Graphs and Algorithms
Lecture Notes for COMP40008
2023-2024

Iain Phillips

Contents

1 Graphs and Graph Algorithms **5**

1.1 Graphs—Basics 5

1.1.1 Definitions 5

1.1.2 Representations of Graphs 6

1.1.3 Big-Oh Notation 8

1.2 Graph Isomorphism and Planar Graphs 8

1.2.1 Isomorphism 8

1.2.2 Planar Graphs 10

1.2.3 Colouring Maps and Graphs 12

1.3 Connectedness, Euler Paths and Hamiltonian Circuits 13

1.3.1 Paths and Cycles 13

1.3.2 Euler Paths and Circuits 14

1.3.3 Hamiltonian Circuits 16

1.4 Trees 18

1.5 Directed Graphs 19

1.6 Graph Traversal 20

1.6.1 Depth-first and Breadth-First Search 20

1.6.2 Applications 22

1.7 Weighted Graphs 28

1.7.1 Minimum Spanning Trees 28

1.7.2 Prim's Algorithm 29

1.7.3 Kruskal's Algorithm 34

1.7.4 The Shortest Path Problem 40
1.7.5 The Travelling Salesman Problem 53

2 Algorithm Analysis **56**
2.1 Introduction 56
2.2 Searching an unordered list 57
2.3 Searching an ordered list 59
2.4 Orders of complexity 62
2.5 Strassen's Algorithm 64
2.6 Sorting 65
2.6.1 Lower Bound for Sorting by Comparison 66
2.6.2 Mergesort 69
2.6.3 Master Theorem 71
2.6.4 Quicksort 75
2.6.5 Heapsort 78
2.7 Dynamic Programming 85

3 Introduction to Complexity **88**
3.1 Tractable problems and P 88
3.2 The complexity class NP 91
3.3 Problem reduction 93
3.4 NP-completeness 94
3.4.1 Intractability via NP-completeness 95

Introduction

This course falls into three parts. In Part 1 we study graphs and graph algorithms. In Part 2 we analyse algorithms for searching and sorting. Some of the algorithms discussed in Part 2 have already been introduced and verified in 141 Reasoning about Programs. In Part 3 we give an introduction to Complexity Theory, and discuss NP-complete problems and the $\mathsf{P} = \mathsf{NP}$ question.

Acknowledgements. Thanks to Istvan Maros for pointing out some errors in an earlier version. I have benefited from consulting Ian Hodkinson's lecture notes on algorithms.

Reading

Main recommendations:

- T. Roughgarden, Algorithms Illuminated: Omnibus Edition. Cambridge 2022. Chapters 1-5, 7-10, 13, 15, 16, 18, 21.1. Note that Chapters 19-24 use a different definition of the complexity class NP, and a different notion of reduction, from the definitions we adopt in these lectures.
- S. Baase & A. Van Gelder, Computer Algorithms: Introduction to Design and Analysis. Third Edition. Addison-Wesley 2000. Chapters 1, 3.6-3.7, 4.1-4.9, 7-10, 13.
- Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C. (2022) Introduction to algorithms. 4th ed. Cambridge, Mass. ; London, MIT press. Chapters 1-3, 4.1-4.5, 6, 7, 8.1, 14.1-14.3, 19.1-19.3, 20.1-20.4, 21, 22.3, 23.2, 34.
- J. Gersting, Mathematical Structures for Computer Science. Seventh edition. Freeman, 2014. Chapters 2, 5, 6.

Also:

- S. Dasgupta, C.H. Papadimitriou, U. Vazirani, Algorithms. McGraw-Hill, 2008.
- Sedgewick, R., Wayne, K.D. (2011) Algorithms. 4th ed. Boston, Mass. ; London, Addison-Wesley.
- Cormen, T.H. (2013) Algorithms Unlocked. MIT Press.
- L. Fortnow, The Golden Ticket: P, NP, and the Search for the Impossible. Princeton University Press, 2013.

Synopsis

Part 1: Graphs, graph isomorphism, planar graphs, Euler paths, Hamiltonian circuits, trees, depth-first search, breadth-first search, weighted graphs, minimum spanning trees, Prim's algorithm, Kruskal's algorithm, Dijkstra's shortest path algorithm, travelling salesman problem, Floyd-Warshall algorithm.

Part 2: Time complexity, worst-case and average case analysis, decision trees, optimality, orders of complexity (big-Oh and Θ), Strassen's algorithm, master theorem for recurrence relations, binary search, quicksort, mergesort, heapsort. Dynamic programming

Part 3: Tractability, polynomial time, decision problems, the classes P and NP, many-one reduction, NP-completeness, the $\mathsf{P} = \mathsf{NP}$ problem

Version of January 11, 2024

Chapter 1

Graphs and Graph Algorithms

1.1 Graphs—Basics

1.1.1 Definitions

Graphs are used to model networks of all kinds. They give rise to many interesting computational problems. It is easiest to think of graphs in terms of pictures. A graph is a set of points joined by lines. For instance a road map may be thought of as a graph, with the points being the towns and the lines the roads. In graph terminology, points are referred to as *nodes* (or *vertices*), and lines are referred to as *arcs* (or *edges*).

Figure 1.1 shows a graph with 7 nodes and 8 arcs. We have labelled the nodes. Notice that each arc has two endpoints. We see that graphs may have the following:

- *parallel (multiple) arcs*—two arcs with same endpoints (here 1 and 7)
- *loops*—a loop is an arc with both endpoints the same (here there is a loop on 3)

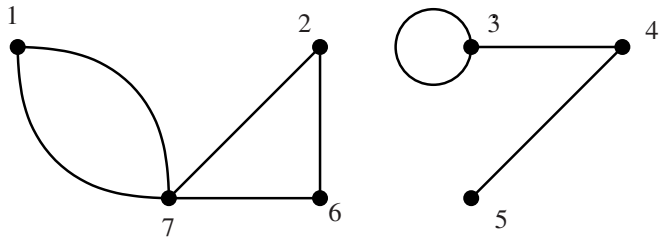


Figure 1.1: A graph

- *disconnected components*—here 1,2,6,7 and 3,4,5

Definition 1.1.1 *An* (undirected) graph *is a set N of nodes together with a set A of arcs, such that each $a \in A$ is associated with an unordered pair of nodes (the* endpoints *of a). A graph is* simple *if it has no parallel arcs and no loops.*

We will refer to graphs by $G, \ldots$. We talk about nodes(G) and arcs(G) in an obvious way. In these notes we only consider finite graphs.

Terminology 1.1.2 *An arc is said to be* incident *on its endpoints. A node n is said to be* incident *on any arc a which has n as one of its endpoints. An arc* joins *two nodes if they are its endpoints. Two nodes n, n' are* adjacent *if they are joined by some arc. The* degree *of a node is the number of arcs incident on it, where loops are counted twice. A node is said to be* odd (even) *if its degree is odd (even).*

For instance in Figure 1.1, the degrees are as follows:

node	1	2	3	4	5	6	7
degree	2	2	3	2	1	2	4

The total of all the degrees is 16, which is twice the number of arcs (8). This is true of all graphs. The reason is that each arc contributes twice to the total, once for each endpoint.

Theorem 1.1.3 *In any graph, the total of the degrees of all the nodes is twice the number of arcs. Moreover the number of odd nodes is even.*

There is a natural notion of subgraph:

Definition 1.1.4 *Let G_1, G_2 be graphs. We say that G_1 is a* subgraph *of G_2 if* nodes$(G_1) \subseteq$ nodes(G_2) *and if* arcs$(G_1) \subseteq$ arcs(G_2).

For example, in Figure 1.2 we see that G_1 is a subgraph of G_2. Notice that nodes in the subgraph G_1 may have fewer connections than nodes in G_2.

Any set $X \subseteq$ nodes(G) *induces* a subgraph $G[X]$ with nodes$(G[X]) = X$ and $G[X]$ inheriting all arcs of G between nodes in X. A graph G' is a *full* (or *induced*) subgraph of G if $G' = G[X]$ for some $X \subseteq$ nodes(G).

If G' is a subgraph of G and nodes$(G') =$ nodes(G), we say that G' *spans* G (or is a spanning subgraph of G).

1.1.2 Representations of Graphs

It is often useful to represent a graph by its *adjacency matrix*. Suppose that a graph has k nodes $n_1, \ldots, n_k$. Define

$$adj(i, j) = \text{the number of arcs joining } n_i \text{ to } n_j$$

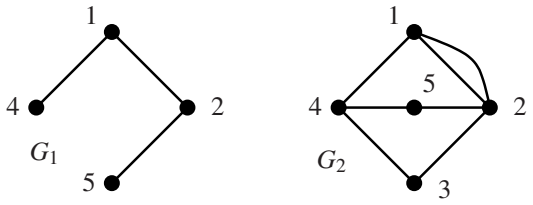


Figure 1.2: G_1 is a subgraph of G_2

For example, the graph in Figure 1.1 has a 7×7 adjacency matrix whose first two rows are:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

As an exercise, complete the remaining 5 rows. You will notice that the matrix is symmetric. This is because the arcs are undirected, so that if node i is joined to j then j must be joined to i. In directed graphs (which we are not considering at present), the arcs have direction—one endpoint is the start and the other the finish. In diagrams the arcs will have arrows on them. We saw such diagrams when considering relations. If we had chosen a different order for the nodes, then we would have obtained a different matrix—the difference being that the rows and columns are reordered (in a consistent way). So a single graph can have a number of adjacency matrices, corresponding to different enumerations of the elements.

If a graph has n nodes then there are n^2 entries in the adjacency matrix. When dealing with a *sparse* graph, i.e. one with much fewer than n^2 arcs, then it can save time and space to represent the graph by its *adjacency list representation*, rather than by its adjacency matrix. The adjacency list representation consists of an array of n pointers, where the ith pointer points to a linked list adj[i] of the nodes which are adjacent to node i. For example, the graph in Figure 1.1 has the following adjacency list representation:

$$\begin{array}{lllllllll}
1 & \to & 7 & \to & 7 & & & & \\
2 & \to & 6 & \to & 7 & & & & \\
3 & \to & 3 & \to & 4 & & & & \\
4 & \to & 3 & \to & 5 & & & & \\
5 & \to & 4 & & & & & & \\
6 & \to & 2 & \to & 7 & & & & \\
7 & \to & 1 & \to & 1 & \to & 2 & \to & 6
\end{array}$$

Notice that each arc is recorded twice (except for loops, which only appear once). If there are n nodes and m arcs, then the total number of entries in the adjacency list representation is bounded by $n + 2m$.

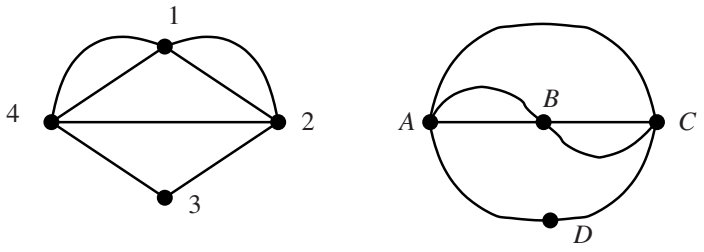


Figure 1.3: Two isomorphic graphs

1.1.3 Big-Oh Notation

Suppose that we wish to multiply together two $n \times n$ matrices, as in the following example for $n = 3$.

$$\begin{pmatrix} 1 & 0 & 2 \\ 3 & 1 & 0 \\ 4 & 2 & 3 \end{pmatrix} \begin{pmatrix} 3 & 6 & 1 \\ 2 & 0 & 5 \\ 1 & 4 & 2 \end{pmatrix} = \begin{pmatrix} 5 & 14 & 5 \\ 11 & 18 & 8 \\ 19 & 36 & 20 \end{pmatrix}$$

Each entry in the product takes n multiplications and $n-1$ additions. Since there are n^2 entries, the total number of multiplications is n^3, the total number of additions is $n^3 - n^2$, and the total number of arithmetical operations is $2n^3 - n^2$. Often we don't wish to be so precise. So we use the 'big-Oh" notation; $O(n^k)$ means bounded by some constant factor times n^k. Thus we can say that the total number of multiplications is $O(n^3)$, the total number of additions is $O(n^3)$, and the total number of arithmetical operations is $O(n^3)$.

For some values of k we use special names:

$O(1)$	constant
$O(n)$	linear (in n)
$O(n^2)$	quadratic (in n)

We shall give the formal definition of $O(n^k)$ later.

1.2 Graph Isomorphism and Planar Graphs

1.2.1 Isomorphism

Consider the two graphs in Figure 1.3. Although the nodes are labelled differently, they are connected in the same way. To make this similarity more precise, let us see how the individual nodes correspond. First, 3 matches with D, since they are the only nodes with degree two. Next, 1 matches with B, since they are not joined to 3, D respectively. Finally 2 matches with A, and 4 matches with C (we could equally well match 2 with C

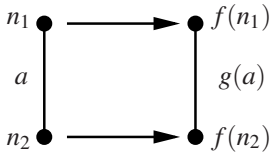


Figure 1.4: Isomorphism

and 4 with A). We have set up a bijection between the two sets of nodes. What is more, this is a bijection which preserves the connectedness of the nodes. Thus the adjacency matrix of the first graph is

$$\begin{pmatrix} 0 & 2 & 0 & 2 \\ 2 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 2 & 1 & 1 & 0 \end{pmatrix}$$

If we order the nodes of the second graph as B, A, D, C we get the same matrix. The bijection we have just given is an example of a graph isomorphism.

