Advanced Algorithms II: Algorithm Design

David Rydeheard
Room 2.115
david@cs.man.ac.uk

See module website (from syllabus page) for documents and supporting material.

Module Overview

Section 1: Classifying Computational Tasks. Jürgen Dix.

We look at computational tasks to determine:

1. Whether algorithms for the task exist at all: Computability.
2. What range of algorithms do exist, in particular whether efficient algorithms exits in terms of space and time complexity and classify tasks according to what algorithms exist: Complexity classes.

Section 2: Algorithm Design and Analysis. David Rydeheard.

In this section we investigate a wide range of useful algorithms, considering how we develop algorithms, classes of related algorithms and what complexities of algorithms are available.

Organisation

The module is in **TWO** sections. Section (1) is taught by Jürgen Dix; Section (2) by David Rydeheard.

The module has a unique feature: We don't use all the lecture slots to lecture, but **compulsory practical exercise** are set. You **must** attempt these as they extend and re-inforce the material of the lectures, and themselves are examinable under the examination for the module. You will be given the additional time of the lecture slots which are unused in order to complete these practical exercises, and clinic sessions will be held, as follows:

- Section 1: 8 lectures, plus 1 week (two lecture slots) for practical exercises.
- Section 2: 8 lectures, plus 1 week (two lecture slots) for practical exercises.
- 2 clinic sessions.

The website

All students should consult the website for this module. It contains (1) all the lecture notes and slides, (2) relevant code, (3) the practical exercises for the module, (4) details of module organisation, (5) useful links to many other relevant sites.

Advanced Algorithms II: Algorithm Design

Aim: To examine a range of algorithmic problems and solutions

- for their intrinsic interest and applicability,
- to illustrate the development and structure of, often quite intricate, algorithms and attendant data structures,
- to develop arguments about correctness and complexity of algorithms,
- to introduce 'families' of algorithms based upon similar ideas/structures,
- to illustrate Advanced Algorithms I: i.e. show how various real algorithmic tasks fit the complexity classes already studied.

Graph Algorithms

The algorithms we study are based upon tasks associated with **Graphs**. Graph algorithms are of interest because:

- They are of wide applicability, often embedded in system software and in many applications.

  Examples:

  Garbage collectors (eg Java VM), process schedulers, timetabling and travel systems, network management systems, associative memory, symbolic processing, automated reasoning, hierarchical structuring, etc

- They illustrate both a wide range of algorithmic designs and also a wide range of complexity behaviours.

Algorithmic Problem on Graphs

- Connectivity and components,
- Path-finding and traversals, including route finding, graph-searching, exhaustive cycles (Eulerian and Hamiltonian),
- Optimisation problems, eg shortest paths, maximum flows,...
- Embedding problems, eg planarity - embedding in the plane,
- Matching, graph colouring and partitioning,
- Graphs, trees and DAGs, including search trees, spanning trees, condensations etc.

Books

- SEDGEWICK, R. Algorithms in C (ISBN 0-201-51425-7) Addison-Wesley 1990. A general, but good quality, book on algorithms, with some treatment of graph algorithms and computational geometry.
- EVEN, S. Graph Algorithms (ISBN 0-91-489421-8) Computer Science Press 1987. A good treatment of graph algorithms. Out of print - but available in the libraries.
- MCHUGH, J.A. Algorithmic Graph Theory. (ISBN 0-13-019092-6) Prentice-Hall International 1990. The best treatment of graph algorithms. Out of print, I believe.

There are many other treatments of graphs and graph algorithms, the above are perhaps the most comprehensive.

Lectures

- Graphs, Graph as datatypes, Graph searching techniques and generic algorithms [2 lectures],
- Components, Strong connectivity, Articulation points: Efficient algorithms [1 lecture]
- Route-finding and shortest path algorithms [1 lecture],
- NP complete problems on graphs, examples and reductions: Exhaustive paths, cliques, reductions to satisfiability [2 lectures],
- Embedding problems: planarity testing and embedding [1 lecture]
- Matching, colouring and partitioning graphs, overview, 5-colouring and 4-colouring of planar graphs [1 lecture].

