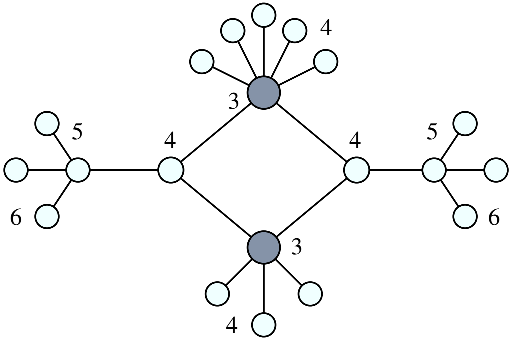
The aim of the first problem family is to determine a location that minimizes the maximum distance to any other location in the network. Suppose that a hospital is located at a vertex $u \in V$.

Eccentricity:

We denote the maximum distance from u to a random vertex v in the network as the eccentricity $e(u)$ of u, where $e(u) = \max\{d(u, v) : v \in V\}$.

Note that v represents the location of a possible accident. The problem of finding an optimal location can be solved by determining the minimum over all $e(u)$ with $u \in V$.

Centrality indexes: eccentricity example



Centrality indexes: eccentricity, formal definition

Definition:

A centrality measure based on the eccentricity can be defined as:

$$c_E(u) = \frac{1}{e(u)} = \frac{1}{\max\{d(u,v) : v \in V\}}$$

This measure is consistent with our general notion of vertex centrality, since $e^{-1}(u)$ grows if the maximal distance of u decreases.
Thus, for all vertices $u \in V$ of the center of G: $c_E(u) \geq c_E(v) \forall v \in V$.

Centrality indexes: closeness

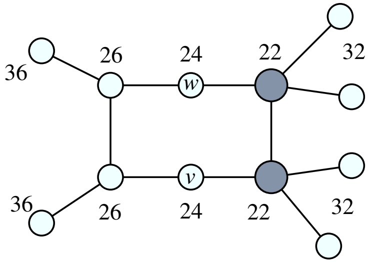
Next we consider the second type of location problems often also called the service facility location problem. Suppose we want to place a service facility such that the total distance to all customers in the region is minimal. This would make traveling to the mall as convenient as possible for most customers.

Definition:

We denote the sum of the distances from a vertex $u \in V$ to any other vertex in a graph $G = (V, E)$ as the total distance $\sum_{v \in V} d(u, v)$.

The problem of finding an appropriate location can be solved by computing the set of vertices with minimum total distance.

Centrality indexes: closeness



Centrality indexes: closeness

In social network analysis a centrality index based on this concept is called closeness. The focus lies here, for example, on measuring the closeness of a person to all other people in the network. People with a small total distance are considered as more important as those with a high total distance.

Definition:

The most commonly employed definition of closeness is the reciprocal of the total distance:

$$c_C(u) = \frac{1}{\sum_{v \in V} d(u, v)}$$

Centrality indexes: centroids

The last centrality index presented here is used in competitive settings. Suppose each vertex represents a customer in a graph. Competitive location problems deal with the planning of commercial facilities which operate in a competitive environment.

For reasons of simplicity, we assume that the competing facilities are equally attractive and that customers prefer the facility closest to them. Consider now the following situation: A salesman selects a location for his store knowing that a competitor can observe the selection process and decide afterwards which location to select for her shop. Which vertex should the salesman choose?

Centrality indexes: centroids

Given a connected undirected graph G of n vertices, for a pair of vertices (u, v), $\gamma_u(v)$ denotes the number of vertices which are closer to u than to v. That is:

$$\gamma_u(v) = |\{w \in V : d(u, w) < d(v, w)\}|$$

If the salesman selects a vertex u and his competitor a vertex v, then he will have

$$\gamma_u(v) + \frac{1}{2}(n - \gamma_u(v) - \gamma_v(u)) =$$
$$\frac{1}{2}n + \frac{1}{2}(\gamma_u(v) - \gamma_v(u))$$

customers.
Let $f(u, v) = (\gamma_u(v) - \gamma_v(u))$, then the competitor will select a vertex v that will minimize $f(u, v)$. But the salesman knows this strategy and calculates for each vertex u the worst case, that is

$$c_F(u) = \min\{f(u, v) : v \in V - u\}$$

Centrality indexes: centroids

$c_F(u)$ is called the **centroid value** and measures the advantage of the location u compared to other locations, that is the minimal difference of the number of customers which the salesman gains or loses if he selects u and a competitor chooses an appropriate vertex v different from u.