Definition 1.2.1 *Let G, G' be graphs. An* isomorphism *from G to G' is a bijection $f : \text{nodes}(G) \to \text{nodes}(G)'$ together with a bijection $g : \text{arcs}(G) \to \text{arcs}(G)'$ such that for any arc $a \in \text{arcs}(G)$, if the endpoints of a are $n_1, n_2 \in \text{nodes}(G)$, then the endpoints of $g(a)$ are $f(n_1)$ and $f(n_2)$. See Figure 1.4. If there is an isomorphism from G to G' we say that G is* isomorphic *to G', or that G and G' are isomorphic.*

It follows that if G is *isomorphic* to G' and $n_1, n_2 \in \text{nodes}(G)$, then $f(n_1)$ and $f(n_2)$ are connected by the same number of arcs in G' as n_1 and n_2 are connected by in G. If two graphs are isomorphic then they must clearly have the same number of nodes. Also their adjacency matrices must be the same, apart from the rows and columns possibly being reordered. If we wish to test whether two graphs are isomorphic, there are a number of obvious checks we can make first. For instance

- do they have the same number of nodes?
- do they have the same number of arcs?
- do they have the same number of loops?
- do their nodes have the same degrees (possibly reordered)?

If they fail tests of this kind then they can't be isomorphic. Suppose that they pass the initial tests. Then we attempt to set up the bijection on nodes. We must match nodes with the same degree, and preserve connectedness, as in the example at the start of the section. If we get the adjacency matrices to match then we know we have been successful. In practice with small graphs the eye can detect an isomorphism, or a reason why there can be no isomorphism, fairly easily. However an algorithm to check

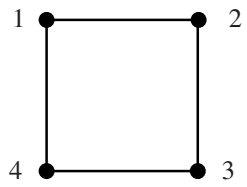


Figure 1.5: A square graph

whether two arbitrary graphs with n nodes each were isomorphic would appear to have to check the $n!$ possible different bijections. This makes the task apparently intractable.

Definition 1.2.2 *Let G be a graph. An* automorphism *on G is an isomorphism from G to itself. It is* nontrivial *if it is not the identity.*

As an example, consider the square graph in Figure 1.5. One possible automorphism maps 1 to 2, 2 to 3, 3 to 4 and 4 to 1. The mapping on arcs is determined once the mapping on nodes is given, since there are no multiple arcs. In this case the automorphism corresponds to the geometrical rotation of the square through 90° clockwise, though in general an automorphism does not necessarily have a geometrical interpretation.

The square graph has other automorphisms. For instance, there is one that maps 1 to 3, 2 to 2, 3 to 1 and 4 to 4, corresponding to reflecting the square across the diagonal from 2 to 4. Suppose we wish to count the number of automorphisms of a graph. We could just list all possibilities, but this could be tedious and we might miss some. A good system is to go through the nodes 1 to n in turn, seeing how many possibilities there are for where node i can go, given that nodes 1 to $i-1$ are already fixed. We then multiply together the numbers of possibilities.

Let us follow this system for the square: To start with, 1 can map to any of the four nodes. Given that 1 is fixed, 2 can map to only two of the remaining three nodes, namely the nodes which are adjacent to where 1 has been assigned. Once both 1 and 2 are fixed, the positions of nodes 3 and 4 are determined. There are therefore $4 \times 2 = 8$ possible automorphisms. One of these is of course the identity.

1.2.2 Planar Graphs

In our diagrams we are perfectly happy to allow arcs to cross. For instance the "cube" graph of Figure 1.6 is fine. For some applications it is better to have graphs whose arcs do not cross—for instance if the graph represents a circuit which is to be engraved on a microchip. Say that a graph is *planar* if it can be embedded in the plane (by rearranging the nodes and arcs).

Exercise 1.2.3 *Show that the cube graph of Figure 1.6 is planar, by redrawing the arcs so that they do not cross.*

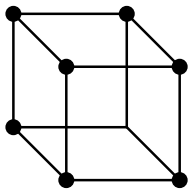


Figure 1.6: A "cube" graph

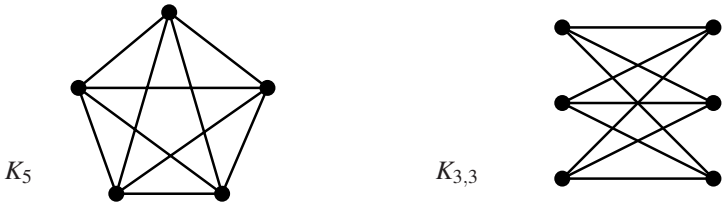


Figure 1.7: Two non-planar graphs

In fact, if a (simple) graph is planar, we can always draw the arcs as straight lines which do not cross.

Exercise 1.2.4 *If you have not already done so, redraw the cube graph with straight lines which do not cross.*

Figure 1.7 shows two examples of non-planar graphs. K_5 is the *complete* graph on five nodes (where every node is joined to every other node). $K_{3,3}$ is the graph with all possible arcs between two sets of size 3. There is no way to redraw either of these so that no arcs cross (though one can reduce the number of crossings).

Definition 1.2.5 *Two graphs are* homeomorphic *if they can be obtained from the same graph by a series of operations replacing an arc $x-y$ by two arcs $x-z-y$.*

Theorem 1.2.6 (Kuratowski, 1930) *A graph is planar iff it does not contain a subgraph homeomorphic to K_5 or $K_{3,3}$.*

So in a sense Figure 1.7 shows all possible ways a graph can fail to be planar.

Hopcroft & Tarjan (1976) gave a linear time algorithm for testing whether or not a graph is planar. "Linear time" here means $O(n+m)$ where n is the number of nodes and m is the number of arcs.

Any planar graph splits the plane into various regions which are called "faces". For instance, the cube graph has 6 faces (the region surrounding the graph counts as a face). Euler gave a formula connecting the numbers of nodes, arcs and faces:

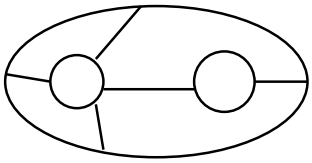


Figure 1.8: A map

Theorem 1.2.7 (Euler, 1752) *Let G be a connected planar graph. Let G have N nodes, A arcs and F faces. Then $F = A - N + 2$.*

1.2.3 Colouring Maps and Graphs

We conclude this section by mentioning a very famous result related to planar graphs—the Four Colour Theorem. Imagine that we have a map and we want to colour the countries in such a way that no two neighbouring countries have the same colour (meeting at a point does not count as sharing a border). How many colours do we need? For instance, how many colours do we need to colour the map in Figure 1.8?

Each map can be turned into a graph by placing a node inside each country (its "capital city"), and joining two nodes precisely if the countries share a border. Call this the *dual graph* of the map. Clearly the dual graph will be (simple and) planar. Try drawing the dual graph for the map in Figure 1.8. Also, any simple planar graph is the dual graph of some map.

Exercise 1.2.8 *Find a map whose dual graph is the cube graph of Figure 1.6. How many colours does it need?*

Any colouring of a map gives rise to a colouring of the *nodes* of the dual graph, in such a way that no two adjacent nodes have the same colour. Say that a map is *k-colourable* if it can be coloured using no more than k colours.

Definition 1.2.9 *A graph G is* k-colourable *if* $\mathrm{nodes}(G)$ *can be coloured using no more than k colours, in such a way that no two adjacent nodes have the same colour.*

So colouring simple planar graphs is equivalent to colouring maps. Colouring graphs is an interesting problem even when the graphs are not planar.

$K_{3,3}$ (Figure 1.7) is an example of a bipartite graph, i.e. a graph which can be split into two parts with all arcs going across from one part to the other.

Definition 1.2.10 *A graph G is* bipartite *if* $\mathrm{nodes}(G)$ *can be partitioned into two sets X and Y in such a way that no arc of G joins any two members of X, and no arc joins any two members of Y.*

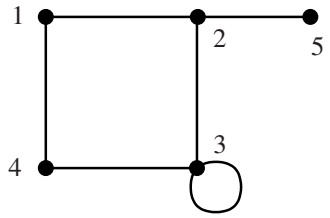


Figure 1.9: A graph

We drew $K_{3,3}$ in such a way that it was obviously bipartite. There are other cases where graphs are bipartite but not obviously so. For instance, the cube graph of Figure 1.6 is bipartite.

Proposition 1.2.11 *A graph is bipartite iff it is* 2*-colourable.*

Returning to map-colouring, for a very long time it was conjectured that every map is 4-colourable, but no proof could be found. The result was finally shown with the aid of a computer which generated and checked 1482 possible cases.

Theorem 1.2.12 (Four Colour Theorem, Appel & Haken 1976) *Every map (equivalently, every simple planar graph) is 4-colourable.*

With such a lengthy proof there were naturally concerns about its correctness. A different computer proof was produced in 1994. This proof was checked by Gonthier and Werner in 2004 using the general-purpose proof assistant Coq. This means that we do not have to trust the various ad hoc computer programs used to check the various cases—we only have to rely on the correctness of Coq.

Of course, if a graph is not planar we may well need more than four colours to colour it.

1.3 Connectedness, Euler Paths and Hamiltonian Circuits

1.3.1 Paths and Cycles

A *path* in a graph is a sequence of nodes and arcs $n_1, a_1, n_2, a_2, \ldots, a_{k-1}, n_k$ such that each a_i joins n_i to n_{i+1}. We generally describe paths by sequences of nodes, omitting the arc names. As an example, consider the graph in Figure 1.9. Possible paths include $1,2,3,4$ and $2,5,2,3,3,4$. Notice that the arc joining 2 to 5 is used twice (in opposite directions) in the latter case. The *length* of a path is the number of arcs in the path. If

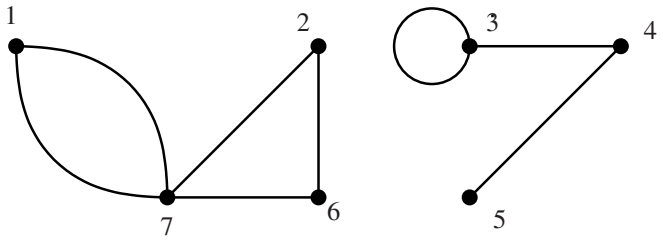


Figure 1.10: A disconnected graph

an arc is used twice it is counted twice. Thus the length of $2,5,2,3,3,4$ is 5. We allow zero-length paths.

A path is *simple* if it has no repeated nodes.

A *cycle* is a path which finishes where it started, i.e. $n_1 = n_k$, which has at least one arc, and which does not use the same arc twice. Cycles are sometimes called *circuits*. In Figure 1.9 possible cycles include $1,2,3,4,1$ and $3,3$. The path $2,5,2$ is not a cycle, since it uses the same arc twice.

Of course, not every graph has a cycle. If a graph has no cycles we say that it is *acyclic*.

Definition 1.3.1 *A graph is* connected *if for any two different nodes n, n' there is a path from n to n' (n and n' need not be joined directly).*

For example, the graph in Figure 1.9 is connected. However the graph in Figure 1.10 is not, since there is no path from 2 to 3 (for instance). Given a graph G, we can define a relation on nodes(G) by: $n \sim n'$ iff there is a path from n to n'. It is easy to see that this is an equivalence relation (reflexive, symmetric and transitive). The equivalence classes of $\sim$ are called the *(connected) components* of G. Each component is a connected graph, and there is no path from a node in one component to a node in another component. For instance, in Figure 1.10, there are two connected components, one with nodes $1,2,6,7$ and one with nodes $3,4,5$.

1.3.2 Euler Paths and Circuits

In the eighteenth century the Swiss mathematician Leonhard Euler (1707-83) considered the *Königsberg Bridge Problem*, which concerns the town of Königsberg[1], where there were two islands in the river Pregel, giving four regions connected by seven bridges as in the schematic map (left-hand diagram) of Figure 1.11. The problem is to start at any of the four regions and cross each bridge exactly once. Euler converted the map into a graph with four nodes and seven arcs (right-hand diagram in Figure 1.11).

[1] Then in Prussia. Now called Kaliningrad and part of Russia, lying between Lithuania and Poland.

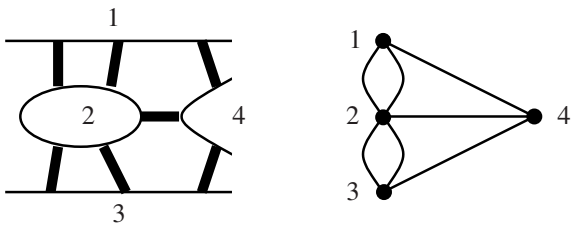


Figure 1.11: The Königsberg schematic map and Euler's graph

The task now is to start at any node and travel along each arc exactly once. Notice that the degrees of 1,2,3,4 are 3,5,3,3, respectively. If we think of the arcs as railway lines joining cities then we have what can be called the *railway inspector's problem*: find a path which traverses every mile of track exactly once.

Terminology 1.3.2 *An* Euler *path is a path which uses each arc exactly once. An* Euler circuit (Euler cycle) *is a cycle which uses each arc exactly once, i.e. a cycle which uses all the arcs of the graph.*

Suppose that a graph G has an Euler path P starting at node n_0 and ending at n_k. If P passes through a node n, it must enter n as many times as it leaves it. Hence the number of arcs incident on n which are used by P must be even. So if n has odd degree, then P cannot use all arcs incident on n—there must be at least one left over. So every node of G must have even degree except for n_0 and n_k. By similar reasoning, if n_0 and n_k are different then they must have odd degree, and if they are the same (so that P is in fact a cycle) then $n_0 = n_k$ must have even degree. Putting this together shows that if a graph has an Euler path then either exactly two nodes have odd degree or every node must have even degree Clearly the Königsberg bridge problem cannot be solved, since the graph has 4 odd nodes.

Theorem 1.3.3 (Euler, 1736) *A connected graph has an Euler path iff the number of odd nodes is either* 0 *or* 2*. A connected graph has an Euler circuit iff every node has even degree.*

The theorem gives us a very simple algorithm to check whether a connected graph has an Euler path (or circuit): just count the number of odd nodes. This can be done in $O(n^2)$ time, where n is the number of nodes—just make a single pass through the adjacency matrix row by row, maintaining a counter for the number of odd nodes and a counter for the degree of the node corresponding to the current row.