Graphs

Terminology: A graph consists of a set of *nodes* and and set of *edges*. Edges link pairs of nodes. For directed graphs each edge has a *source* node and a *target* node. Nodes linked by an edge are said to be *adjacent*.

Alternatives: Points/lines, Vertices/arcs ...

There are a variety of different kinds of graphs:

- Undirected graphs,
- Directed graphs,
- Multi-graphs,
- Labelled graphs (either node-labelled, or edge-labelled, or both).

[Examples]

Representing graphs as datatypes

Graphs may be represented as data types in programming languages in various ways (chosen for faithfulness of the representation, and efficiency):

- Adjacency lists,
- Adjacency matrices,
- Incidence matrices.

Graphs are often represented implicitly in software.

[Examples]

Traversal techniques: Trees and Graphs

On trees

Terminology: Family trees/forest trees! Root, tip, branches, children, descendents, ancestors, siblings.

Idea: Visit every node following the structure of the tree.

There are various methods:

- Depth-first search (DFS): Visit all descendents of a node, before visiting sibling nodes;
- Breadth-first search (BFS): Visit all children of a node, then all grandchildren, etc;
- Priority search: Assign 'priorities' to nodes, visit unvisited child of visited nodes with highest priority.

Priority search: Applications in heuristics, eg in game playing.

[Illustrations]

Graph traversal

A tree is a graph such that:

> There is a distinguished node (the root) such that there is a unique path from the root to any node in the graph.

To modify tree traversal for graphs we:

1. We may revisit nodes: so mark nodes as visited/unvisited and only continue traversal from unvisited nodes.
2. There may not be one node from which all others are reachable: so chose node, perform traversal, then start traversal again from any unvisited nodes.

[Illustrate]

A generic traversal algorithm: for trees

We present a generic search algorithm searching trees from the root.

It uses operations of push, pop, top, empty and

- for a stack, it provides a Depth-First Search,
- for a queue, it provides a Breadth-First Search, and
- for a priority queue, it provides a Priority Search.

This is a generic search routine. It labels each node u with search-num(u) giving the order the nodes are encountered.

```
set u to be the root;
set s to be empty;
set i = 0;
push(u,s);
while s not empty do
 { set i = i+1;
   search-num(top(s)) = i;
   set u = top(s);
   pop(s);
   forall v children of u
     push(v,s) }
```

Recursive depth-first search algorithm for graphs

Allocates a number dfsnum(u) to each node u in a graph, giving the order of encountering nodes in a depth-first search.

```
forall nodes u set dfsnum(u) = 0;
set i = 0;

visit(u) =
   { set i = i+1;
     set dfsnum(u) = i;
     forall nodes v adjacent to u
       { if dfsnum(v) = 0 then visit(v) };

forall nodes u
   {if dfsnum(u) = 0 then visit(u)};
```

Analysis of DFS

A DFS traversal of a directed graph, defines a subgraph which is a collection of trees (ie a forest).

An edge from u to v is a *tree edge* if v is unvisited when we traverse from u to v.

A Depth-first search traversal of a directed graph partitions the edges of the graph into four kinds: An edge from u to v is either a

- *tree edge*,
- *back edge*, if v is an ancestor of u in the traversal tree,
- *forward edge*, if v is a descendent of u in the traversal tree,
- *cross edge* otherwise.

Note: For edge from u to v:

- dfsnum(u) $<$ dfsnum(v) if the edge is a tree edge or a forward edge,
- dfsnum(u) $>$ dfsnum(v) if the edge is a back edge or cross edge.

Could modify DFS routine to identify status of each edge.

For undirected graphs, DFS routine is essentially unchanged but there are only *tree edges* and *back edges*.

[Illustration]

The complexity of DFS

For a graph with N nodes and E edges:

- For the adjacency list representation, the complexity is linear $O(N + E)$,
- For the adjacency matrix representation, the complexity is quadratic $O(N^2)$.

