CS_232 Algorithms and Complexity
Week 1, Wednesday, 3/10/2007

Oliver Kullmann

Department of Computer Science
University of Wales Swansea
Swansea, SA2 8PP, UK

e-mail: O.Kullmann@Swansea.ac.uk
http://cs.swan.ac.uk/~csoliver

November 22, 2007

Lecture held on Wednesday, October 3, 2007, in the Glyndwr Building, Lecture Hall C, from 12:00 - 13:00.

I Organisation

II Overview on the module

III Algorithms and Concepts

I Course home page

On the course home page

`http://cs-svr1.swan.ac.uk/~csoliver/AlgorithmsComplexity200708/index.html`

you find always the up-to-date versions of the slides (which serve also as the script). You will find also there all relevant news regarding the module, and hints for further study, so please visit it regularly.

Different ways to access it:

- directly via the above address;
- go to the computer science home page `http://www.swan.ac.uk/compsci/`, and follow the links "People" and "Academic Staff";
- Type in "Oliver Kullmann" into `google`, find my home page and click on the link to information on modules I teach (near the top of the page).

Assessment

Approximately at the end of week 5 and week 9 there will be course works, each having a weight of 10%, while the examination counts 80%.

The course work shall help you to ensure that you don't fall behind.

In the lecture

At the beginning of the lecture the script is handed out. Corrected versions of the script are available on the course home page (but won't be printed out again, except of cases of substantial changes).

Left-over print-outs of the script are found in the student's office, to be picked up by those who couldn't come to the lecture (hopefully due to good reasons(!)). If no print-outs are left, you can ask the secretary to make a copy from the master-copy (but please don't ask me).

The script handed out in the lecture is there to accommodate making notes, but the final, "official" version of the script is available at the course home page.

You also have to sign the module attendance form.

At the end of each lecture there will be a summary in order to facilitate the learning process.

Literature

Algorithms in Pseudo-Code:

- *Foundations of Algorithms Using C++ Pseudocode*, Richard E. Neapolitan and Kumarss Naimipour, Jones and Bartlett Publishers, 2004, Third Edition. Main recommended text.
- *Introduction to Algorithms*, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, MIT Press, 2001, Second Edition. Quite comprehensive, and covers 2/3 of the course.
- *Introduction to The Design and Analysis of Algorithms*, Anany Levitin, Addison-Wesley, 2003. An interesting book, especially due to its chapter "Coping with the Limitations of Algorithm Power".
- *Algorithm Design*, Jon Kleinberg und Éva Tardos *Algorithm Design*, Addison-Wesley, 2005. Very good explanations; contains interesting chapters on NP and PSPACE.

1. These books focus mostly on the ideas of the algorithms, and on their analysis.

2. Implementing the algorithms (efficiently, and according to modern software standards) is yet another issue, and discussed in the following books.

Literature on algorithms, supporting specific programming languages

- *Algorithms in C / C++ / Java*, Robert Sedgewick, Addison-Wesley (use always the latest edition!). Yet there are two volumes, the first contains parts 1 - 4, and focuses on fundamentals and searching and sorting, while the second volume contains part 5 and is about graph algorithms, which is especially relevant for this module. These books are well written. C++ is the arguable the most appropriate language for implementing algorithms and libraries.
- *Data Structures and Algorithms in C++ / Java*, Michael T. Goodrich, Roberto Tamassia and David Mount, John Wiley & Sons, Inc. Especially the graphical visualisations of algorithms are nice; the C++ used is not really up-to-date (but this in fact holds for most books in the area).

Literature on complexity and cryptography

- *Algorithmics: The Spirit of Computing*, David Harel (with Yishai Feldman), third edition, Addison-Wesley, 2004. This book covers a very broad range, from algorithms to complexity and cryptography to computability and software engineering. Nice to obtain an overview. Be aware that the first edition appeared 1987, and consequently the nowadays very important practical applications in NP (and beyond) are not on the horizon (and also in other respects the book is somewhat misleading regarding practical applications).
- *Complexity and Cryptography: An Introduction*, John Talbot and Dominic Welsh, Cambridge University Press, 2006. Good book to obtain a deeper understanding of complexity theory and cryptography.

How to work/learn

Use books!

Use additional sources!