Remarks on the proof of Theorem 1.3.3. We have just seen that the condition (the number of odd nodes is either 0 or 2) is necessary. It is also sufficient. A rigorous proof of this is beyond the scope of these notes. So instead we sketch an algorithm for finding an Euler path in a connected graph where exactly two nodes (n, n') have odd

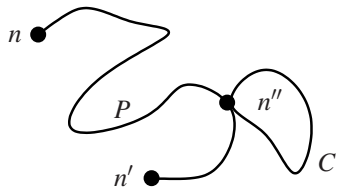


Figure 1.12: Finding an Euler path

degree: Start at node n and follow any path until you can go no further. You must have reached n'. This is because you can't have finished at n, since you would then have used up evenly many arcs incident on n, and there would be a spare arc. Similarly, you can't have stopped at n'' different from n, n', since you would have used up an odd number of arcs incident on n'', and n'' has even degree. Notice that now every node has an even number of unused arcs. Call the path so far P. If you have an Euler path then stop. Otherwise take some node n'' on your path so far which has unused arcs (such a node must exist by the connectedness condition), and start a new path there.This must return to its start n'' to form a cycle C, since if it stopped at any other node there would be an unused arc. Now merge P and C to form a longer path (see the diagram in Figure 1.12). The problem is to start at any of the four regions Keep on doing this until all arcs are exhausted. This gives the desired Euler path.

1.3.3 Hamiltonian Circuits

Very similar to the problem of whether we can traverse every arc of a graph exactly once is the problem of whether we can find a path which visits every *node* exactly once.

Definition 1.3.4 *A* Hamiltonian[2] path *is a path which visits every node in a graph exactly once. A* Hamiltonian circuit *is a cycle which visits every node exactly once. The* Hamiltonian circuit problem *(HCP) is: given a graph G, determine whether G has a Hamiltonian circuit. The Hamiltonian path problem is defined in a similar fashion.*

For instance, the graph in Figure 1.13 has a Hamiltonian path, but no Hamiltonian circuit (once node 1 is reached, which must be from 2, we cannot leave it without repeating node 2).

Exercise 1.3.5 *Find a graph with no Hamiltonian path.*

We can make some easy observations about whether a graph has a Hamiltonian circuit. Firstly, we may as well only consider simple graphs, since removing loops and par-

[2]Named after William Rowan Hamilton (1805-65), who first posed the problem

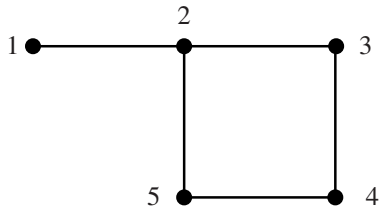


Figure 1.13: A graph with a Hamiltonian path

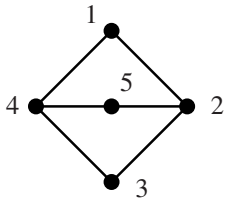


Figure 1.14: A graph with no Hamiltonian circuit

allel arcs makes no difference to whether a Hamiltonian circuit exists. Secondly, two conditions which are clearly necessary are:

1. The graph must be connected (also needed for a Hamiltonian path to exist).
2. Each node must have degree at least two, since we must approach it and leave it via two different nodes.

But even with both these conditions satisfied, there is no guarantee of a Hamiltonian circuit existing. For instance, the graph in Figure 1.14 does not have a Hamiltonian circuit.

Of course, since graphs are finite, we can simply check each possible circuit in turn and see whether it is Hamiltonian. Suppose that G has n nodes labelled $1,\ldots,n$. Possible Hamiltonian circuits involve all n nodes exactly once. So they can be described by a permutation π of $1,\ldots,n$. Such a permutation is a bijection $\pi : \{1,\ldots,n\} \to \{1,\ldots,n\}$. There are $n!$ different permutations of $1,\ldots,n$. We generate and check each possible circuit π to see whether it is an actual circuit in G. Checking π means asking for each $i = 1,\ldots,n$: is $\pi(i)$ is adjacent to $\pi(i+1)$? Here we adopt the convention that $n+1$ is identified with 1. For instance, letting G be the graph in Figure 1.14, the possible circuit $5,4,3,1,2$ is not an actual circuit, since 3 is not adjacent to 1. The checking of each π can be done quickly—it is $O(n)$. The problem is the number of permutations to be checked: $n!$ is worse than exponential (larger than 2^n). As n gets large our method becomes far to slow to be practical.

A better algorithm is known — a 'dynamic programming' method, due to Bellman, Held, and Karp (1962), can solve HCP in $O(n^2 2^n)$ which is still exponential, of course. This will be discussed later in the course.

Of course, the above discussion does not rule out the existence of some clever algorithm that solves the HCP much more quickly. But such an algorithm has not been found, and in fact there is good reason to think that it will not be, since HCP has been shown to be *NP-complete*. NP-complete problems are believed not to admit polynomial (that is, $O(n^k)$ for some k) solutions.

Thus the HCP is strongly contrasted with the Eulerian path problem, since we saw that the latter can be solved in polynomial (in fact $O(n^2)$) time.

1.4 Trees

Definition 1.4.1 *A* rooted graph *is a pair (G,x) where G is a graph and $x \in \text{nodes}(G)$. The node x is the* root *of the tree. A* tree *is a rooted, acyclic, connected graph. An* nonrooted *tree is an acyclic, connected graph.*

Note that sometimes the word "tree" is used to refer to nonrooted trees. In this case what we have called a "tree" will be called a "rooted tree". We use $T, \ldots$ to stand for trees.

In a tree there is a unique (non-repeating) path between any two nodes. This is because if there were two even partially different paths then the tree would have a cycle, which is not allowed. We can therefore meaningfully talk about the *depth* of a node x, which is defined to be the length of the (unique) path from the root to x. If x is not the root, we can also define the *parent* of x to be the unique node which is adjacent to x on the path from x to the root.

The *depth* of a tree is the maximum of the depths of all its nodes.

Now any arc a of a tree T joins a unique non-root node $f(a)$ to its parent. Also, if x is a non-root node then there is a unique arc $g(x)$ which joins x to its parent. Clearly f and g are mutual inverses. So there is a bijection between non-root nodes and arcs. If T has n nodes, then it has $n-1$ non-root nodes. Hence:

Proposition 1.4.2 *Let T be a tree with n nodes. Then T has $n-1$ arcs.*

The same is true for nonrooted trees—just make any node into the root.

Definition 1.4.3 *Let G be a graph. An nonrooted tree T is said to be a* spanning tree *for G if T spans G, i.e. T is a subgraph of G and* $\text{nodes}(T) = \text{nodes}(G)$.

Suppose that G is a connected graph. Then we can obtain a spanning tree as follows: If G has a cycle C then remove any arc of C, joining nodes x and y, say. Now there

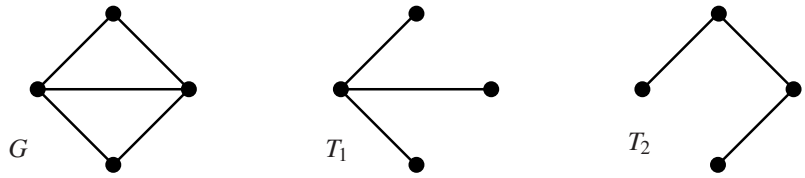


Figure 1.15: A graph and two spanning trees

is still a path from x to y going round the remainder of C. Hence the new graph G_1 is still connected. Also we have nodes(G_1) = nodes(G). Continue this process to get graphs $G_1, G_2, \ldots$. The process must terminate (why?). Eventually we must arrive at an acyclic graph, which will be a spanning tree for G. Hence:

Proposition 1.4.4 *Let G be a connected graph. Then G has a spanning tree.*

Spanning trees are not necessarily unique. Graph G in Figure 1.15 has both T_1 and T_2 as spanning trees. Of course, any two spanning trees for the same graph with n nodes must have the same number of arcs, namely $n-1$.

1.5 Directed Graphs

So far we have only considered *undirected* graphs. For many applications it makes sense for arcs/edges to be *directed*.

Definition 1.5.1 *A* directed graph *is a set N of nodes and a set A of arcs such that each $a \in A$ is associated with an* ordered *pair of nodes (the* endpoints *of a)*

- In diagrams the arcs are shown with arrows from source node to target node.
- In a path $a_1, \ldots, a_n$ in a directed graph the source of a_{i+1} must match the target of a_i (for $i = 1, \ldots, n-1$)
- If for any pair of nodes x, y there is at most one arc from x to y then we can refer to this arc as (x, y).

Definition 1.5.2 *The* indegree *of a node x is the number of arcs entering x. The* outdegree *of a node x is the number of arcs leaving x.*

It is clear that for any directed graph, the sum of the indegrees of all nodes equals the sum of the outdegrees, which both equal the number of arcs.

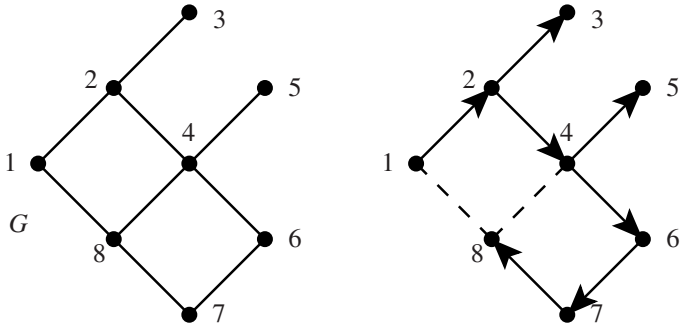


Figure 1.16: Depth-first Search

Definition 1.5.3 *A directed graph is* strongly connected *if for any* $x, y \in \text{nodes}(G)$ *there is a path from x to y.*

So for any pair of nodes x, y we need paths both from x to y and from y to x.

Convention: In this course (though not necessarily elsewhere) by default a graph is undirected unless we state that it is directed.

1.6 Graph Traversal

Suppose that we wish to visit and list every node in a connected graph by following some path. We shall not mind if we reuse arcs or revisit nodes (though if we revisit a node we don't write it down again). This process is called a *traversal* of the graph. In this section we shall look at various algorithms for graph traversal.

1.6.1 Depth-first and Breadth-First Search

In *depth-first search (DFS)* we start from a particular node `start` and we continue outwards along a path from `start` until we reach a node which has no adjacent unvisited nodes. We then backtrack to the previous node and try a different path. The procedure continues until we have backtracked to `start` and all nodes adjacent to `start` are visited. At this point every node will have been visited.

We illustrate on graph G in Figure 1.16. The starting node is 1. From 1 we go to 2. We could have chosen 8 instead. Next we go to 3. The path can go no further, and so we backtrack to 2. From here we go to 4 and 5 (we could have chosen 6 instead of 5). We backtrack to 4 and then go to 6, 7 and 8. At 8 we see that all adjacent nodes are visited. We backtrack to 7, 6, 4, 2 and 1, at which point the DFS is complete. The right hand diagram in Figure 1.16 shows the forward direction of the search. We see that it forms

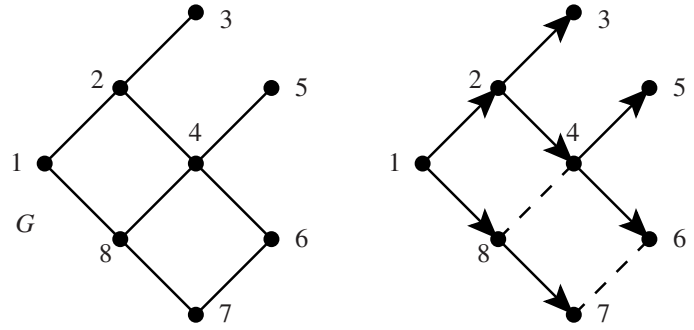


Figure 1.17: Breadth-first Search

a spanning tree. The root of the tree is the start node. Notice that node 8 is at depth 5 in the search tree, while its shortest distance from the start node is only 1.

In *breadth-first search (BFS)* we start from a particular node `start` and we fan out from there to all adjacent nodes, from which we then run breadth-first searches.

In our example graph G (Figure 1.17) we start at node 1. We next go to 2 and 8 (in either order). Next we fan out from 2 to visit 3 and 4 from 2, and 7 from 8. We don't visit 4 from 8, since we already visited 4 from 2. Next we visit 5 and 6 from 4 and finally 7 from 8. Summing up, the order of visiting the nodes is $1,2,8,3,4,7,5,6$. We get a spanning tree from BFS which is different from the DFS tree. In terms of the BFS tree, we visit all the nodes at depth k before visiting any node at depth $k+1$. Notice also that the depth of a node in the BFS tree is its shortest distance from the start node.

Both search procedures traverse all nodes in a connected graph, but the order of visiting is different. For some purposes either procedure will do, while for other purposes one procedure is to be preferred.

Let us now formalise DFS and BFS as algorithms. We assume that the graph to be traversed is given as an adjacency list (Section 1.1.2). We shall create a Boolean array to keep track of visited nodes, and a parent function to keep track of the search tree. We shall output nodes in the order visited. Initially no nodes are visited.

DFS is most naturally carried out by recursion:

Algorithm Depth-first Search (DFS):
```
procedure dfs(x):
   visited[x] = true
   print x
   for y in adj[x]:
      if not visited[y]:
         parent[y] = x
         dfs(y)
```

```
    # backtrack to x
```

Once DFS is completed we see that every node has a parent except for the start node. The parent of a node is its parent in the DFS tree. For instance, in our example the parent of 8 is 7. Of course, if we don't need the parent information we can omit the line parent[y] = x from the code.

It is easy to see that dfs(x) is applied to each node at most once (in fact it will be exactly once if the graph is connected). Also each application of dfs(x) runs through the arcs incident on x (i.e. adj[x]) exactly once. Therefore the running time of DFS is $O(n+m)$, where n is the number of nodes and m is the number of arcs.

