Graphs and Graph Algorithms

Graphs and graph algorithms are of interest because:

- Graphs model a wide variety of phenomena, either directly or via construction, and also are embedded in system software and in many applications.
- Graph algorithms illustrate both a wide range of algorithmic designs and also a wide range of complexity behaviours, from linear and polynomial-time, to exponential-time algorithms.

The aim is to:

- explore a little of the wide range of graph algorithms,
- introduce some of the relevant techniques, and
- consider what algorithms and performances are available, and the distinction between tractable and intractable problems.

Algorithmic Problems on Graphs

There is a wide range of computational tasks 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, travelling salesperson...
- Embedding problems, eg planarity - embedding in the plane,
- Matching ('marriage problems'), graph colouring and partitioning,
- Graphs, trees and DAGs, including search trees, traversals, spanning trees, condensations, etc.

Books

Several recommended books:

- SEDGEWICK, R. Algorithms (in C, and in C++ and in Java): Graph Algorithms, Addison-Wesley. Several books, standard texts and of good quality.
- JUNGNICKEL, D. Graphs, Networks and Algorithms, Springer 2008. A nicely presented and fairly comprehensive text.
- 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.

Basic concepts and terminology

Terminology:
A graph consists of a set of nodes 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 (not 'connected'- this is used for a more general concept).

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

There is a variety of different kinds of graphs:

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

Basic concepts and terminology - continued

In an undirected graph, the degree of a node is the number of edges incident at it. For a directed graph, each node has an in-degree and an out-degree.

A path in a directed graph is a (possibly empty) sequence of edges, $e_1, \ldots, e_n$ such that for all i $(1 \leq i \leq n-1)$,

$$\mathit{target}(e_i) = \mathit{source}(e_{i+1}).$$

A node n is reachable from m if there is a path from m to n.

For undirected graphs, paths are similar but with the requirement that adjacent edges in the sequence are incident at the same node.

A cycle is a path starting and finishing at the same node. A loop is a 1-edge cycle. A graph is acyclic when it has no cycles.

Representing graphs as datatypes

How are graphs represented in programming languages?
(Reminder!)

Several standard techniques:

- **Adjacency lists**,
- **Adjacency matrices**.

There are others, such as incidence matrices, and graphs also often appear implicitly in programs.

Which representation to choose depends on

- properties of the graph (e.g. 'sparseness'),
- the algorithms we wish to implement,
- the programming language and how data structures are implemented, and
- application of the code.

Representing graphs: Adjacency lists

An adjacency list representation of a graph consists of

- a list of all nodes, and
- for each node n, a list of all adjacent nodes (for directed graphs, these are the nodes that are the target of edges with source n).

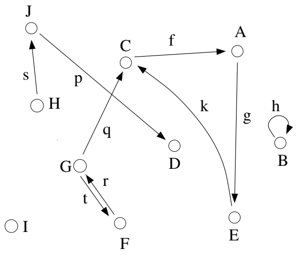

Figure: A directed graph.

Node	Adjacent nodes
A	E
B	B
C	A
D	
E	C
F	G
G	F, C
H	J
I	
J	D

Figure: Its adjacency list.

Representing graphs: Adjacency matrices

An adjacency matrix representation of a graph consists of a 2-dimensional array (or matrix), each dimension indexed by the nodes of the graph. The entries in the matrix are:

- 1 at index (m, n) if there is an edge from m to n,
- 0 at index (m, n) if there is no edge from m to n.

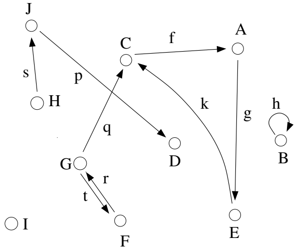

Figure: A directed graph.

	A	B	C	D	E	F	G	H	I	J
A	0	0	0	0	1	0	0	0	0	0
B	0	1	0	0	0	0	0	0	0	0
C	1	0	0	0	0	0	0	0	0	0
D	0	0	0	0	0	0	0	0	0	0
E	0	0	1	0	0	0	0	0	0	0
F	0	0	0	0	0	0	1	0	0	0
G	0	0	1	0	0	1	0	0	0	0
H	0	0	0	0	0	0	0	0	0	1
I	0	0	0	0	0	0	0	0	0	0
J	0	0	0	1	0	0	0	0	0	0

Figure: Its adjacency matrix.