A very good exercise is to implement the algorithms discussed in this module. Of course, this means a lot more work, because the ADT's (*Abstract Data Types*) and the concrete *Data Structures* have to be developed, so it is very helpful using the literature with support for programming.

Don't fall behind!

II What you will learn here

There are three main topics:

1. Graphs and graph algorithms
2. Cryptography
3. Complexity theory (with focus on P and NP)

A pervasive topic is that of algorithms for tackling hard problems.

We will not discuss implementations in detail, but abstract data types (ADT's) and data structures are important to us.

Graphs

A graph consists of

- **vertices** (alternatively "nodes") and
- **edges** connecting the vertices.

We consider

- **undirected graphs** (simply called "graphs"), and
- **directed graphs** (also called "digraphs" for short).

In addition we will also consider **general graphs** and **general directed graphs**, where (self-)loops and parallel edges are possible.

As special graph types we consider **bipartite graphs**.

Graph algorithms

We are considering algorithms related to question about

reachability How to search through a graph? Can I reach a node from another node? If yes, what is the shortest way? ...

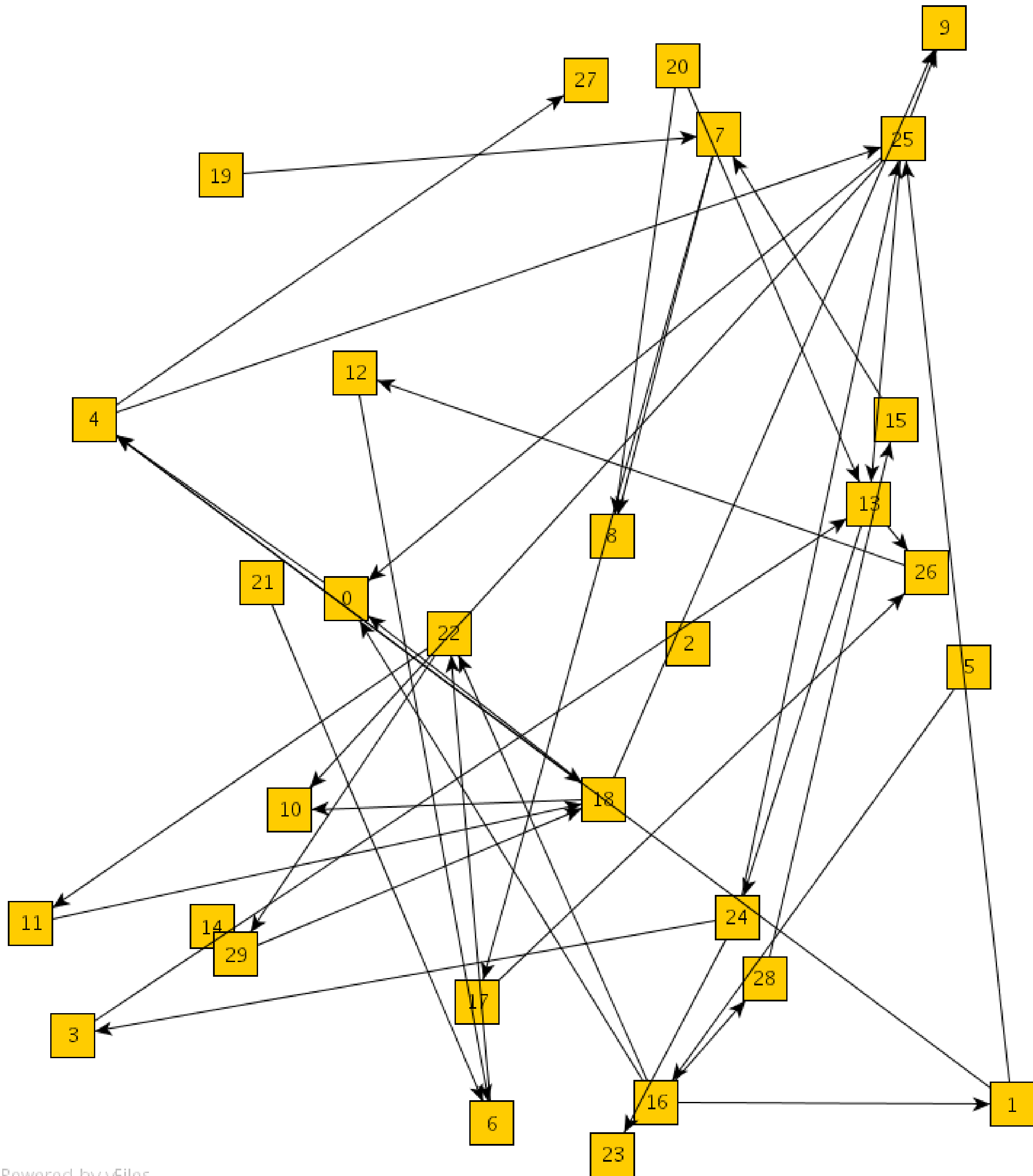
matching assignment problems and generalisations (starting from the famous "marriage problem").