BFS is naturally expressed using a *queue* of nodes. Queues store items in such a way that they leave the queue in the same order that they were placed in the queue—FIFO (first in, first out) order. The queue is initialised to contain just the start node x. We then process nodes from the front of the queue. For each node we visit its immediate neighbours, adding them to the back of the queue.

Algorithm Breadth-first Search (BFS):

```
visited[x] = true
print x
enqueue(x, Q)
while not isempty(Q):
   y = front(Q)
   for z in adj[y]:
      if not visited[z]:
         visited[z] = true
         print z
         parent[z] = y
         enqueue(z, Q)
   dequeue(Q)
```

Let us look at how this works on our example graph G. Let us write the queue as a list with the head of the queue to the left and nodes added to the right. Initially the queue is $[1]$. We process 1, adding 2 and 8 and removing 1 to get $[2,8]$. We then process 2 to get $[8,3,4]$, and so on. The nodes are added to the queue in the order $1,2,8,3,4,7,5,6$ and removed in the same order. Notice that the queue grows and shrinks during the computation. The size of the queue represents in some sense the "breadth" of the front on which the traversal is being carried out.

Much as with DFS we see that each node is processed once and each adjacency list is processed once. Therefore the running time of BFS is again $O(n+m)$.

1.6.2 Applications

We discuss three applications of DFS and BFS.

Determining Whether a Graph Is Connected

So far we have assumed that the graph to be traversed is connected. But we can traverse non-connected graphs as well. Of course we will only visit nodes which are in the same connected component as the start node. We can easily adapt either DFS or BFS to return a list of visited nodes. Clearly the graph is connected iff this list is the same (up to reordering) as the complete list of nodes. This gives us an $O(n+m)$ algorithm to determine whether a graph is connected.

Determining Whether a Graph Has a Cycle

Suppose a connected graph has n nodes. If it has $\geq n$ arcs then it contains a cycle (exercise: in fact this holds even if graph not connected). We can use this to check easily whether a graph has a cycle. But this method does not find a cycle. As an alternative, we can use DFS.

Proposition 1.6.1 *Let G be a connected graph, and let T be a spanning tree of G obtained by DFS starting at node* start. *If a is any arc of G (not necessarily in T), with endpoints x and y, then* either *x is an ancestor of y in T* or *y is an ancestor of x in T.*

Here 'x is an ancestor of y in T' means that x lies on the (unique) path from start *to y in T.*

Proof. Suppose that we visit x before visiting y. Then we must visit y from x directly (x is the parent of y) unless we have already visited y via calling DFS on some other node z adjaceent to x. In either case x is an ancestor of y. Similarly, if we visit y before x then y is an ancestor of x.

Suppose that we are using DFS to traverse a connected graph. If when at node x we encounter a node y which we have already visited (except by backtracking), this tells us that the node can be approached by two different routes from the start node. Hence there is a cycle in the graph. By the proof of Proposition 1.6.1 node y must be an ancestor of x.

Conversely, if we never encounter an already visited node, it is reasonably clear that the graph is in fact a tree, with no cycles (this statement would require some proof, which we omit). Therefore we can adapt DFS to test whether a graph has a cycle.

Algorithm Test for Cycles Using DFS:

```
procedure cycleDfs(x):
   visited[x] = true
   # print x
   for y in adj[x]:
      if visited[y] and y ≠ parent[x]:
         # cycle found involving x and y
         return (x,y)
```

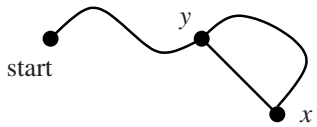


Figure 1.18: Using DFS to test for cycles

```
if not visited[y]:
    parent[y] = x
    pair = cycleDfs(y)
    # backtrack to x
    if pair:
        return pair
```

The algorithm can be used to find a cycle if one exists. Suppose that nodes x and y are returned. Then y must be an ancestor of x, as in Figure 1.18. The cycle will be x, parent[x],. . . ,y, x.

If we apply the algorithm to our example graph G in Figure 1.16 the first cycle we find is when processing node 8. We find that node 1 is already visited, and return $(8, 1)$. This corresponds to the cycle $8, 7, 6, 4, 2, 1, 8$.

We could have used BFS instead of DFS.

Calculating Distances From the Start Node

Comparing Figures 1.16 and 1.17 we see that BFS finds the shortest path from the start node to any reachable node, while DFS may well give a longer distance than necessary. We can easily adapt BFS to calculate the distance from the start node. We add a new function distance which records the depth of each visited node in the BFS tree.

Algorithm Shortest Path (Unweighted Graph):

```
visited[x] = true
distance[x] = 0
enqueue(x, Q)
while not isempty(Q):
    y = front(Q)
    for z in adj[y]:
        if not visited[z]:
            visited[z] = true
            parent[z] = y
            distance[z] = distance[y] + 1
            enqueue(z, Q)
    dequeue(Q)
```

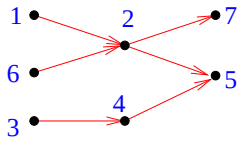


Figure 1.19: Directed graph of tasks

Clearly the shortest path from a node y to the start node can be read off from the parent array as y, parent[y], parent[parent[y]], ..., `start`.

Topological Sorting

Suppose that we have a list of tasks to be completed and some tasks have to be completed before others. We can view them as a directed graph, with arcs representing the dependincies, so that an arc (x,y) denotes that task x must be completed before task y. See Figure 1.19. This graph must be *acyclic* or else we cannot find an order in which to complete the tasks.

The *topological sorting problem* is as follows: Given a directed acyclic graph (DAG) G with n nodes, find a total ordering of the nodes $x_1,\ldots,x_n$ such that for any $i,j \leq n$, if $j > i$ then there is no path from x_j to x_i in G. Such a total ordering is called a *topological sort* of G. It could be presented as a list or array of nodes.

For the diagram in Figure 1.19 a topological sort could be $1,6,3,2,4,7,5$ or $6,1,2,7,3,4,5$, etc.

DAGs are closely related to partial orderings.

Definition 1.6.2 *A weak partial ordering on a set X is a binary relation $\leq$ satisfying:*

1. *reflexive:* $\forall x \in X . x \leq x$
2. *transitive:* $\forall x,y,z \in X$, *if* $x \leq y$ *and* $y \leq z$ *then* $x \leq z$
3. *antisymmetric:* $\forall x,y \in X$, *if* $x \leq y$ *and* $y \leq x$ *then* $x = y$

As an example, the inclusion ordering $\subseteq$ on sets is a weak partial ordering.

Given a DAG G, let $x \leq y$ iff there is a path from x to y. Then $\leq$ is a (weak) partial ordering on nodes(G).

Conversely, if $(X,\leq)$ is a partial ordering, let G be the directed graph with nodes X and arcs $\{(x,y) : x \leq y\}$. Then G is acyclic.

So a topological sorting of a DAG amounts to a *linearisation* of a partial ordering, i.e. a linear order which extends the partial ordering.

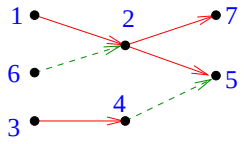


Figure 1.20: The forest got from applying DFS to the graph in Figure 1.19

We can perform topological sorting using DFS. The idea is that when we have finished processing a node x we must have finished with all nodes which are reachable from x, and which must come after x in the sorting. So we can add a node to the sorted list (starting from the top end) once we have finished processing it (once we exit the node). What we here call 'exiting' is just what we earlier called 'backtracking' (unless we are at a start node for DFS).

Consider the graph in Figure 1.19. If we apply DFS we enter (visit) the nodes in the order

$$1,2,5,7,3,4,6$$

assuming that the adjacency lists are given in numerical order. See Figure 1.20 for the resulting DFS forest. We exit the nodes in the order

$$5,7,2,1,4,3,6$$

By reversing the order in which the nodes are exited we get the sort $6,3,4,1,2,7,5$.

Here is the pseudocode for DFS adapted to perform topological sorting:

Algorithm Topological sort:
Given: a directed graph G with n nodes.
Return: topological sort of G as array ts of nodes if G acyclic (else abort).

```
procedure dfsts(x)
   entered[x] = true
   for y in adj[x]:
      if entered[y]:
         if not exited[y]: abort      # cycle
      else:
         parent[y] = x
         dfsts(y)
   exited[x] = true
   ts[index] = x
   index = index −1

index = n − 1
for x in nodes(G):
   if not entered[x]:
      dfsts(x)
```

Note that we add nodes to the array working downwards as they are exited.

We now show:

1. if the algorithm terminates then it produces a topological sort;
2. if the algorithm aborts then there is a cycle.

Proposition 1.6.3 *When performing the algorithm on a directed graph, when we exit a node x we have already exited all nodes reachable from x.*

Proof. Suppose that this is not the case. Let x be the first node that we exit but where there is some node z reachable from x that has not been exited. Since $z \neq x$, clearly z must be reachable from some y in adj[x]. But by the code we have already exited from y. By assumption this means that we have exited from all nodes reachable from y, including z. Contradiction.

We can deduce from Proposition 1.6.3 that if there is a cycle then the computation can never exit from any node of the cycle, and so must abort.

From the code we can see that if the computation does not abort:

1. Each node is entered exactly once;
2. Each node is exited after entry;
3. upon exit each node is added to the array ts in descending order.

Suppose for a contradiction that x is added to the array after y but that there is a path from y to x. Then we exited x after we exited y. But by Proposition 1.6.3 we can only exit y after exiting x since x is reachable from y. Contradiction. We conclude that if the computation does not abort then it produces a topological sort.

Proposition 1.6.4 *When performing DFS on a directed graph, if we enter a node x after entering a node y but before exiting y then x is reachable from y.*

Proof. Suppose that this is not the case. Let x be the first node that we enter after entering some node y but before exiting y where x is not reachable from y. Let z be the last node which is entered but not exited before entering x (so that z may or may not be y). Since y has not been exited when entering x, the invocation of DFS on x is not at the top (non-recursive) level and so we must have entered x as part of processing z, and so x is in adj[z]. But by our assumption z is reachable from y, and so x is reachable from y. Contradiction.

Finally we argue that if the computation aborts then there is a cycle. Suppose we abort while running dfsts(x) and discovering that y in adj[x] has been entered but not exited. By Proposition 1.6.4 we see that x is reachable from y. Hence we have a cycle involving x and y.

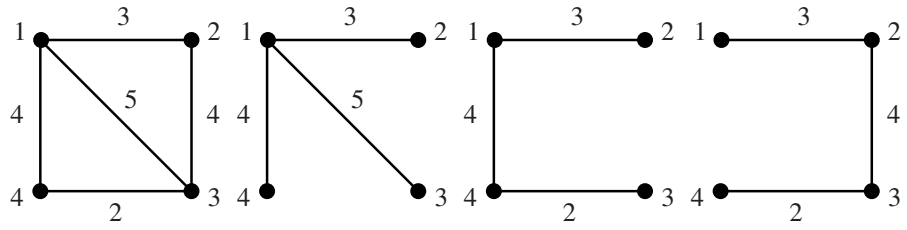


Figure 1.21: Minimum and non-minimum spanning trees

1.7 Weighted Graphs

Many networks have a *cost* associated with each arc. For example, in a transport network, the cost of an arc could be the distance or the travelling time. Such costs are often referred to as *weights*.

Definition 1.7.1 *A* weighted graph *is a simple graph G together with a weight function $W : \text{arcs}(G) \to \mathbf{R}^+$ (where $\mathbf{R}^+$ denotes the real numbers ≥ 0).*

We shall look at the problems of finding a minimum spanning tree, finding shortest paths, and finding a shortest circuit. These are all problems where we are trying to minimise cost, and so the restriction to simple graphs is sensible; if there are two parallel arcs then we will always choose the cheaper, and if there is a loop then we will never wish to use it. With simple graphs an arc can be specified uniquely by giving its endpoints, and we can regard the weight function as acting on pairs of nodes.

1.7.1 Minimum Spanning Trees

Recall that every connected graph has a spanning tree. When dealing with weighted graphs we would naturally like to find a *minimum* spanning tree, that is, a spanning tree where the sum of the weights of its arcs is as small as possible.

For instance, if we had to build a road network joining some cities, a minimum spanning tree would represent the cheapest network which would connect all the cities (assuming that we cannot fork roads except at a city).

Definition 1.7.2 *Let G be a weighted graph. The* weight *of a spanning tree T for G is the sum of the weights of the arcs of T. T is a* minimum spanning tree (MST) *for G if T is a spanning tree for G and no other spanning tree for G has smaller weight.*

Consider the graph drawn on the left in Figure 1.21. We show three spanning trees. The first has a weight of 12, while the other two both have a weight of 9. The last two clearly have minimum weight, since any spanning tree must have three arcs, and

they have selected the three arcs of least weight. The example shows that minimum spanning trees are not necessarily unique.

1.7.2 Prim's Algorithm

Suppose we want to grow an MST starting from a root node. At each stage the best thing we can do in the short term is to add the shortest arc which will extend the tree. This is the so-called "greedy" approach, where we do what gives a short-term advantage, even if it may not be the best overall. Let us apply this approach to the graph in Figure 1.21, choosing 1 as our start node. Since the arc to 2 is the shortest, we add it to our tree. Now we look for the shortest arc which joins either 1 or 2 to one of the remaining nodes. We can choose either the arc $(1,4)$ or the arc $(2,3)$. Suppose we choose $(1,4)$. We now look for the shortest arc which joins node 3 to the tree. This will be the arc $(4,3)$. We end up with the first of the two MSTs in Figure 1.21. The greedy algorithm we used is due to Prim.