Representing graphs: Notes

- These are both tabular representations. The exact choice of types to represent them depends on the application. For example, an adjacency list may be an array of linked lists, if we wish to have fast (random) access to the lists of adjacent nodes, but to iterate through these lists.
- Notice how sparse the adjacency matrix is: most entries are zero as there are few edges in the graph.
- Adjacency matrices allow us to do arithmetic! We may add and multiply matrices, take determinants etc.
- The representations need modification to include various data. For example, for a multigraph, we may record the edges as well as the adjacent nodes in a list representation, or the number of edges in a matrix representation. Also for edge-weighted graphs, we include the weights in the representation.
- Notice that for undirected graphs, the matrix is symmetrical.

Traversal techniques: Trees

Tree terminology:
Family trees and forest trees... Root, tip, branches, children, descendants, ancestors, siblings, etc.

Traversal idea: Visit every node following the structure of the tree.

There are various methods (these are rough descriptions - the precise descriptions are the algorithms themselves):

- Depth-first search (DFS): Visit all descendants 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, e.g. in game playing.

A generic traversal for trees

There is a generic traversal algorithm which, with the same code, traverses a tree from the the root by the various traversal methods.

It uses a datatype with operations of push, pop, top, empty:

- 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.

A generic search algorithm for trees

This traversal routine for trees labels each node u with search-num(u) giving the order the nodes are encountered. For different implementations of push and pop we get different traversals.

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

Tranversal techniques: Graphs

A tree is a directed 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:

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 choose node, perform traversal, then restart traversal from any unvisited nodes.

Recursive depth-first search algorithm for graphs

DFS and recursion and stacks are closely related:

This recursive DFS 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 { dfsnum(u) := 0};
i := 0;

visit(u) =
   { i := i+1;
     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 (i.e. a forest).

An edge from u to v is a tree edge if v is unvisited when we traverse from u to v. These form a collection of trees – a forest.

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 exactly one of the following:

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

Analysis of DFS - continued

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. How?

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

Complexity of DFS

For a graph with N nodes and E edges:

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

A DFS algorithm: Cycle detection

We may use DFS to check for cycles in a directed graph:

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 (i.e. minimum dfsnum) is node u. Now the preceding node in the cycle v is reachable from u via the cycle so is a descendant 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 detected.

Connected components

For directed graphs: 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, $0 \geq i < n$.

Definition: A (connected) component of a graph is a maximal set of connected nodes, i.e. 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.

This is a natural partitioning of the nodes of a graph.

Strongly connected components

Another notion of ‘component’ in a graph is:

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.

Applications?

Articulation points

For undirected graphs:

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

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 subgraphs called 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 linear-time algorithms, based on DFS, for calculating components, strongly connected components, articulation points and blocks.

This may be considered surprising!

Reference: Tarjan, R.E. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1. 1972. pp 146–160.

In fact, DFS techniques yield many efficient (often linear-time) algorithms across the range of tasks on graphs.

Survey of some computation tasks on graphs: 1

We now consider a range of computational tasks on graphs and the algorithms available.

- Path finding: There are various path-finding problems in graphs.
    - All paths between all pairs of nodes ('transitive closure'), or
    - All paths between a fixed pair of nodes, or
    - All paths from one node to all others (the 'single source problem').

  The second and third problems are equivalent.
  When the edges are labelled with numerical values, then we can ask for shortest paths - paths with a minimum sum of edge labels.
  Algorithms: Floyd's algorithm finds shortest paths between all pairs of nodes in $O(N^3)$.
  Dijkstra's algorithm solves the shortest path single source problem in $O(E \times \log(N))$ (with an efficient implementation).

Survey of some computation tasks on graphs: 2

- Planarity:
  An embedding of a 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.
  Some graphs can be embedded, others not. Hopcroft and Tarjan introduced the first linear-time planarity algorithm (1974) based on Depth-First Search.
- Subtrees in graphs: A spanning tree of an undirected graph is a subgraph which (a) is a tree (i.e. no cycles), and (b) includes all nodes. For integer-labelled graphs, a minimum spanning tree has minimum label-sum.
  Finding minimum spanning trees is an optimisation problem for which there are greedy solutions with worst-case time complexity $O(E \times \log(N))$.