Cycle detection by DFS

As an example of an algorithm using DFS to traverse directed graphs:

Definition: A *path* in a graph is a sequence (possibly empty) of edges $e_1, e_2, \ldots, e_n$ such that for $1 \leq i < n$, $\text{target}(e_i) = \text{source}(e_{i+1})$.

A *cycle* is a non-empty path beginning and ending at the same node. A graph is *acyclic* when it has no cycles.

Cycle detection using DFS is based on the following result:

Proposition A graph is acyclic just when in any DFS there are no back edges.

Proof

1. If there is a back edge then there is a cycle.
2. Conversely, suppose there is a cycle and the first node of the cycle visited in the DFS (ie minimum dfsnum) is node u. Now the preceding node in the cycle v is reachable from u via the cycle so is a descendent in the DFS tree. Thus the edge from v to u is a back edge.

   So a cycle in the graph implies the existence of a back edge in any DFS, as required.

We can use this to construct a linear cycle detection algorithm: Simply perform a depth-first search, and a cycle exists if and only if a back edge is encountered.

Connected components, Strongly connected components

For directed graphs:

Definition Two nodes u and v in a graph are *linked* if there is an edge from u to v OR from v to u.

Two nodes u and v are *connected* if there is a, possibly empty, sequence of nodes $u = u_0, u_1, \ldots, u_n = v$ with u_i linked to u_{i+1} for all i, $0 \geq i < n$.

[Illustration. General idea: Transitive Closure]

Definition A *connected component* of a graph is a maximal set of connected nodes, ie A set of nodes C is a connected component just when:

1. Every pair of nodes in C is connected, and
2. There is no node outside C connected to any node in C.

[Illustration: This is a natural partitioning of the nodes of a graph.]

Definition Two nodes u and v of a graph are *strongly connected* if there is a path (possibly empty) from u to v AND a path from v to u.

A *strongly connected component* is a maximal set of nodes each pair of which is strongly connected.

[Illustration]

Articulation points

For *undirected* graphs:

Definition An *articulation point* of a graph is a point whose removal increases the number of components.

[Illustrate]

Application: Articulation points in a network are those which are critical to communication: for an articulation point, all paths between certain nodes have to pass through this point.

Articulation points divide a graph into *blocks*. Within each block there are multiple non-intersecting paths between all pairs of nodes, and blocks are maximal with this property.

Linear Depth First Search Algorithms

There are algorithms, based on DFS, for calculating components, strongly connected components, articulation points, blocks and similar graph structures, which are linear.

This may be considered surprising! We illustrate with articulation points.

Reference: Tarjan, R.E. Depth first search and linear graph algorithms. SIAM Journal of Computing, 1. pp 146–160.

An efficient algorithm for articulation points

Proposition

In a DFS tree of an undirected graph, a node u is NOT an articulation point just when, for every child v of u, there is a descendent in the DFS tree from v which is connected by a back edge to a node higher in the tree than u.

(Root is not an articulation point just when it has one or zero children.)

To calculate articulation points we compute for each node u a number lowpoint(u).

Lowpoint(u) is the least number dfsnum(v) in a DFS traversal of a graph such that:

> node v can be can be reached by a, possibly empty, path of tree edges followed by at most one back edge.

Proposition

A node u is an articulation point just when in a DFS of the graph there is a child node v such that lowpoint(v) $\geq$ dfsnum(u).

To print all articulation points in a graph: Modify the recursive DFS algorithm to return lowpoint(u) starting at u. Print nodes u for which there is a child v with lowpoint(v) $\geq$ dfsnum(u):

```
set i = 0;
visit(u) =
  { set i = i+1;
    set dfsnum(u) = i;
    set min = i;
    forall nodes v adjacent to u
      { if dfsnum(v) = 0 then
          { m = visit(v);
            if (m < min) then set min = m;
            if (m >= dfsnum(u)) then print(u) }
          else if (dfsnum(v) < min)
                    then set min = dfsnum(v) };
     return min }
```

This is a *linear* algorithm.