Centrality indexes: shortest paths

The first centrality index based on enumeration of shortest paths is **stress centrality** $c_S(x)$. Here we are concerned with the question of how much “work” is done by each vertex in a communication network. The assumption is that counting the number of shortest path that contain an element x gives an approximation of the amount of “work” or “stress” the element has to sustain in the network. With this, an element is the more central the more shortest paths run through it.

Centrality indexes: stress centrality

Definition:

Formally the stress centrality is defined as:

$$c_S(v) = \sum_{s \neq v \in V} \sum_{t \neq v \in V} \sigma_{st}(v)$$

where $\sigma_{st}(v)$ denotes the number of shortest paths through v.

The calculation of this centrality index is given by a variant of a simple all-pairs shortest-paths algorithm (the **Floyd-Warshall algorithm**) that not only calculates one shortest path but all shortest paths between any pair of vertices.

Centrality indexes: Shortest-Path Betweenness Centrality

Shortest-path betweenness centrality can be viewed as some kind of relative stress centrality. Here, we will first define it and then discuss the motivation behind this centrality index: let $\delta_{st}(v)$ denote the fraction of shortest paths between s and t that contain vertex v:

$$\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$$

where σ_{st} denotes the number of all shortest paths between s and t.

The ratio $\delta_{st}(v)$ can be interpreted as the probability that vertex v is involved into any communication between s and t. Note, that the index implicitly assumes that all communication is conducted along shortest paths.

Centrality indexes: Shortest-Path Betweenness Centrality

Definition:

Formally, the betweenness centrality is defined as:

$$c_B(v) = \sum_{s \neq v \in V} \sum_{t \neq v \in V} \delta_{st}(v)$$

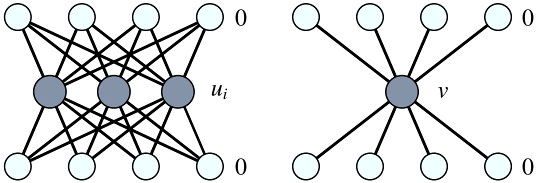
As for stress centrality, the shortest paths ending or starting in v are explicitly excluded. The motivation for this is that the betweenness centrality of a vertex measures the control over communication between others.

Centrality indexes: Shortest-Path Betweenness Centrality

The next Figure gives an example why this definition might be more interesting than the one using the absolute number of shortest paths.

It shows two tripartite graphs in which the middle layer mediates all communication between the upper and the lower layer. The stress centrality of vertices in the middle layer is the same in both graphs but the removal of the middle vertex on the right would disconnect the whole system whereas in the left graph the removal of a single vertex would not. This is because the former has full responsibility for the communication in its graph whereas on the left side every vertex just bears one third of it.

Centrality indexes: Shortest-Path Betweenness Centrality



(left): $c_S(u_i) = 16$ and $c_B(u_i) = 1/3 \forall i = 1, 2, 3$,
(right): $c_S(v) = 16$ but $c_B(v) = 1$

Centrality indexes: feedback

This section presents centralities in which a node is the more central the more central its neighbors are.

Note that in the following centrality indices will be denoted as vectors. All feedback centralities are calculated by solving linear systems, such that the notation as a vector is much more convenient than using a function expressing the same. We just want to state here that all centrality indices presented here are fulfilling the definition of a structural index.

Note also that due to lack of time we will focus only on **web centralities**.

Centrality indexes: Web Centralities

Due to the immense size of the Web Graph, powerful search engines are required. It is necessary to score the Web pages according to their relevance: this is first done by a pure text search within the content of the pages; then, we use the structure of the network to rank pages and this is where centrality indices come into play.
In this section we discuss the Page Rank Web-scoring algorithms.
PageRank only takes the topological structure into account, while other recent ranking metrics combines the “textual importance” of the Web page with its “topological importance”.