At an arbitrary stage in Prim's MST algorithm, there are three kinds of nodes:

- those which are in the tree constructed so far (tree nodes),
- those which are candidates to join at the next stage (so-called *fringe* nodes), and
- the rest (so-called *unseen* nodes)

Initially all nodes are unseen. Here is an informal statement of the algorithm:

Algorithm Prim's MST Algorithm—Scheme:
Choose any node start as the root
Reclassify start as tree
Reclassify all nodes adjacent to start as fringe
while fringe nonempty:
  Select an arc of minimum weight between a tree node t and a fringe node f (*)
  Reclassify f as tree
  Add arc (t,f) to the tree
  Reclassify all unseen nodes adjacent to f as fringe

If we analyse the algorithm as it stands, we see that each time the while loop is executed another node is added to the tree. Hence the while loop is executed $O(n)$ times. The line marked (*) involves finding the shortest arc among all possible arcs between $O(n)$ tree nodes and $O(n)$ fringe nodes. This is therefore $O(n+m)$, which makes the whole algorithm $O(n(n+m))$.

We can improve the performance of the algorithm if we keep track of which arcs might be used. Consider the example graph of Figure 1.21 when the tree so far is just the arc $(1,2)$. The fringe nodes are 3 and 4. The arc $(1,3)$ of weight 5 was an option before 2 was added, but now it is longer than $(2,3)$ and therefore no longer a candidate. The candidate arcs are $(1,4)$ and $(2,3)$.

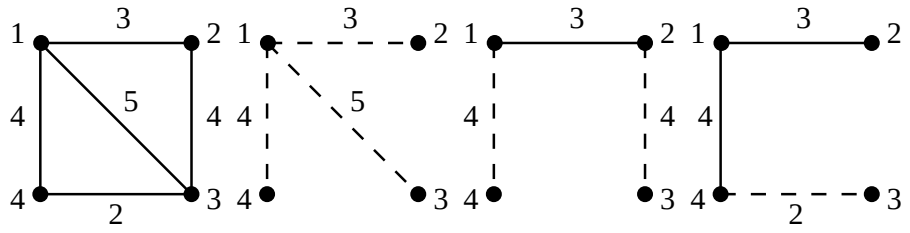


Figure 1.22: Candidate arcs (dashed lines)

It is convenient to look at this using the parent array we adopted for graph traversal. We let the parent of a node f in the fringe be the node in the tree t such that (t, f) has least weight. So initially in our example parent[3] = 1, but after 2 is added we change to parent[3] = 2, making $(2,3)$ a candidate arc. See Figure 1.22.

This leads us to recast Prim's Algorithm as in Figure 1.23. Clearly we can obtain the weight of the MST by just adding the weights of the nodes (apart from start).

We analyse this more refined version:

- As before there are $O(n)$ executions of the while loop.
- Testing whether the fringe is empty is $O(n)$.
- Finding the fringe node f such that weight[f] is minimum is $O(n)$.
- The updating of the candidate arc for each y in adj[f] is $O(1)$. Hence the for loop is $O(n)$.
- We conclude that the algorithm is $O(n^2)$, which is an improvement on our earlier estimate of $O(n(n+m))$ (recall that m can be as large as n^2).

We now turn to the correctness of Prim's Algorithm.

Theorem 1.7.3 *Let G be a connected weighted graph. Then Prim's algorithm constructs an MST for G.*

Proof. For simplicity we concentrate on the schematic version of the algorithm (page 29) rather than the optimised version of Figure 1.23.

Let G have n nodes. Each stage of the algorithm adds an arc to the subgraph constructed so far. Let the subgraphs constructed at each stage be $T_0, \ldots, T_k, \ldots$. We start with T_0 having just the node start. Let T_{k+1} be got from T_k by adding arc a_{k+1}. Since a new node from the fringe is added at each stage, clearly T_k has $k+1$ nodes, and so there are $n-1$ stages, with T_{n-1} being returned by the algorithm.

We shall show by induction on k that each T_k is a subgraph of an MST T' of G.

Algorithm Prim's MST Algorithm ('classic' version):
Choose any node start as the root
tree[start] = true
for x in adj[start]:
  # add x to fringe
  fringe[x] = true
  parent[x] = start
  weight[x] = $W[\text{start}, x]$
while fringe nonempty:
  Select a fringe node f such that weight[f] is minimum
  fringe[f] = false
  tree[f] = true
  for y in adj[f]:
    if not tree[y]:
      if fringe[y]:
        # update candidate arc
        if $W[f, y] <$ weight[y]:
          weight[y] = $W[f, y]$
          parent[y] = f
      else:
        # y is unseen
        fringe[y] = true
        weight[y] = $W[f, y]$
        parent[y] = f

Figure 1.23: Prim's MST Algorithm

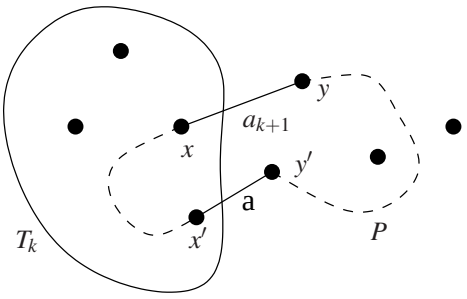


Figure 1.24: Correctness of Prim's algorithm

Base case $k = 0$. T_0 has one node and no arcs. Clearly $T_0 \subseteq T'$ for any MST T' of G.

Induction step. Assume that $T_k \subseteq T'$, some MST T' of G. Let the new arc a_{k+1} join node x of T_k to a new fringe node y not in nodes(T_k).

If $a_{k+1} \in \text{arcs}(T')$ then $T_{k+1} \subseteq T'$ as required. So suppose $a_{k+1} \notin \text{arcs}(T')$. Since T' is a spanning tree, there must be a path P in T' from x to y. So $P \cup \{a_{k+1}\}$ forms a cycle. There must be an arc a in P which joins a node x' of T_k to a fringe node y' not in nodes(T_k). See Figure 1.24. We can form a new spanning tree T'' from T' by removing a and adding a_{k+1}. Since the algorithm chose a_{k+1} rather than a, we have $W(a_{k+1}) \leq W(a)$. Hence $W(T'') \leq W(T')$ and so T'' is an MST (since all MSTs have the same weight, we must actually have $W(a_{k+1}) = W(a)$). Also, $T_{k+1} \subseteq T''$ as required.

Now T_{n-1} has $n-1$ arcs, and $T_{n-1} \subseteq T'$ for some MST T'. Since all spanning trees for G have $n-1$ arcs, we must have $T_{n-1} = T'$. Hence T_{n-1} is an MST, as required.

Remark 1.7.4 *We can regard the induction hypothesis in Theorem 1.7.3 as an* invariant*. This invariant is established initially, and maintained through each execution of the while loop of the code (Figure 1.23).*

Remark 1.7.5 *It is clear that each T_k constructed by Prim's algorithm is connected. Also, though this would need proving, in fact T_k is an MST for the subgraph of G* induced *by* nodes(T_k) *(i.e. the subgraph with nodes* nodes(T_k) *and all arcs of G which join nodes in* nodes(T_k)*). We did not require either of these pieces of information when proving Theorem 1.7.3.*

For a different and sometimes more efficient version of Prim's algorithm we can use *priority queues*. Unlike ordinary FIFO queues, each item x of the queue has a priority key$[x]$—usually a natural number. In a *min* PQ, as considered here, this key represents cost. Items are removed from the queue lowest key first.

The PG abstract data type has the following operations:

- Q = PQcreate(): create an empty PQ
- isEmpty(Q): tests whether empty
- insert(Q,x): add a new item x to the queue. It will have a key value key[x].
- getMin(Q): returns an item with lowest key value
- deleteMin(Q): deletes the item returned by getMin(Q)
- decreaseKey(Q,x,newkey): updates key[x] = newkey

We can use a priority queue to store the candidate arcs in order, by storing the fringe nodes and their associated parents and using the weight of the candidate arc as the key.

It is convenient to also place the unseen nodes in the queue, with key ∞, which can be taken to be any number greater than the weight of any arc.

Algorithm Prim's algorithm with priority queues:

```
Q = PQcreate()
for x in NodesG:
   key[x] = ∞;
parent[x] = nil
   insert(Q,x)
decreaseKey(Q,start,0)
while not isEmpty(Q):
   f = getMin(Q); deleteMin(Q)
   tree[f] = true
   for y in adj[f]:
      if not tree[y]: # so y in Q
         if W[f,y] < key[y]:
            decreaseKey(Q,y,W[f,y])
            parent[y] = f
```

With n nodes and m arcs the number of PQ operations in the above algorithm is:

- insert $O(n)$
- isEmpty $O(n)$
- getMin $O(n)$
- deleteMin $O(n)$
- decreaseKey $O(m)$

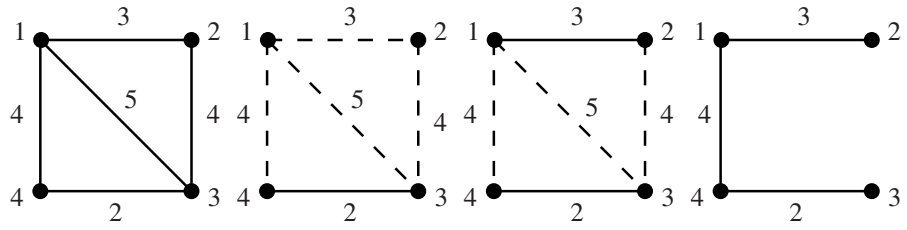


Figure 1.25: MST constructed by Kruskal's algorithm

You will see a good implementation of priority queues using Binary Heaps later in this course. For a PQ of length N all operations are $\log N$, apart from isEmpty and getMin which are $O(1)$.

So Prim with PQ has overall complexity $O(m \log n)$, assuming that $n < m$ as is usually the case.

Which is better—'classic' Prim with candidate arcs or Prim with PQ?

If the graph is sparse, say $m \leq n \log n$, we have $O(m \log n) = O(n \log^2 n)$, which is better than $O(n^2)$. So Prim with PQ is better.

On the other hand, if the graph is dense, with $O(n^2)$ arcs, we have $O(m \log n) = O(n^2 \log n)$, which is worse than $O(n^2)$. So classic Prim is better.

1.7.3 Kruskal's Algorithm

We have seen that Prim's algorithm is "greedy": it always chooses the shortest candidate arc, thereby pursuing short-term advantage. Remarkably, this turned out to give optimal results, as we saw that the algorithm is guaranteed to construct an MST.

An even greedier strategy is the following: At each stage choose the shortest arc not yet included, except when this would give a cycle. This is the basis of Kruskal's MST algorithm. We illustrate on the graph in Figure 1.25. First we choose $(3,4)$ as it has the least weight. Then we choose $(1,2)$. This shows that during the construction process we have a *forest* (i.e. an acyclic graph) rather than a tree, since the graph need not be connected. Finally we choose one of $(1,4)$ or $(2,3)$. The process is finished, since adding any other arc would give a cycle.

Here is the algorithm in outline:

Algorithm Kruskal's MST algorithm—Scheme:
$F = \emptyset$ # forest being constructed
$R = \text{arcs}(G)$ # remaining arcs
while R nonempty:
  remove a of smallest weight from R

if a does not make a cycle when added to F:
  add a to F
return F

Notice that arcs are added in increasing order of weight. So a good implementation strategy could be to work on a list of arcs sorted by weight in increasing order.

Theorem 1.7.6 *Let G be a connected weighted graph. Then Kruskal's algorithm constructs an MST for G.*

Proof. Our strategy is much the same as for Prim's algorithm (Theorem 1.7.3). Let G have n nodes. Each stage of the algorithm adds an arc to the subgraph constructed so far. Let the subgraphs constructed at each stage be $F_0, \ldots, F_k, \ldots$. We start with F_0 empty. Let F_{k+1} be got from F_k by adding arc a_{k+1}. So each F_k has k arcs.

Clearly there should be $n-1$ stages, since any spanning tree must have $n-1$ arcs. But we need to check this, since the algorithm executes the while loop as long as it can add a new arc without creating a cycle. If there is any unused node or F_k is not connected then a further stage is possible. Any connected acyclic graph with n nodes must have $n-1$ arcs (Proposition 1.4.2). Hence there are at least $n-1$ stages. Since we then have a spanning tree we stop at stage $n-1$, with F_{n-1} being returned by the algorithm.

We shall show by induction on k that each F_k is a subgraph of an MST T' of G.

Base case $k = 0$. F_0 is empty. Clearly $F_0 \subseteq T'$ for any MST T' of G.

Induction step. Assume that $F_k \subseteq T'$, some MST T' of G. Let the new arc a_{k+1} join node x to node y. If $a_{k+1} \in \text{arcs}(T')$ then $T_{k+1} \subseteq T'$ as required. So suppose $a_{k+1} \notin \text{arcs}(T')$. Since T' is a spanning tree, there must be a path P in T' from x to y. So $P \cup \{a_{k+1}\}$ forms a cycle. There must be an arc a in P which does not belong to F_k, since otherwise the algorithm could not add a_{k+1} to F_k as it would form a cycle. Suppose that a joins x' to y'. There cannot be a path from x' to y' in F_k or else there would be a cycle in T' (recall that $F_k \subseteq T_k$). Hence a is a candidate for the algorithm to add to F_k. Since the algorithm chose a_{k+1} rather than a, we must have $W(a_{k+1}) \leq W(a)$. We can form a new spanning tree T'' from T' by removing a and adding a_{k+1}. We have $W(T'') \leq W(T')$ and so T'' is an MST (since all MSTs have the same weight, we must actually have $W(a_{k+1}) = W(a)$). Also, $T_{k+1} \subseteq T''$ as required.

Now T_{n-1} has $n-1$ arcs, and $T_{n-1} \subseteq T'$ for some MST T'. Since all spanning trees for G have $n-1$ arcs, we must have $T_{n-1} = T'$. Hence T_{n-1} is an MST, as required.