Survey of some computation tasks on graphs: 3

- Matching problems ('Dating sites'): A matching in an undirected graph is a collection of edges which share no nodes. A spanning (or maximum) matching includes all nodes of the graph.
  There is a wide variety of algorithms for finding maximum matchings (based on matrices and linear algebra, flow algorithms, probabilistic methods, integer programming, etc.) with worst-case time complexities as good as $O(N^{2.5})$.
- Flow problems: Consider a directed graph with edges labelled with non-negative integers. Select two nodes (the source and sink). Interpret the edges as pipes and the labels as their capacity. Then the problem is to determine a maximum flow from the source to the sink through the graph.
  Celebrated problem with a well-developed theory: Best algorithms have $O(N^2 \times \sqrt{E})$ worst-case time complexity.

Exponential algorithms on graphs

For many familiar and commonly occurring computational problems on graphs, the only algorithms that are available are exponential-time in the worst case.

Many of these belong to complexity classes called NP-complete and NP-hard (see later in the course) and are called computationally intractable problems.

Whether NP-complete problems admit polynomial-time algorithms is one of the most important open problems in Computer Science. One million dollars to anyone who solves it!

At present only exponential-time algorithms are available. However, there are approaches to these problems which provide more efficient computation (see later in the course).

Why are exponential algorithms intractable? (Reminder)

Exponential behaviour includes: 2^N, $N!$ (the factorial of N) and N^N.

These run very slowly as the size of the input N increases and on small inputs require unfeasible computational resources:

N	2^N	$N!$	N^N
10	10^3	4×10^6	10^{10}
20	10^6	2×10^{18}	10^{26}
100	10^{30}	9×10^{157}	10^{200}
$1,000$	10^{301}	10^{2672}	10^{3000}
$10,000$	10^{3000}	10^{36701}	10^{40000}

Why are exponential algorithms intractable? (Reminder)

Exponential behaviour includes: 2^N, $N!$ (the factorial of N) and N^N.

These run very slowly as the size of the input N increases and on small inputs require unfeasible computational resources:

N	2^N	$N!$	N^N
10	10^3	4×10^6	10^{10}
20	10^6	2×10^{18}	10^{26}
100	10^{30}	9×10^{157}	10^{200}
$1,000$	10^{301}	10^{2672}	10^{3000}
$10,000$	10^{3000}	10^{36701}	10^{40000}

The age of the universe is approximately 4×10^{26} nanoseconds.

It is thought that there are approximately 10^{80} electrons in the universe (the Eddington number).

Complexity classes and computational tasks

Finding how difficult a computational task is, is difficult!

Problems which appear very similar, or even variants of each other, can have very widely different complexities.

An example: Eulerian circuits and Hamiltonian circuits

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

Proposition: (Leonhard Euler, 1735): A graph has an Eulerian circuit just when it is connected and every node has even degree.

The proof of this gives a linear-time algorithm for testing for, and constructing, an Eulerian circuit.

Complexity classes and computational tasks (continued)

A variant of Eulerian circuits, is Hamiltonian circuits (William Hamilton, 1850s):

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

No simple criterion exists: Deciding whether a graph has a Hamiltonian circuit is NP-complete. The only known algorithms are exponential-time.

Exponential algorithms - techniques

Exponential algorithms arise from various algorithmic techniques:

- Exhaustive enumeration - listing all possibilities and checking each for the solution requirement,
- Systematic search of the space of (partial) solutions using unbounded backtracking and navigation methods,
- Dynamic programming techniques, especially for optimisation problems.

Graph Colouring

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.

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.

For $k \geq 3$, this is an NP-complete problem and the only algorithms available are exponential in the worst case.

An exponential-time colouring algorithm - by enumeration

Exhaustive enumeration is the simplest approach to colouring a graph with k colours:

1. Enumerate all allocations of k colours to the nodes of the graph,
2. Check each allocation to see if it is a valid colouring.