Centrality indexes: Random surfer model (1)

Before defining centrality indices suitable for the analysis of the Web graph it might be useful to model the behavior of a Web surfer. The most common model simulates the navigation of a user through the Web as as a **random walk** within the Web graph.
A random walk in a simple directed graph $G = (V, E)$ is a Markov chain with $S = V$ and

$$\Pr[X_{t+1} = v | X_t = u] = \begin{cases} \frac{1}{d^+(u)} & \text{if } (u, v) \in E \\ 0 & \text{otherwise} \end{cases}$$

Centrality indexes: Random surfer model (2)

In every step, the random walk picks a random edge leaving the current vertex and follows it to the destination of that edge. The random walk is well defined only if $d^+(v) \geq 1 \forall v \in V$. In this case, the transition matrix of the random walk is the stochastic $|V| \times |V|$ matrix $T = (t_{ij})$, where:

$$t_{ij} = \begin{cases} \frac{1}{d^+(i)} & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Note that the Markov chain given by a random walk in a directed graph G is irreducible if and only if G is strongly connected.

Centrality indexes: Random surfer model (3)

- The Web graph $G = (V, E)$ is formally defined as V the set of all Web pages p_i where an edge $e = (p_i, p_j) \in E$ exists iff p_i link to page p_j
- As the Web graph is usually not strongly connected the underlying transition matrix T is not irreducible and may not even be stochastic as sinks (vertices without outgoing links) may exist
- T has to be modified such that the corresponding Markov chain converges to a stationary distribution

To make T stochastic we assume that the surfer jumps to a random page after he arrived at a sink, and therefore we set all entries of all rows for sinks to $1/n$.

Centrality indexes: Random surfer model (4)

The definition of the modified transition matrix T' is

$$t'_{ij} = \begin{cases} \frac{1}{d^+(i)} & \text{if } (i,j) \in E \\ \frac{1}{n} & \text{if } d^+(i) = 0 \end{cases}$$

This matrix is stochastic but not necessarily irreducible and the computation of the stationary distribution π' may not be possible. We therefore modify the matrix again to get an irreducible version T''. Let $E = \frac{1}{n}\mathbf{1}_n^T\mathbf{1}_n$ be the matrix with all entries $1/n$. This matrix can be interpreted as a "random jump" matrix. Every page is directly reachable from every page by the same probability. To make the transition matrix irreducible we do:

$$T'' = \alpha T' + (1-\alpha)E$$

Centrality indexes: Random surfer model (5)

Factor α is chosen from the range 0 to 1 and can be interpreted as the probability of either following a link on the page by using T' or performing a jump to a random page by using E. The matrix T'' is by construction stochastic and irreducible and the stationary distribution π'' can be easily computed.

Google's PageRank (1)

The main idea is to score a Web page with respect to its topological properties, i.e., its location in the network, but independent of its content. PageRank is a feedback centrality since the score or centrality of a Web page depends on the number and centrality of Web pages linking to it:

$$\mathbf{c}_{PR}(p) = d \sum_{q \in \Gamma_p^-} \frac{\mathbf{c}_{PR}(q)}{d^+(q)} + (1 - d)$$

where $\mathbf{c}_{PR}(q)$ is the Page Rank of page q and d is a damping factor.

Google's PageRank (2)

The corresponding notation of the Page Rank is:

$$\mathbf{c}_{PR} = dP\mathbf{c}_{PR} + (1-d)\mathbf{1}_n$$

where the transition matrix P is defined by

$$p_{ij} = \begin{cases} \frac{1}{d^+(j)} & \text{if } (j,i) \in E \\ 0 & \text{otherwise} \end{cases}$$

The linear system above is solved by a simple power (or Jacobi) iteration:

$$\mathbf{c}_{PR}^{k} = dP\mathbf{c}_{PR}^{k-1} + (1-d)\mathbf{1}_n$$

and there are guarantees for the convergence to a unique solution of this iteration if $d < 1$.