For the implementation of Kruskal's algorithm, we have to do two things:

1. Look at each arc in ascending order of weight. We can use a priority queue here (or just sort the arcs at the start).

2. Check whether adding the arc to the forest so far creates a cycle. Here we use *dynamic equivalence classes*. Put nodes in the same equivalence class if they belong to the same connected component of the forest constructed so far. Map

each node to the representative of its equivalence class. An arc (x,y) can be added if x and y belong to different equivalence classes. If (x,y) is added, then *merge* the equivalence classes of x and y.

Dynamic equivalence classes can be handled using the *Union-Find* data type. Each set has a *leader* element which is the representative of that set.

- find: find the leader of the equivalence class
- union: merge two classes

Operations:

- sets = *UFcreate*(n) — creates a family of singleton sets $\{1\},\{2\},\ldots,\{n\}$ with find(sets,x) = x
- x' = *find*(sets,x) — finds the leader x' of x within sets
- *union*(sets,x,y) — merge the sets led by x and y
  and use one of x or y as the new leader

  NB x and y must be the leaders of their sets, so that x = find(sets,x) and y = find(sets,y)

So our implementation scheme for Kruskal is as follows:

Algorithm Kruskal implementation scheme:
Let G have n nodes numbered from 1 to n.
Build a priority queue Q of the edges of G with the weights as keys
sets = UFcreate(n) *# initialise Union-Find with singletons* $\{1\},\ldots,\{n\}$
$F = \emptyset$ *# forest being constructed*
while not isEmpty(Q):
  (x,y) = getMin(Q); deleteMin(Q)
  x' = find(sets,x); y' = find(sets,y)
  if $x' \mathrel{!}= y'$: *# no cycle*
    add (x,y) to F
    union(sets,x',y') *# merge the two components*

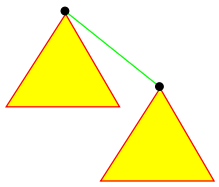
We now consider how to implement union-find. A naive implementation would be as follows:

Maintain an array leader of nodes:

- leader[x] stores the leader of the set to which node x belongs.
- Initially leader[x] = x for all nodes.

Find is now $O(1)$. However union takes $O(n)$. That means that it takes $O(n^2)$ to perform the $O(n)$ unions required for Kruskal.

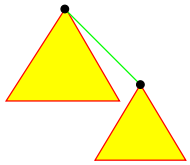
Instead, each set is stored as a (non-binary) tree. The root node is the representative (leader) of the set. We merge two sets by appending one tree to the other, so that the root of one tree is the child of the root of the other tree.



To store the tree structure, for each node x we maintain parent[x], where parent[x] = x if x is the root (the leader). Initially parent[x] = x for any node x. Then union(sets,x,y) just involves setting parent[y] = x (or vice versa) — constant time $O(1)$ However find involves following parent[x] up to the root.

The time taken is bounded by the depth of the tree. With naive merging of trees as above, this can be as much as n, so that find becomes $O(n)$.

We clearly need to keep the depth as low as possible. A possible strategy is to use weighted union: since we can append trees in either order, always append the tree of lower size to the one of greater size.



This requires us to store the size of the tree and update this — easy to do.

Lemma 1.7.7 *Using weighted union, the depth of a tree of size k is $\leq \lfloor \log k \rfloor$.*

Proof. Proved by (strong) induction on k.

For $k = 1$ depth is $0 = \lfloor \log 1 \rfloor$.

Suppose true for all $k' < k$. Let tree T of size k and depth d be got from T_1 of size k_1 and depth d_1 and T_2 of size k_2 and depth d_2.

Suppose that $k_2 \leq k_1$ and T_2 is appended to T_1 to form T. By induction hypothesis, $d_1 \leq \lfloor \log k_1 \rfloor$ and $d_2 \leq \lfloor \log k_2 \rfloor$. Also $d = \max(d_1, d_2 + 1)$.

Now $k_1 \leq k$. So $d_1 \leq \lfloor \log k \rfloor$. Also $k_2 \leq k/2$. So $d_2 \leq \lfloor \log k/2 \rfloor = \lfloor \log k \rfloor - 1$.

Combining: $d \leq \lfloor \log k \rfloor$ as required.

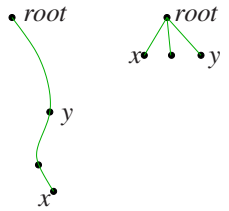


Figure 1.26: Path Compression.

Using the Lemma, with weighted union, each find takes $O(\log n)$, and each union takes $O(1)$.

For Kruskal, there will be:

- $O(m)$ inserts to build the PQ — time taken $O(m \log m)$
- $O(m)$ getMins and $O(m)$ deleteMins — time taken $O(m \log m)$
- $O(m)$ finds — time taken $O(m \log n)$
- $O(n)$ unions — time taken $O(n)$

So overall time taken is $O(m \log m)$ (assuming $m \geq n$, as is normally the case).

The number of arcs m is bounded by n^2. So $O(m \log m) = O(m \log n)$. Hence the overall complexity for Kruskal is $O(m \log n)$. This is the same as Prim with priority queues.

Remark 1.7.8 *In fact we can build the PQ in time $O(m)$ rather than $O(m \log m)$. But this does not bring down the overall complexity here.*

We can improve union-find using *path compression*. The complexity for the union-find part of Kruskal then reduces to

$$O((n+m) \log^* n)$$

Here $\log^* n$ is an extremely slow-growing function. Indeed, $\log^* n \leq 5$ for any conceivable n that might be used.

Path compression works as follows: when finding the root (leader) for a node x, if this is not parent[x] then make parent[y] = root for all y on the path from x to the root. See Figure 1.26.

We have extra work in updating parent, but we keep the depth of the nodes lower so that future finds are faster.

This procedure combines well with weighted union (on size) since size is unchanged by path compression.

Normal find:

```
Algorithm proc find(x)::
  y = parent[x]
  if y == x:     # x is the root
    root = x
  else:
    root = find(y)
  return root
```

We can modify to perform the path compression:

```
Algorithm proc cfind(x)::
  y = parent[x]
  if y == x:     # x is the root
    root = x
  else:
    root = cfind(y)
    if root != y:
      parent[x] = root
  return root
```

Comparing the various algorithms for finding MSTs:

Kruskal: $O(m \log n)$

Prim with PQ (binary heap): $O(m \log n)$

Classic Prim: $O(n^2)$

Which is better? As when comparing classic Prim and Prim with PQ:

- On dense graphs where m is large (order n^2) then Kruskal gives $O(n^2 \log n)$ and classic Prim is to be preferred.
- On sparse graphs, where m is small (say $O(n \log n)$), then Kruskal (or Prim with PQ) give better results than classic Prim:

$$O(m \log n) = O(n \log^2 n)$$

Priority queues can also be implemented with *Fibonacci heaps* rather than binary heaps. All operations are $O(1)$ apart from deleteMin, which is $O(\log n)$.

Complexity of Prim with PQ (Fibonacci heap):

$$O(m + n \log n)$$

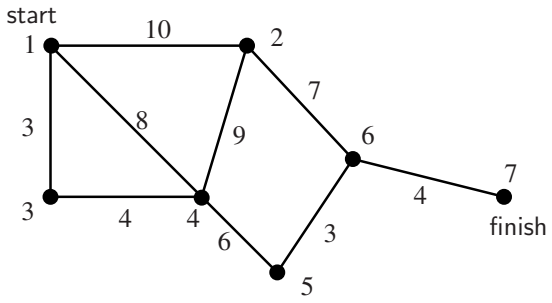


Figure 1.27: The shortest path problem

In practice the memory usage and constant factors can be high.

1.7.4 The Shortest Path Problem

Suppose we have a network of cities connected by roads of varying lengths and we wish to find the shortest route between two cities.

We can represent this by a weighted graph where the arcs are labelled with distances. In Figure 1.27 there are seven cities (labelled from 1 to 7). The problem might be to find the shortest path from 1 to 7. The path 1,2,6,7 has length 10+7+4 = 21. A shorter path is 1,3,4,5,6,7 which has length 20.

THE SHORTEST PATH PROBLEM: Given a weighted graph (G,W), and two nodes start and finish, find the shortest path from start to finish in the graph.

Contrast with

THE ALL PAIRS SHORTEST PATH PROBLEM: Given a weighted graph G, find the shortest paths between all pairs of nodes of G.

We shall see an algorithm due to Floyd later in these lecture notes. This is an excellent algorithm for finding all shortest paths between any pairs of nodes. It runs in $O(n^3)$ time. Suppose instead that we only want to find the shortest distance from a start node to a finish node. Floyd's algorithm requires us to compute all shortest paths simultaneously. There seems to be no way of speeding up the algorithm if we are only interested in particular i, j. However we can compute the shortest path between two particular nodes in time $O(n^2)$ using an algorithm due to Dijkstra which is very closely related to Prim's MST algorithm.

Call the two nodes start and finish. As in Prim's algorithm, we build up a spanning tree starting from the start node. We classify nodes into

- *tree* nodes: already included

- *fringe* nodes: not in the tree yet, but adjacent to a tree node
- *unseen* nodes: the rest

The new idea in the shortest path algorithm is that we have already computed the shortest path from start to all the tree nodes: it is the path given by the tree. As far as the fringe nodes are concerned, we know the shortest path using the tree constructed so far. This path might be improved as the tree grows. We know nothing about the shortest path to an unseen node.

We store two values for each tree or fringe node: its parent node in the tree, and the length of the shortest path known. At each stage the next node to be added is the fringe node with the smallest length. We can obtain the shortest path to a node x in reverse order from the parent function:

$$x, \text{parent}[x], \text{parent}[\text{parent}[x]], \ldots, \mathsf{start}$$

Let path(x) denote this path in the forwards direction:

$$\mathsf{start}, \ldots, \text{parent}[\text{parent}[x]], \text{parent}[x], x$$

This is defined for all tree or fringe nodes.

Dijkstra's algorithm is stated in Figure 1.28. It should be compared with Prim's algorithm (Figure 1.23). Notice that the algorithm does not need to complete computing the entire spanning tree, since it can stop as soon as finish joins the tree. When the algorithm terminates, the length of the shortest path is distance[finish] and we can read off the path through the tree using the parent function.

The result of running the algorithm on the example of Figure 1.27 is shown in Figure 1.29. In this case we got a spanning tree for the entire graph, since the finish node was the last to be added to the tree. It is not a *minimum* spanning tree.

Remark 1.7.9 *The algorithm will work perfectly well for directed graphs.*

The running time of Dijkstra's algorithm is $O(n^2)$ by much the same analysis as for Prim's algorithm.

It is easy to see that the algorithm terminates, since we clearly increase the tree each time we execute the while loop. To see why the algorithm is correct we need to formulate an invariant. It has three conditions:

1. If x is a tree or fringe node (other than start) then parent[x] is a tree node.
2. If x is a tree node (other than start) then distance[x] is the length of the shortest path, and parent[x] is its predecessor along that path.
3. The fringe nodes are precisely those non-tree nodes f reachable from start by a path in the tree (except for the final arc to f). If f is a fringe node then distance[f] is the length of the shortest path *where all nodes except f are tree nodes*. Furthermore, parent[f] is its predecessor along that path.

Algorithm Dijkstra's Shortest Path Algorithm:
Input: Weighted graph (G,W) together with a pair of nodes start, finish
Output: Length of shortest path from start to finish

```
tree[start] = true
for x in adj[start]:
   # add x to fringe
   fringe[x] = true
   parent[x] = start
   distance[x] = W[start,x]
while not tree[finish] and fringe nonempty:
   Select a fringe node f such that distance[f] is minimum
   fringe[f] = false
   tree[f] = true
   for y in adj[f]:
      if not tree[y]:
         if fringe[y]:
            # update distance and candidate arc
            if distance[f] + W[f,y] < distance[y]:
               distance[y] = distance[f] + W[f,y]
               parent[y] = f
         else:
            # y is unseen
            fringe[y] = true
            distance[y] = distance[f] + W[f,y]
            parent[y] = f
return distance[finish]
```

Figure 1.28: Dijkstra's Shortest Path Algorithm

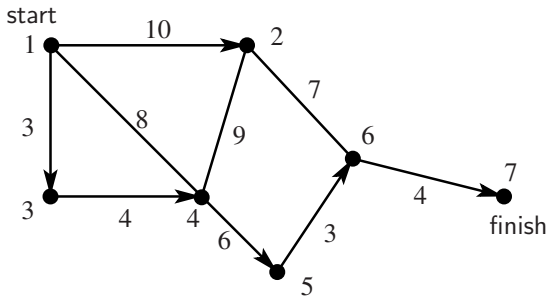


Figure 1.29: Solution to the shortest path problem

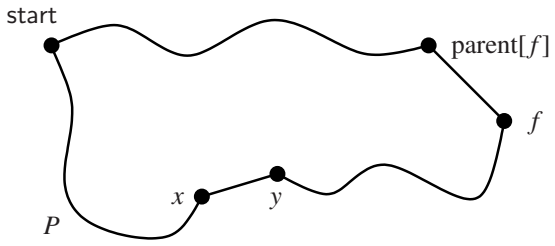
When the program terminates, finish is a tree node, and by (2) we then have the required shortest path.

It is easy to see that the initialisation establishes the invariant. It is also easy to see that (1) is always true. It guarantees that when we find a shortest path it will lie wholly inside the tree. The main work is in seeing that parts (2) and (3) of the invariant are maintained by the while loop.

To see that (2) is maintained when f is added to the tree, we need to check that distance[f] is the length of the shortest path, and that parent[f] is the predecessor along that path. Let path(f) be the path given by the algorithm:

$$\mathsf{start}, \ldots, \text{parent}[f], f$$

Suppose we have a different and shorter path P (not necessarily in the tree). Let x be the last node on P to belong to the tree, and let y be the next node along P (possibly f itself). In the diagram, path(f) is along the top and P is along the bottom:



Now by (3) we know that distance[y] $\leq$ the distance from start to y using P. Hence the length of $P \geq$ distance[y]. But distance[y] $\geq$ distance[f] by our choice of f. Hence P is at least as long as path(f).