Similar techniques apply to calculating strong components etc in linear time.

Path-finding algorithms

There are numerous path-finding problems in graphs and a variety of algorithms.

- To find all paths between all pairs of nodes ('transitive closure'), or
- To find all paths between a fixed pair of nodes, or
- To find all paths from one node to all others (the 'single source problem').

When the edges are labelled with numerical values, then we can ask for *shortest paths*, by which we mean a path of minimum length, where the length of a path is the sum of its edge labels. Each problem above yields a shortest path problem.

Optimization problems.

In fact, the second and third problems are equivalent.

Optimization and shortest paths

There is an 'optimization property' concerning shortest paths:

> If p is a shortest path from node u to node v via node w, then the portions of p from u to w and from w to v are both shortest paths.

Proof: Evident!

Such optimization properties mean that there are direct methods of computing shortest paths: To accumulate shortest paths we need combine only other shortest paths (and not consider any other paths).

A shortest paths algorithm

As an example, we give Floyd's algorithm for computing shortest paths (we simplify and calculate only shortest distance) between ALL PAIRS of nodes.

Let d(i,j) be the label of the edge from i to j, if there is such an edge, otherwise ∞.

We calculate successively (in-place) an array D(i,j) which at the k-th step is the shortest distance from i to j, using only intermediate nodes 1,... k. We set D(i,j) to be d(i,j) initially and then compute:

```
for k from 1 to N
 for i from 1 to N
   for j from 1 to N
     set D(i,j) = min(D(i,j), D(i,k)+D(k,j))
```

Correctness: This is the optimization property, but requires that if there are negative labels then no cycle is of negative length.

Complexity of Floyd's algorithm: $O(N^3)$, N the number of nodes.

Other problems:

For the single source shortest-path problem (and positive numbers as labels) the standard algorithm is Dijkstra's algorithm, which is $O(N^2)$, but with more efficient data structures (heaps) can be implemented as an $O(E \times \log(N))$ algorithm which for E much less that N^2 is an improvement.

Complexity classes and computational tasks

Most of the graph algorithms we have met so far are polynomial-time algorithms i.e. in the class **P** as defined in the first section of this module.

Are there any computational problems on graphs which are in other complexity classes?

To answer this we must first link the definitions of the classes in terms of words in a language (as given in the first section of this module) with decision problems and computations over graphs.

Let X be **a decision problem for graphs** i.e. a problem of the form: For any graph g, does g satisfy property P?

Examples: (1) Is g connected? (2) Does g have k articulation points, or less than k articulation points, etc.

We can always convert decision problems for graphs into the problem of determining whether a word w is in a language L, as follows:

Represent graphs as words either using standard computational representations of graphs (as described earlier) or, more directly, e.g. $(a, b, c; x : a \to b, y : b \to c, z : c \to a)$ represents the three-cycle as a word.

A decision problem for the graph property P can then be expressed as a decision problem for languages as follows:

Let L be the language of words representing the graphs that satisfy property P. Then for graph g represented by word w, $w \in L$ just when g satisfies P.

In this way, the complexity classes described in the first section of this module, apply equally well to problems for graphs (and indeed for any other data structures that can be encoded as finite words).

NP-completness

Reminder: A computational problem X is NP-complete just when:

1. X is in NP, i.e. can be solved with a non-deterministic polynomial-time algorithm, and
2. X is *complete* in this class, i.e. for all Y in NP, Y *reduces* to X in polynomial-time.

The notion of *reduction* is that Y reduces to X in polynomial-time, if each instance of Y can be translated in polynomial-time to an instance of X such that the Y-instance has a solution (i.e. returns true) if and only if its translation does.

(Sometimes polynomial-time is replaced here by smaller classes, eg log-space, but these tend to lead to the same subset of NP.)

Recall: NP-complete problems are believed to be intrinsically hard, that is, the only algorithms available are exponential-time

((P=NP?).