Google's PageRank (3)

Homework 2:

Implement in Python the PageRank algorithm: this includes the generation of a input graph (representing the Web), and the computation of the PageRank for each of its nodes.

[`Optional`:] For those who followed the Clouds course, you may also want to implement the PageRank algorithm in MapReduce (or Pig). Please refer to the Laboratory material to understand how to proceed.

Hurray! We're done with **element analysis**!!

Now, what's next?

Group analysis: clustering

- What is Clustering?

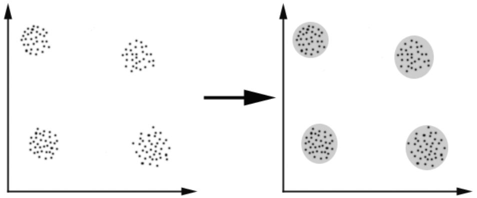
Clustering can be considered the most important unsupervised learning problem: it deals with finding a structure in a collection of unlabeled data.

A loose definition:

Clustering is the process of organizing objects into groups whose members are similar in some way.

Group analysis: clustering, continued...

A cluster is therefore a collection of objects which are “similar” between them and are “dissimilar” to the objects belonging to other clusters. We can show this with a simple graphical example:



In this case we easily identify the 4 clusters into which the data can be divided; the similarity criterion is **distance**: two or more objects belong to the same cluster if they are “close” according to a given distance (in this case geometrical distance). This is called distance-based clustering.

The goals of clustering

The goal of clustering is to determine the intrinsic grouping in a set of unlabeled data. But how to decide what constitutes a good clustering?

It can be shown that there is no absolute “best” criterion which would be independent of the final aim of the clustering. Consequently, it is the user which must supply this criterion, in such a way that the result of the clustering will suit their needs.

For instance, we could be interested in finding representatives for homogeneous groups (data reduction), in finding “natural clusters” and describe their unknown properties (“natural” data types), in finding useful and suitable groupings (“useful” data classes) or in finding unusual data objects (outlier detection).

Applications of clustering

- Marketing: finding groups of customers with similar behavior given a large database of customer data containing their properties and past buying records;
- Biology: classification of plants and animals given their features;
- City-planning: identifying groups of houses according to their house type, value and geographical location;
- WWW: document classification; clustering weblog data to discover groups of similar access patterns.

Clustering algorithms

- *k*-means algorithm
- Autonomous clustering algorithm
- ...

K-means algorithm: in words (1)

K-means (MacQueen, 1967) is one of the simplest algorithms that solve the well known clustering problem. The procedure follows a simple way to classify a given data set through a certain number of clusters (assume *k* clusters) fixed a priori.

- The main idea is to define *k* centroids, one for each cluster. These centroids should be placed in a cunning way because of different location causes different result
- So, the better choice is to place them as much as possible far away from each other. The next step is to take each point belonging to a given data set and associate it to the nearest centroid.
- When no point is pending, the first step is completed and an early groupage is done.

K-means algorithm: in words (2)

- We now need to re-calculate k new centroids as barycenters of the clusters resulting from the previous step. Then, a new binding has to be done between the same data set points and the nearest new centroid
- A loop has been generated. As a result of this loop we may notice that the k centroids change their location step by step until no more changes are done
- Centroids do not move any more

This algorithm aims at minimizing the objective function:

$$J = \sum_{j=1}^{k} \sum_{i=1}^{n} \|x_i^j - c_j\|^2$$

where $\|x_i^j - c_j\|^2$ is a distance measure between a data point x_i^j and the cluster centre c_j.

K-means algorithm

```
input  : $P = \{p_1, ..., p_n\}$
         $k$ = number of clusters
output : $C = \{c_1, ..., c_k\}$ : cluster centroids
         $m : P \rightarrow \{1, ..., k\}$ : cluster membership

Set $C$ to initial value (e.g. random position)
foreach $p_i \in P$ do
    $m(p_i) = \arg\min_{j \in \{1,...,k\}} distance(p_i, c_j)$
end
while m has changed do
    foreach $i \in \{1, ..., k\}$ do
        Recompute $c_i$ as the centroid of $\{p | m(p) = i\}$
    end
    foreach $p_i \in P$ do
        $m(p_i) = \arg\min_{j \in \{1,...,k\}} distance(p_i, c_j)$
    end
end
```

And now, let’s move to network statistics and comparison!!