We leave the proof that (3) is maintained as a (not so easy) exercise.

Hence we have established:

Theorem 1.7.10 *If (G,W) is a connected weighted graph with nodes* start *and* finish*, then Dijkstra's algorithm finds the shortest path from* start *to* finish.

Much as for Prim's algorithm, we can implement Dijkstra's algorithm using priority queues:

Algorithm Dijkstra's algorithm with priority queues:
Q = PQcreate()
for x in Nodes$G$:
  key[x] = ∞;
parent[x] = nil
  insert(Q,x)

```
decreaseKey(Q, start, 0)
while not tree[finish] and not isEmpty(Q):
   f = getMin(Q); deleteMin(Q)
   tree[f] = true
   for y in adj[f]:
      if not tree[y]: # so y in Q
         if key[f]+W[f,y] < key[y]:
            decreaseKey(Q, y, key[f] + W[f,y])
            parent[y] = f
```

The analysis is very much the same as for Prim's algorithm.

- Dijkstra with PQ (binary heap) is overall $O(m \log n)$ assuming that $n < m$ as is usually the case.
- Dijkstra with PQ (Fibonacci heap) is overall $O(m + n \log n)$.

A* algorithm

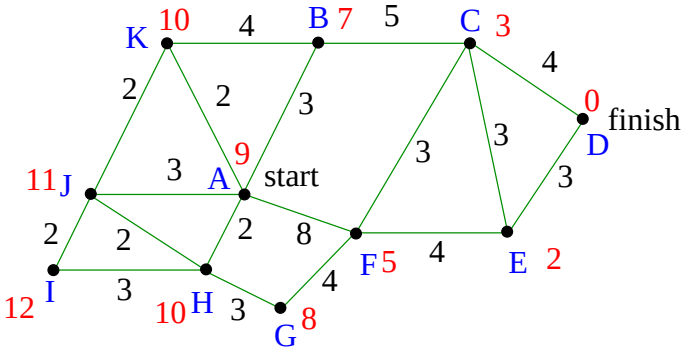
We now consider a different algorithm for the single pair shortest path problem.

The A* algorithm is due to Hart, Nilsson and Raphael (1968).

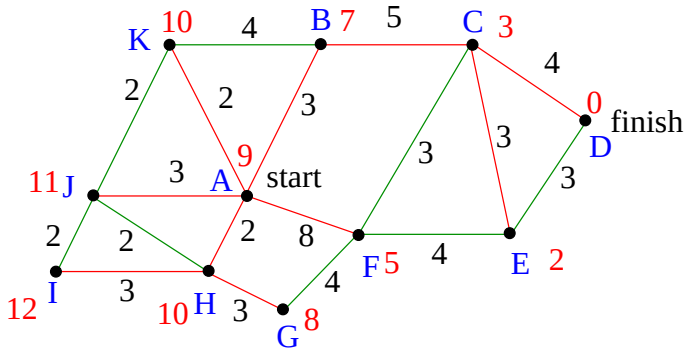
We assume that we have a heuristic function $h(x)$ which *underestimates* the distance from any node x to the finish node.

If we are dealing with cities on a map, h could be the Euclidean distance (as the crow flies).

Example 1.7.11 *Each node x has heuristic value $h(x)$ shown in red.*



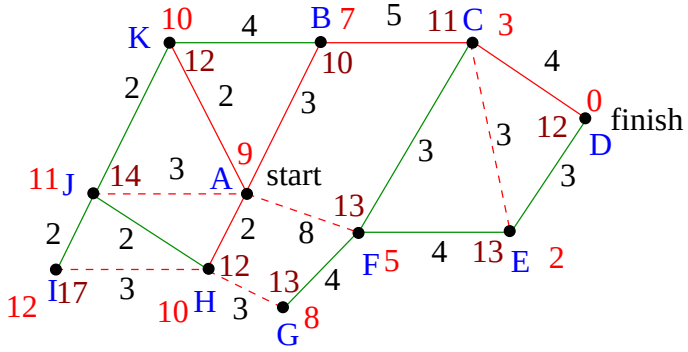
Using Dijkstra's algorithm we have to compute the entire shortest path tree before reaching node D.



We had to find the shortest paths to all the nodes which are closer to start *than D.*

Of course we did not use the heuristic information.

With A:*



A heuristic function is *consistent* if

1. for any adjacent nodes x, y we have $h(x) \leq W(x,y) + h(y)$
2. $h(\mathsf{finish}) = 0$

This is clearly satisfied by the Euclidean distance heuristic function if $W(x,y)$ represents distances on a map.

Proposition 1.7.12 *Let h be a consistent heuristic function, and let P be a path from node x to node y. Then $h(x) \leq len(P) + h(y)$.*

A heuristic function is *admissible* if for any node x we have $h(x) \leq$ the weight of the shortest path from x to the goal finish.

It is easy to check that if h is consistent then h is admissible (exercise).

Algorithm A* algorithm:
Input: Weighted graph (G,W) together with a pair of nodes start, finish
and consistent heuristic function h
Output: Length of shortest path from start to finish
tree[start] = true
g[start] $= 0$
f[start] $= g$[start] $+ h$[start]
for x in adj[start]:
  # add x to fringe
  fringe[x] = true
  parent[x] = start
  $g[x] = W[\text{start}, x]$
  $f[x] = g[x] + h[x]$
while finish not a tree node and fringe non-empty:
  Select a fringe node x s.t. $f[x]$ is minimum
  fringe[x] = false
  tree[x] = true
  for y in adj[x]:
    if not tree[y]:
      if fringe[y]: # update $g(y)$, $f(y)$ and candidate arc
        if $g[x] + W[x,y] < g[y]$:
          $g[y] = g[x] + W[x,y]$
          $f[y] = g[y] + h[y]$
          parent[y] = x
      else: # y is unseen
        fringe[y] = true
        $g[y] = g[x] + W[x,y]$
        $f[y] = g[y] + h[y]$
        parent[y] = x
return g[finish]

Remark 1.7.13
1. *The set of tree nodes is often called the* closed set *and the set of fringe nodes is the* open set.
2. *If we set $h(x) = 0$ for all nodes x then h is consistent, and the A* algorithm is just Dijkstra's algorithm.*
3. *We deduce that the running time for A* is the same as for Dijkstra in the worst case, though we hope to do better on average, depending on h.*
4. *We have presented A* for* consistent *heuristics, for simplicity and for its closeness to Dijkstra's algorithm*
5. *There is a more general version of A* which is guaranteed to give the correct solution for* admissible *heuristics.*
   The difference is that we may have to re-examine nodes that are already in the closed set (the tree).

We now show that A* correctly computes the shortest path.

The algorithm terminates, since we clearly increase the tree each time we execute the while loop.

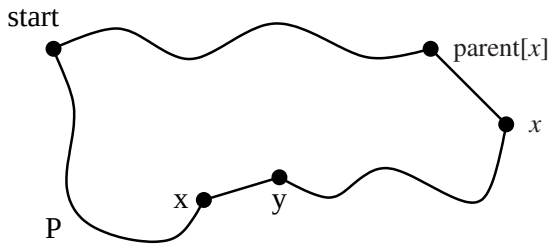
To see why the algorithm is correct we need to formulate an invariant, which is essentially the same as for Dijkstra's algorithm.

Invariant:

1. If x is a tree or fringe node (other than start) then parent[x] is a tree node.

2. If x is a tree node (other than start) then $g[x]$ is the length of shortest path, and parent[x] is its predecessor along that path.

3. If x is a fringe node then $g[x]$ is the length of the shortest path where all nodes except x are tree nodes. Furthermore, parent[x] is its predecessor along that path.



When the program terminates, finish is a tree node, and by (2) we then have the required shortest path.

So it remains to show that the invariant is

- established before the while loop
- maintained during the while loop

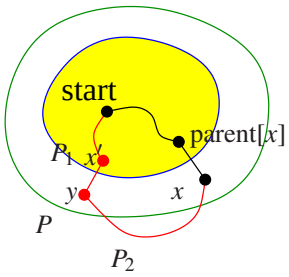
We just show that (2) is maintained, and omit the rest of the proof.

Suppose x is added to the tree.

We need to check that we have found the shortest path.

The path given by the algorithm: start, ..., parent[x], x has length $g[x]$.

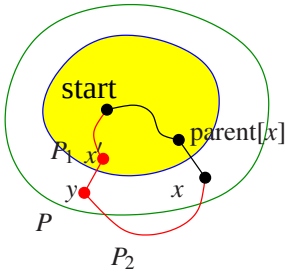
Suppose we have a different and shorter path P
(not necessarily in the tree).



Then $\text{len}(P) < g[x]$.

Let y be the first node on P not to belong to the tree.

Let P_1 be P from start to y, and P_2 be P from y to x.



$$\begin{array}{rcll} f[y] & = & g[y]+h[y] & \\ & \le & g[y]+\text{len}(P_2)+h[x] & \text{consistency of } h \\ & \le & \text{len}(P_1)+\text{len}(P_2)+h[x] & \text{by (3) for } y \\ & = & \text{len}(P)+h[x] & \\ & < & g[x]+h[x] & \text{assumption} \\ & = & f[x] & \end{array}$$

But $f[x] \le f[y]$ by our choice of x. Contradiction.

Algorithm A* algorithm with priority queues:
Q = PQcreate()
for x in nodes(G):
  $g[x] = \infty$; key[x] = ∞; parent[x] = nil
  insert(Q, x)
g[start] $= 0$; decreaseKey(Q, start, g[start] $+ h$[start])
while not tree[finish] and not isEmpty(Q):
  x = getMin(Q); deleteMin(Q)
  tree[x] = true
  for y in adj[x]:
    if not tree[y]: # so y in Q
      if $g[x] + W[x,y] < g[y]$:

$g[y] = g[x] + W[x,y]$; decreaseKey$(Q, y, g[y] + h[y])$
parent[y] = x

Warshall's Algorithm

We briefly review the concept of the *transitive closure* of a binary realtion.

Definition 1.7.14 *Let $R \subseteq X^2$ be a binary relation.*

1. *We say that R is* transitive *if for all $x,y,z \in X$, if $R(x,y)$ and $R(y,z)$ then $R(x,z)$ holds.*
2. *The* transitive closure *of R, denoted R^+, is the smallest relation $S \subseteq X^2$ such that $R \subseteq S$ and S is transitive.*

Note in particular that if R is transitive then $R^+ = R$.

An equivalent definition is that R^+ is the intersection of all transitive relations on X which include R as a subset. It is easy to check that the intersection of a family of transitive relations is also transitive.

An alternative characterisation of R being transitive is that $R \circ R \subseteq R$ where $\circ$ is relational composition.

Let R^k be $R \circ \cdots \circ R$ (k times). Then we can show that

$$R^+ = \bigcup_{k=1}^{\infty} R^k$$

If X is finite and $|X| = n$ then

$$R^+ = \bigcup_{k=1}^{n} R^k$$

We can interpret R as a directed graph G. The nodes of G are just the members of X, and there is an arc from x to y iff $R(x,y)$. Note that there are no parallel arcs; we could have loops where $R(x,x)$.

It is easy to see that $R^k(x,y)$ iff there is a path of length k from x to y. So $R^+(x,y)$ iff there is a path of length ≥ 1 from x to y.

Suppose $X = \{1,\ldots,n\}$. Clearly if we set

$$A[i,j] = \begin{cases} 1 & \text{if } R(i,j) \\ 0 & \text{otherwise} \end{cases}$$

then A is the adjacency matrix of G.

We can compute R^k using matrix multiplication:

$$R^k(i,j) \text{ iff } A^k[i,j] > 0$$

Let

$$B = \sum_{k=1}^{n} A^k$$

Then

$$R^+(i,j) \text{ iff } B[i,j] > 0$$

So far we have been building up the transitive closure by looking for paths of length k for $k = 1,\ldots,n$. We now look at a quite different and more efficient method.

Suppose that the nodes are $\{1,\ldots,n\}$. Consider a path $p = x_1,x_2,\ldots,x_k$ from x_1 to x_k. We say that nodes $x_2,\ldots,x_{k-1}$ are *intermediate* nodes of p.

We look for paths which use nodes $\leq k$ as intermediate nodes.

Let $B_k[i,j] = 1$ iff there is a path from i to j which uses intermediate nodes $\leq k$ (set $B_k[i,j] = 0$ otherwise).

Clearly $B_0[i,j] = A[i,j]$ since we only have paths of length one, as there can be no intermediate nodes ≤ 0.

Also $R^+(i,j)$ iff $B_n[i,j] = 1$, since B_n allows all possible intermediate nodes, and so all possible paths.

Now we just need to calculate B_k from B_{k-1} (for $k = 1,\ldots,n$).

Suppose we have a path p from i to j using intermediate nodes $\leq k$.

There are two cases:

1. k is not an intermediate node of p. Then $B_{k-1}[i,j]$ already.
2. k is an intermediate node of p.

   We can assume that k occurs only once, since if it occurs multiple times we can shorten the path by removing the cycle(s) from k to k.

   But then we have paths i to k and k to j which just use intermediate nodes $\leq k-1$. So $B_{k-1}[i,k]$ and $B_{k-1}[k,j]$.

This is the idea behind Warshall's algorithm.

Algorithm Warshall's algorithm:
input A
copy A into B (array of Booleans) # $B = B_0$
for k = 1 to n:
  # $B = B_{k-1}$
  for i = 1 to n:

```
      for j = 1 to n:
         b_ij = b_ij or (b_ik and b_kj)
   # B = B_k
# B = B_n
return B
```

The complexity is clearly $O(n^3)$.

To show correctness of the algorithm we formulate an invariant

$$B = B_{k-1}$$

We can recast the outer for loop as a while loop and show that this invariant is established initially and then maintained by each iteration of the while loop.