For a graph with N nodes and E edges, there are k^N allocations of k colours to the nodes.

Checking each allocation to see if it is a valid colouring, is $O(E)$.

So this algorithm has time complexity $O(E \times k^N)$.

A graph colouring algorithm by systematic search

Here is an algorithm for k-colourability using exhaustive unlimited back-tracking.

Number the nodes $1, \ldots, N$. For each node encountered, it successively tries the colours. If all fail, it undoes the colour of the previous node and tries new colours for it. If it succeeds, it then tries to colour the next node.

```
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 n := n-1 else fail
    else if n=N then print colouring else n := n+1
```

Time complexity: This is exponential in worst-case. Why?

The travelling salesperson problem

The travelling salesperson problem (TSP) is one of the most celebrated of the NP-hard problems and also is widely applicable.

The problem: Find a tour of N cities in a country (assuming all cities to be visited are reachable). The tour should (a) visit every city just once, (b) return to the starting point and (c) be of minimum distance.

This is an optimisation problem - not just find a solution but find ‘the best’ solution.

Here the relevant undirected graph is dense - there is an edge between every pair of nodes.

(TSP is an extension of the Hamiltonian circuit problem.)

Travelling salesperson: Exhaustive enumeration solution

Exhaustive enumeration:

1. Starting at any city, enumerate all possible permutations of cities to visit,
2. Find the distance of each permutation and choose one of minimum distance.

There are $(N-1)!$ permutations for N cities (the starting city is arbitrary).

Stirling's formula:

$$N! \approx \sqrt{2\pi N}(N/e)^N$$

where $e = 2.71828\ldots$ the base of natural logarithms (Euler's number) Thus this is not only exponential, but badly so!

A dynamic programming solution

Faster than the exhaustive enumeration (but still exponential) is a dynamic programming approach. References:

(1) 'A dynamic programming approach to sequencing problems', Michael Held and Richard M. Karp, *Journal for the Society for Industrial and Applied Mathematics* 1:10. 1962.

and

(2) 'Dynamic programming treatment of the travelling salesman problem', Richard Bellman, *Journal of Assoc. Computing Mach.* 9. 1962.

There is an optimisation property for TSP:

> *Every subpath of a path of minimum distance is itself of minimum distance.*

Let us number the cities $1, 2, \ldots, N$ and assume we start at city 1, and the distance between city i and city j is $d_{i,j}$.

The travelling salesperson: dynamic programming

The Held-Karp-Bellman algorithm:

Consider subsets $S \subseteq \{2, \ldots, N\}$ of cities and, for $c \in S$, let $D(S, c)$ be the minimum distance, starting at city 1, visiting all cities in S and finishing at city c.

We have, if $S = \{c\}$, then $D(S, c) = d_{1,c}$. Otherwise:

$$D(S, c) = \min_{x \in S-c}(D(S - c, x) + d_{x,c})$$

Why?

Then the minimum distance for a complete tour of all cities is

$$M = min_{c \in \{2,\ldots,N\}}(D(\{2, \ldots, N\}, c) + d_{c,1})$$

A tour $\langle n_1, \ldots n_N \rangle$ is of minimum distance just when it satisfies

$$M = D(\{2, \ldots, N\}, n_N) + d_{n_N,1}.$$

The travelling salesperson: dynamic programming

A dynamic programming algorithm: Compute the solutions of all subproblems starting with the smallest. Whenever computing a solution requires solutions for smaller problems using the above recursive equations, look up these solutions which are already computed.
To compute a minimum distance tour, use the final equation to generate the lst node, and repeat for the other nodes.

For this problem, we cannot know which subproblems we need to solve, so we solve them all.

The worst-case time complexity of this algorithm is $O(N^2 \times 2^N)$.

(Compare this formulation with Knapsack problems and their dynamic programming solutions.)

To come...

Continuing the course...

Part 2. Complexity

Complexity measures and complexity classes - defining polynomial classes, NP classes and completeness. The $P = NP$ problem. The widespread occurrence of NP-complete and NP-hard problems.

Part 3. Tackling intractability

Approaches to efficient solutions of NP-complete and NP-hard problems: Approximation and non-deterministic methods.