colouring How can we assign colours to vertices of a graph such that connected vertices get different colours?

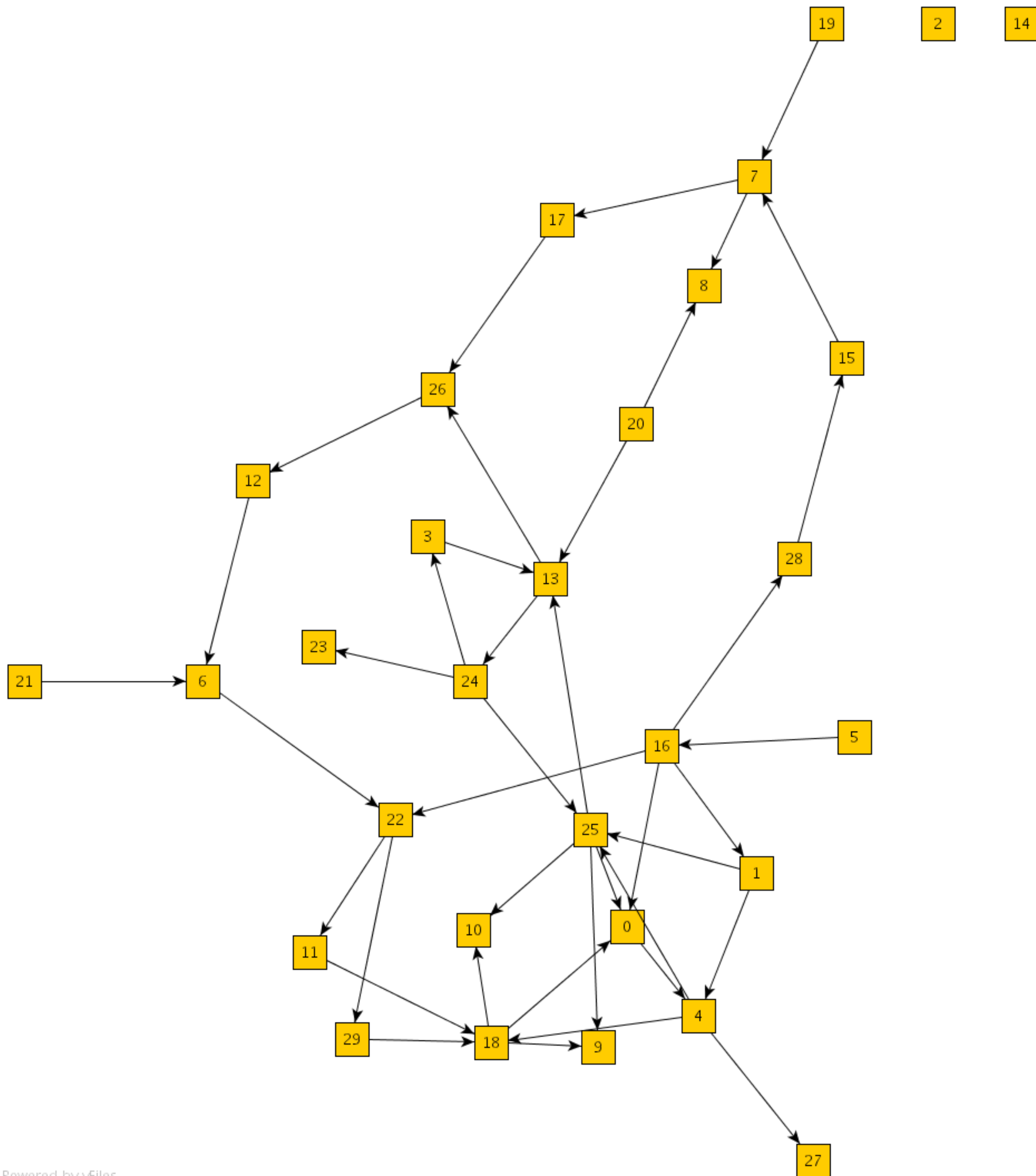
isomorphism When are two graphs to be considered "essentially the same" (ignoring names of vertices and edges)?