How to show a computational task is NP-complete

Two methods: (1) Directly, using the definition of NP-completeness; or, (2) via translation using a known NP-complete problem.

Direct method: This involves first showing that the problem is in NP, and then that *all NP problems* reduce to this problem in polynomial time.

The later task is very difficult because we have to translate every possible NP problem into the one we are investigating. In general, we *do not* use this method.

Indirect method: If we are given a computational task that is already established as NP-complete, then to show another task is NP-complete, we can use the following method.

NP-completeness via translation

Proposition

If X is a computational task that is NP-complete, then a task Y in NP is NP-complete exactly when X is polynomial-time reducible to Y.

Proof:

1. If Y is in NP-complete then, because X is in NP, then X is polynomial-time reducible to Y.
2. Conversely, suppose that X is NP-complete, and X is polynomial-time reducible to task Y in NP. To show that Y is NP-complete, we need to show that for any Z in NP, Z reduces in polynomial-time to Y. But as X is NP-complete, Z reduces in polynomial-time to X, which itself reduces in polynomial-time to Y. Composing these two reductions gives a polynomial-time reduction of Z to Y as required.

So, all we need to do establish that problem Y is NP-complete is

1. Show the problem is in NP. This is usually easy: just show that 'checking a solution' takes polynomial-time; and then
2. Define a polynomial-time translation from a given NP-complete problem into Y. This usually requires considerable ingenuity.

A known NP-complete problem

For the above scheme to work, we need a given NP-complete problem. Do we have one? Yes:

Cook's Theorem says:

> The problem of determining whether logical expressions can be satisfied (ie the logical variables can be replaced by truthvalues to make the expression true) is NP-complete.

Proving this is *not easy*. It involves coding every NP-problem as an instance of satisfiability, and this involves coding non-deterministic Turing machines in terms of logical expressions, including their tapes, their codes, and their moves, in such a way that every NP-problem is reduced (in poly-time) to satisfiability.

NP-complete graph problems

Many simple and common problems on graphs are NP-complete. We prove one such problem is NP-complete by polynomial-time reducing *satisfiability* of boolean expressions to this problem. As satisfiability is NP-complete so must be this problem.

Definition: For undirected graphs, a *clique* is a subgraph such that any pair of nodes in the subgraph is connected by an edge (otherwise called *complete subgraphs*).

The problem is to determine for any graph whether or not it has a clique with k nodes.

Cliques and satisfiability

Proposition

The problem of determining whether or not a graph has a clique with k nodes is NP-complete.

Proof: We show that any satisfiability problem converts to a graph, and that existence of cliques corresponds to a substitution which satisfies the boolean expression.

Thus let a e be a boolean expression in the form

$$(x_1 \vee x_2 \vee \ldots) \wedge (y_1 \vee y_2 \vee \ldots) \wedge (z_1 \vee \ldots) \ldots$$

where each entry is a literal, ie a variable or the negation of a variable.

Now form a graph: Nodes are pairs (x, i) for each literal x in the i-th clause. Nodes (x, i) and (y, j) are connected by an edge just when x is not the negation of y and $i \neq j$.

Then an expression of k clauses is satisfiable just when this graph has a clique of k nodes. A clique of k nodes must select a literal from each clause, and no variable and its negation can be in the clique. Likewise if the expression is satisfiable, then we must be able to find a literal in each clause which we can make simultaneously true, ie no variable and its negation is allowed.

[Example: Try $(x \vee y) \wedge (\bar{x} \vee z) \wedge (x)$.]

Many other graph problems are NP-complete, as we shall see, but it is not always obvious whether a problem is NP-complete, and closely allied problems can differ widely in their complexity.

Eulerian Paths: Euler 1780s

An *Eulerian circuit* of an undirected graph is a cycle that passes along each edge just once.

Proposition (for undirected graphs)

A graph has an Eulerian circuit just when it is connected and every node has even degree.

(For directed graphs the corresponding property is: For each node n, outdegree(n) = indegree(n).)