Network statistics and comparison

Owing to the sheer size of large and complex networks, it is necessary to reduce the information to describe essential properties of vertices and edges, regions, or the whole graph. Usually this is done via network statistics, i.e., a single number, or a series of numbers, catching the relevant and needed information for the whole graph.
A network statistic should:

- describe essential properties of the network
- differentiate between certain classes of networks
- be useful in algorithms and applications

Degree statistics

The most common and computationally easy statistic is the vertex degree. Depending on the underlying network and its application, it may be a simple measure for the strength of connection of a specific vertex to the graph, or (as in the case of indegrees) a measure for the relevance.

- Instead of using this statistic directly, the main interest lies in the absolute number or the fraction of vertices of a given in-, out-, or total degree
- NOTE: It has been discovered that the distribution of degrees in many naturally occurring graphs significantly differs from that of classical random graphs

Degree statistics: example (1)

- Random graph:

In a classical undirected random graph $G_{n,p}$ the fraction of vertices of degree k is expected to be the **binomial distribution**

$$\binom{n-1}{k} p^k (1-p)^{n-k-1} \text{ if } n \text{ is small}$$

or the **Poisson distribution**

$$\frac{(np)^k}{k!} e^{-np} \text{ if } n \text{ is big}$$

Degree statistics: example (2)

- “Natural” graph: e.g. WWW, Internet, Friendship, ...

In many natural graphs the degree distribution seems to follow a **power law**:

$$ck^{-\gamma} \text{ with } \gamma > 0 \text{ and } c > 0$$

To characterize it, it is sufficient to determine the constant exponent γ, which can be derived with the linear regression of the log-log plot of the distribution.

Distance statistics

Another basic, but computationally more complex statistic is the distance between two vertices, defined as $d(u, v) = \min\{|P| | P \text{ is a path from } u \text{ to } v\}$.

- Arranging the distances leads to a $V \times V$-matrix D, whose columns and rows are indexed by the vertices of the graph, with

$$D = (d(u, v))_{u,v \in V}$$

- For arbitrary edge weights $w : E \to \mathbf{R}$ the problem of finding a shortest path is **NP**-hard

Distance statistics: examples (1)

- Characteristic distance:

The average or characteristic distance $\bar{d}$ is the arithmetic mean of all distances in the graph:

$$\bar{d} := \frac{1}{|V|^2 - |V|} \sum_{u \neq v \in V} d(u, v)$$

NOTE: for disconnected graphs, we have $\bar{d} = \infty$.
We also look at the **average connected distance**:

$$\bar{d} := \frac{1}{k} \sum_{u \neq v \in V, 0 < d(u,v) < \infty} d(u, v)$$

Distance statistics: examples (2)

- Neighborhoods:

The h-neighborhood $\text{Neigh}_h(v)$ of a vertex v is the set of all vertices u with distance less than or equal to h from v:

$$\text{Neigh}_h(v) := \{u \in V \mid d(u, v) < h\}$$

The (absolute) hop plot $P(h)$ eliminates the dependence on the vertex by assigning the number of pairs (u, v) with $d(u, v) \leq h$ to each parameter h:

$$P(h) := |\{(u, v) \in V^2 \mid d(u, v) < h\}| = \sum_{v \in V} N(v, h)$$

where :

$$N(v, h) := |\text{Neigh}_h(v)|$$

Clustering Coefficient

The clustering coefficient introduced by Watts and Strogatz in the year 1998 has become a frequently used tool in network analysis.

- For a node v the clustering coefficient $c(v)$ is supposed to represent the likeliness that two neighbors of v are connected
- The clustering coefficient $C(G)$ of a graph is the average of $c(v)$ taken over all nodes

Very informally, $C(G)$ of a random graph tends to 0, while it tends to 1 for small-world graphs.