On the next three slides, you will see the same directed graph with 30 vertices and 40 edges, created as a "random graph" via the graph editor `yEd`, using different drawing methods.

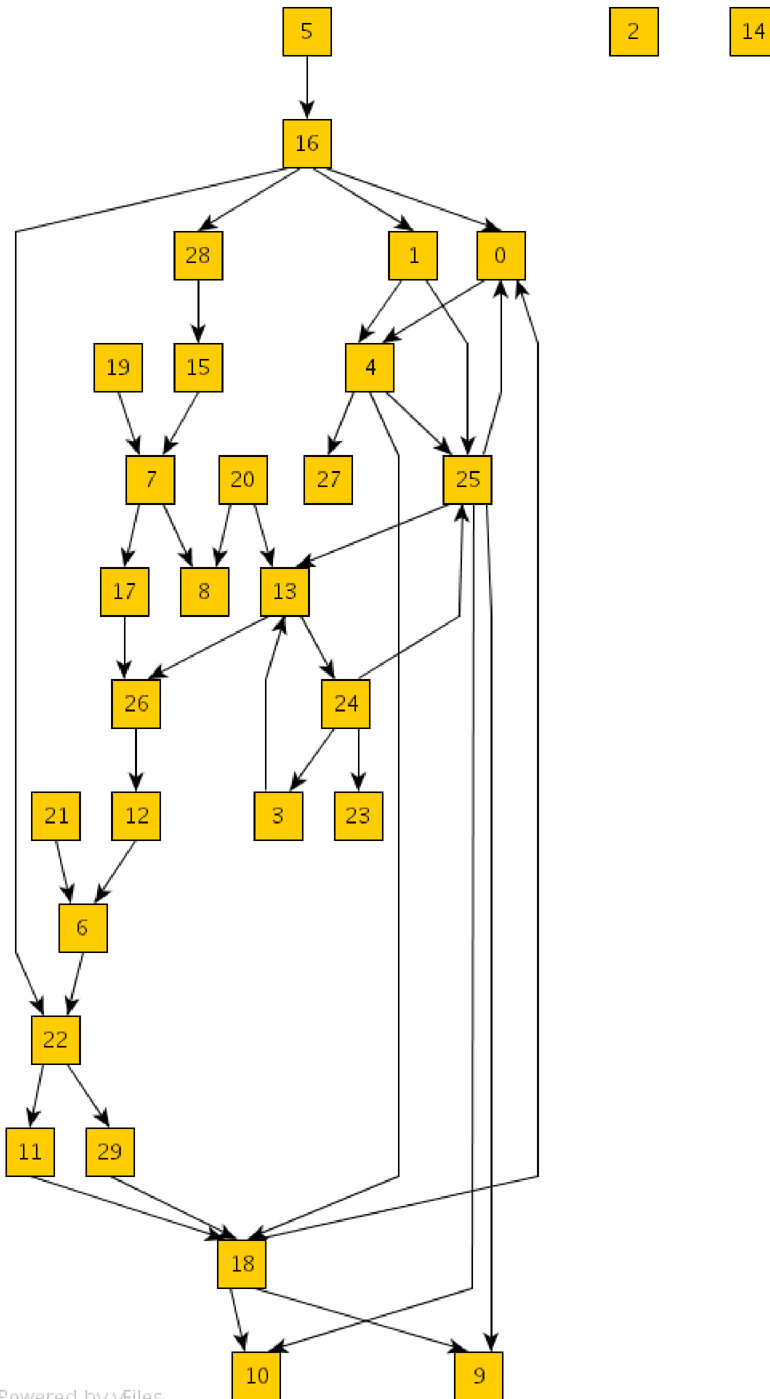
Example: Random layout



Example: Organic layout



Example: hierarchic layout



Zero knowledge proofs and cryptography

Applying some hard problem from graph theory (the graph colouring problem), we consider

Zero Knowledge Proofs,

where you prove something (for example, your identity) without enabling a listener to the conversation to imitate you (steal your identity).