[Proof and (linear) algorithm]

Hamiltonian Paths: Hamilton 1850s

A *Hamiltonian circuit* of an undirected graph is a cycle that passes through *each node* just once.

Related to the celebrated *travelling salesperson problem*: In a graph with edges labelled with distances, to determine (if it exists) a minimum length Hamiltonian circuit.

No simple criterion: Deciding whether a graph has a Hamiltonian circuit is NP-complete, as is the travelling salesperson problem. Algorithms, in general, are by *exhaustive unlimited back-tracking*.

Planarity

Planarity is about depicting graphs in 2-D space - how do we 'draw' graphs and can we do so without any 'overlapping'. It has applications (of a sort) in planar networks, eg in embedding circuits on a chip, or whenever overlapping is difficult to accommodate, or in depicting graphs in a simple form in 2-D (eg in graphics).

Definition: An *embedding* of a (directed or undirected, usually the latter) graph in the plane is an allocation of distinct points to the nodes and distinct continuous lines (not necessarily straight - that is another problem) to the edges, so that no two lines intersect.

There may be more than one 'way' of embedding a graph in the plane. But can all graphs be embedded in the plane?

A graph that can be embedded in the plane is called a *planar* graph.

Planarity algorithms

The graphs C_5 (the complete undirected graph on 5 nodes, ie all edges present), and $C_{3,3}$ (the complete bipartite graph - with three nodes in each set, and all edges between nodes in different sets) are both *non-planar*.

[Illustrate]

In a sense these are the only non-planar graphs. This gives an unusual criterion for recognising planar graphs, based on the following.

Proposition (Kuratowski 1930)

A graph is planar just when it does not contain any subgraph homeomorphic to C_5 or $C_{3,3}$.

A graph is *homeomorphic* to another if they differ only in the addition and deletion of nodes of degree two (ie nodes 'along edges').

[Illustrate]

Proof omitted: involved but not difficult!

This gives a means of determining whether or not a graph is planar - exhaustively search for subgraphs homeomorphic to these two graphs. This is not efficient, nor does it yield a planar embedding if one exists. The proof when examined may do - but remains inefficient.

There are, however, efficient planarity testing algorithms which also yield a planar embedding if it exists.

- Demoucron's algorithm (1964) works by successively building a planar embedding, 'flipping over' parts of graphs where necessary,
- Hopcroft and Tarjan introduced a linear-time planarity algorithm (1974).

Graph Colouring (for undirected graphs)

A *colouring* of a graph with k colours is an allocation of the colours to the nodes of the graph, such that each node has just one colour and nodes linked by an edge have different colours.

The minimum number of colours required to colour a graph is its *chromatic number*.

[Examples]

Applications: Graphs are usually used where 'connectedness' is being recorded. Colouring is about the opposite: edges are present when nodes need to be separated. The k colours partition the nodes into sets of nodes that can co-exist. Examples: Scheduling under constraints - nodes: tasks, edges - constraints on simultaneous running.

Colouring algorithms

To determine whether a graph can be coloured with k ($k \geq 3$) colours is an NP-complete problem.

The only algorithms that exist in general for colourability are *exhaustive unlimited back-tracking* algorithms, as below.

```
set n = 1;
while n =< N do
  { attempt to colour node n with
                next colour not tried for n:
     if there is no such colour
      then if n>1
          then set n = n-1 else fail
      else if n=N then print colouring else set n = n+1 }
```

The 4-colouring of planar graphs...

... is a celebrated problem.

Proposition

Every planar graph can be coloured with just 4 colours.

- Discussed as a possibility in the 1850s,
- 1879: Kempe publishes a 'proof' of the 4-colour theorem,
- 1890: Heawood finds error in Kempe's proof, but shows his proof establishes the 5-colourability of planar graphs,
- Since then finding a proof of 4-colourability has been a catalyst for much combinatorial mathematics,
- 1977: Computer-aided proof of 4-colourability: reduced the graphs required to be coloured to a finite number (over 1000) and then used a computer to generate colourings.

[Give: Proof and algorithm for 5-colouring.]