We consider **public key cryptography**, and we will go to the encryption and decryption process of **RSA** in detail.

To do so, we will need some elementary algorithms from number theory and algebra:

1. the Euclidean algorithm
2. fast exponentiation
3. Euler's phi function.

Notions of complexity theory

Instead of algorithms, now we consider the **underlying problems**, and we ask:

Are there better algorithms than those we know?!

We are mainly concerned here with the notions of

efficiently decidable problems which can be "completely solved" in an "efficient manner";

efficiently verifiable problems for which at least, given an alleged solution, we can "efficiently" decide whether it is a correct solution or not.

For solving problems (like in graph theory), we want **efficient algorithms**, while for cryptography **we do not want** certain efficient algorithms to exist!

Algorithms to tackle hard problems

We are considering the "NP-complete" problem of deciding whether a

propositional formula
(build from boolean variables by means of
"and", "or" and "not")

is **satisfiable**.

This is a hard problem. We will see several basic techniques (among others also applying what we learned from graph theory).

We will also discuss algorithms for other hard problems like the graph colouring problem, graph isomorphism problem and scheduling problems.

III Algorithms

Consider a computer program. It uses many *algorithms*.

An algorithm solves a specific *problem*, where a "problem" is a precisely defined *map* from a set of *inputs* to a set of *outputs*.

Thus an algorithm is like an implementation of a Pascal (or C/C++) function

```
function f(m, n : integer) : integer;
```

(here the input consists of two integers, and the output is one integer), where

- we do not consider any side effects (like input or output);
- where our set of basic operations is not restricted to Pascal (or C++) operations, but depends on our setting.

The theory of algorithms

The theory of algorithms gives (some) answers to the following question:

- Given a problem, how can I solve it (i.e., how to find an algorithm)?
- Given a solution to the problem (that is, an algorithm), how much *resources* are needed, where the main resources are
  - time (number of steps)
  - space (size of memory)

  but also other measures are important.
- Given the problem, what are the best solution (i.e., algorithms) for it?!

Concepts and models

When implementing algorithms, we want to be as close as possible to the abstract formulation of the algorithm given by the pseudo-code, while at the same time we need to go into much greater details. **Generic programming** supports implementing algorithms at a high level of generality, without sacrifying efficiency. (Object orientation is a special technique within the field of generic programming, more concerned with certain data types associated for example with graphical user interfaces, but less suited for algorithms.)

The basis of generic programming is given by the notion of a **concept**, which defines in a general way for example for what kind of inputs a graph traversal procedure can be used. A **concept** consists of:

Syntax "how can we use something"

Semantics "what is actually achieved"

Complexity "what are the costs of performing an operation".

As you should know:

Data structures realise **Abstract Data Types (ADT's)**.

Now

"Concepts" enhance Abstract Data Types,

while the realisations of concepts are called **models**.

For example, a concept of a graph must tell us how to access the vertices and edges in a graph, how the different operations play together, and how much time and memory it takes to perform these operations.

A generic graph traversal algorithm for example then takes as input a "generic graph", that is, some objects to which we can apply the general operations given by the concept of a graph, and in this way we can guarantee that our algorithm will work for *any implementation* (model) of the graph concept (without any need to tie your graph into some artificial class hierarchy), and furthermore, based on the complexity guarantee given by the concept, we can also give general complexity guarantees for the graph traversal.

Summary

Notions to keep (and move) in your mind:

- Graph theory:
  - graphs (directed and undirected)
  - general graphs (and simple graphs)
  - vertices and edges
  - parallel edges and loops
  - graph drawings
- Cryptography
  - zero knowledge proofs
  - public key cryptography and RSA
- Complexity theory
  - problems and algorithms
  - efficiently decidable
  - efficiently verifiable
  - P and NP
- Propositional logic; the satisfiability problem (SAT)
- Concepts and models; ADT's.