Algorithm Warshall's algorithm with while loop:

```
copy A into B      # B = B_0
k = 1      # B = B_{k-1}
while k ≤ n:
   # B = B_{k-1}
   for i = 1 to n:
      for j = 1 to n:
         b_ij = b_ij or (b_ik and b_kj)
   k = k + 1
# k = n + 1, B = B_n
return B
```

Floyd's Algorithm

We return to the All Pairs Shortest Path Problem.

THE ALL PAIRS SHORTEST PATH PROBLEM: given a weighted *directed* graph G, find the shortest paths between all pairs of nodes of G.

This can be solved efficiently using a simple modification of Warshall's algorithm.

Let G be a weighted *directed* graph with nodes $\{1,\ldots,n\}$ and adjacency matrix A.

Let $B_k[i,j]$ be the length of the shortest path from i to j *which uses intermediate nodes* $\leq k$.
If there is no such path set $B_k[i,j] = \infty$.

Clearly $B_0[i,j] = \begin{cases} A[i,j] & \text{if } A[i,j] \\ \infty & \text{otherwise} \end{cases}$

Also $B_n[i,j]$, will be the length of the shortest path from i to j.

Now we just need to calculate B_k from B_{k-1} (for $k = 1,\ldots,n$). Suppose we have a shortest path p from i to j using intermediate nodes $\leq k$ of length d.

There are two cases:

1. k is not an intermediate node of p. Then $B_{k-1}[i,j] = d$ already.
2. k is an intermediate node of p.

   Clearly k occurs only once, since p is shortest path.

   But then we have paths i to k and k to j which just use intermediate nodes $\leq k-1$.

   These must be shortest paths just using intermediate nodes $\leq k-1$ (or else p could be shorter).

   So $d = B_{k-1}[i,k] + B_{k-1}[k,j]$.

We see that $B_k[i,j] = \min(B_{k-1}[i,j], B_{k-1}[i,k] + B_{k-1}[k,j])$.

(This also works if there is no shortest path just using nodes $\leq k$.)

Algorithm Floyd's algorithm:
input A

set $B[i,j] = \begin{cases} 0 & \text{if } i = j \\ A[i,j] & \text{if } i \neq j \text{ and there is an arc } (i,j) \\ \infty & \text{otherwise} \end{cases}$

\# $B = B_0$
for k = 1 to n:
  for i = 1 to n:
    for j = 1 to n:
      $b_{ij} = \min(b_{ij}, b_{ik} + b_{kj})$
return B

The complexity is clearly $O(n^3)$ as for Warshall's algorithm.

Dynamic Programming

Warshall's algorithm and Floyd's algorithm are both examples of *dynamic programming*.

In dynamic programming:

- break the main problem down into sub-problems
- the sub-problems are ordered (e.g. increasing size) and culminate in the main problem

To solve the main problem:

- move through the sub-problems in order

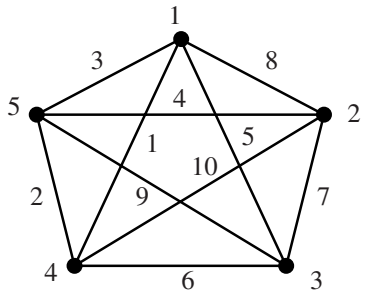


Figure 1.30: The travelling salesman problem

- solve each sub-problem using the stored solutions of the previous sub-problems and storing the new solution for later use
- solve the main problem as the final sub-problem

1.7.5 The Travelling Salesman Problem

Given some cities and roads between them, a travelling salesman wants to find a tour of all the cities with the shortest distance.

THE TRAVELLING SALESMAN PROBLEM (TSP): Given a *complete* weighted graph (G,W), find a way to tour the graph visiting each node exactly once and travelling the shortest possible distance.

An example with five cities is shown in Figure 1.30.

The restriction to complete graphs is not quite as strong as it might seem, since if arcs were missing from the graph we could make it complete by adding fictitious arcs with weights made high enough to ensure that they would never be chosen.

TSP is clearly related to both the Hamiltonian Circuit Problem (HCP) (Section 1.3.3) and the Shortest Path Problem (Section 1.7.4). We have to find a Hamiltonian circuit (HC) which is of minimum weight. The difficulty is not in finding an HC, since we have assumed that the graph is complete. But to find the shortest HC involves potentially checking $n!$ different tours if G has n nodes.

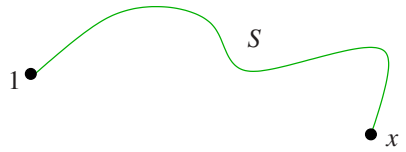
Just like HCP, TSP is NP-complete and so unlikely to have a polynomial solution. A better algorithm than just checking every possible tour yields a running time of $O(n^2 2^n)$, which is still worse than exponential. This is the Bellman-Held-Karp algorithm, which is another example of dynamic programming.

Let (G,W) have Nodes $= \{1,\ldots,n\}$.

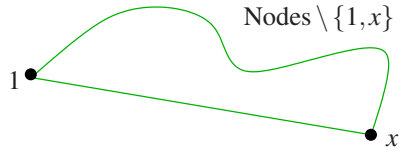
Idea: Fix a start node, say 1.

For each $x \neq 1$ and each $S \subseteq \text{Nodes} \setminus \{1,x\}$:

find and store the minimum cost $C(S,x)$ of a path from node 1 to node x using set of intermediate nodes precisely S.



A TS tour can start at 1 wlog (without loss of generality). Let the last node before returning to 1 be x. The least cost of such a tour is $C(\text{Nodes} \setminus \{1,x\},x) + W(x,1)$.



So the solution to TSP is

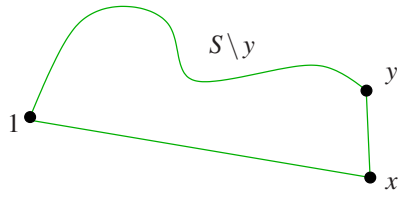
$$\min_{x \neq 1} C(\text{Nodes} \setminus \{1,x\},x) + W(x,1)$$

We calculate $C(S,x)$ in increasing order of size: do all S of size 0, then 1, up to $n-2$.

Clearly $C(\emptyset,x) = W(1,x)$ as no intermediate nodes are allowed.

Assume we know $C(S,x)$ for all S of size k.
Suppose $|S| = k+1$.

Consider the last intermediate node y in a least cost path from 1 to x using intermediate nodes S (including y).



The cost must be $C(S \setminus y, y) + W(y,x)$. So

$$C(S,x) = \min_{y \in S} C(S \setminus y, y) + W(y,x)$$

Here is the algorithm in pseudocode:

Algorithm Bellman-Held-Karp algorithm:
Input (G,W)
Choose $\mathsf{start} \in \text{Nodes}G$
for $x \in \text{Nodes} \setminus \{\mathsf{start}\}$:
  $C[\emptyset,x] = W[\mathsf{start},x]$ # Process sets S in increasing order of size.
for $S \subseteq \text{Nodes} \setminus \{\mathsf{start}\}$ with $S \neq \emptyset$:
  for $x \in \text{Nodes} \setminus (S \cup \{\mathsf{start}\})$:
    # Find $C[S,x]$
    $C[S,x] = \infty$
    for $y \in S$:
      $C[S,x] = \min(C[S \setminus \{y\},y] + W[y,x], C[S,x])$
# Now have calculated and stored all values of $C[S,x]$
opt = ∞
for $x \in \text{Nodes} \setminus \{\mathsf{start}\}$:
  opt = $\min(C[\text{Nodes} \setminus \{\mathsf{start},x\},x] + W[x,\mathsf{start}]$, opt)
return opt

We analyse the time complexity as follows: For each subset of Nodes (roughly speaking) we do $O(n^2)$ work with the two for loops. So overall this gives $O(n^2 2^n)$.

Remark 1.7.15 *The Bellman-Held-Karp algorithm can be adapted to solve the Hamiltonian circuit problem. The complexity is still $O(n^2 2^n)$.*

Despite exact solutions taking too long, we still want to solve TSP—there are applications in circuit design, etc. So we are led to consider approximate methods. For instance, we could try a "greedy" algorithm which always chooses the shortest available arc, the so-called *nearest neighbour heuristic (NNH)*.

On the example of Figure 1.30, starting from node 1 we get the tour $1,4,5,2,3,1$ with a total weight of $1+2+4+7+5 = 19$ which is quite good. However such a shortsighted method can also fail dramatically. Suppose that we change the weight of $(1,3)$ from 5 to 500. Then clearly this arc should be avoided, but NNH is forced to choose it on the last step.

Chapter 2

Algorithm Analysis

2.1 Introduction

In Part 2 of the course we look at how to measure the efficiency of various algorithms, some of which will have been developed in 141 Reasoning about Programs.

Suppose we have a problem P and S is the set of all possible algorithms which solve P. For instance the problem might be to sort a list, or to find the shortest path between two nodes in a graph. We would like to know which of the available algorithms is best. This will mean ranking the members of S in some way. In these lectures we shall rank algorithms according to how fast they run—their time complexity, in the jargon. There are other methods of ranking which may be just as valid, depending on the circumstances, but there are certainly obvious reasons for preferring algorithm A to another algorithm A' which takes twice as long as A.

Apart from discovering which of the available algorithms is best, we would also like to know the answer to a more fundamental question: can we improve our algorithms and solve P faster? Is there a best algorithm, and have we found it, or should we go on looking? This seems like a very difficult question, since there may be many algorithms in S which have not been invented, so that endless improvement might be possible.

The solution to this difficulty is to reason about the problem P itself, and the steps that an algorithm must take if it is to solve P. For instance, an algorithm to find the greatest element of an unordered list must inspect every element of the list, or else it could not be correct. Reasoning of this kind gives us a minimum amount of work which any algorithm must do if it is to solve P. We are putting a lower bound on the time complexity of all members of S. If we have already found a member A of S which does no more than this amount of work, then we can safely assert that A is optimal. No amount of ingenuity can improve A.

We shall concentrate on the problems of searching and sorting. This is for two reasons:

- Since searching and sorting are carried out so frequently, it is especially important that they are carried out as efficiently as possible.
- The theory is well worked out and optimal algorithms are known.

Terminology 2.1.1 *When we talk about searching through a list L or sorting it, we assume random access to L. The kth element of L is denoted $L[k]$ (starting to count from 0). So we are really regarding a list as an array.*

2.2 Searching an unordered list

We start by considering a very simple problem, and use it to introduce some of the key concepts, such as worst-case and average-case analysis.

Problem 2.2.1 (Searching an unordered list) *Given a list L of length $n \geq 1$ over a data type D and a value $x \in D$, if x is in L return k ($1 \leq k \leq n$) such that $L[k] = x$, else return "not found". If x occurs several times, any k such that $L[k] = x$ will do.*

Here is a particular algorithm which solves the problem:

Algorithm Linear Search (LS):
$k = 0$
while $k < n$:
  if $L[k] == x$:
    return k
  else:
    $k = k + 1$
return "not found"

Clearly LS finds the *least* k such that $L[k] = x$, if x is in L.

We would like to measure how long LS takes. A perfectly sensible measure would be how many times the while loop is entered. However we want to compare LS to other algorithms, which might not use while loops. So instead it is convenient to measure the number of comparisons of the form $L[k] = x$, since this will generalise to other algorithms.

It is reasonable to expect the time an algorithm takes to depend on the size of the input. In this case a reasonable measure of input size is the length of L. But now consider the following examples. Take $n = 4$ and D to be the natural numbers and x to be 5. Then LS makes 1 comparison on input $[5,4,3,6]$ and 4 comparisons on $[3,4,6,2]$. In general with input size n, LS makes anything from 1 to n comparisons.

There are two standard solutions to this problem—*worst-case* and *average-case* analysis. We describe them for LS, but clearly the definitions can be made in general for any algorithm.

- Worst-case analysis: Let $W(n)$ be the largest number of comparisons made when performing LS on the whole range of inputs of size n. In this case $W(n) = n$.

- Average-case analysis: Let $A(n)$ be the average number of comparisons made when performing LS on the whole range of inputs of size n.

It might seem obvious that average-case analysis is to be preferred. However before we can say that it gives a better measure, we need to know what the probability distribution of inputs is. If we don't know this, we have to make assumptions, such as that x occurs with equal probability in all positions in the list. Furthermore, worst-case analysis has two great advantages:

1. It offers us a guarantee that LS will never take longer than $W(n)$.

2. $W(n)$ is easier to compute than $A(n)$, and often yields almost as good results. When we discuss orders of complexity we shall see that $W(n)$ and $A(n)$ are often of the same order.

For these reasons it is usual to start with worst-case analysis.

Remark 2.2.2 *One might think that best-case analysis $B(n)$ could be worthwhile. However it is easy to fake good results. Suppose that an algorithm is very poor in general, but happens to do well when x occurs in the first position. Then perhaps $B(n) = 1$, but this is highly misleading. By judicious tampering the most inferior algorithm can be tuned up to look good on hand-picked inputs.*

Terminology 2.2.3 *We shall refer to functions from* **N** *to* **N** *like $W(n)$ and $A(n)$ as* complexity functions.

Now LS can be varied (e.g. we could search L backwards, or in some other order), but it is hard to see how it can be improved. In fact LS is optimal. The argument is as follows. Take any algorithm A which solves the problem. We claim that if A returns "not found" it must have inspected every entry of L. Suppose for a contradiction that A did not inspect $L[k]$ (some $k < n$). Define a new list L' as follows. L' is the same as L except that $L'[k] = x$. Since A did not inspect $L[k]$ it will not inspect $L'[k]$ either, since the two lists are otherwise identical. So A must still return "not found". However A has now given the wrong answer. Contradiction. We conclude that in worst case A must inspect every entry, meaning that n comparisons are needed. So LS is optimal.

Notice how the argument was about *all possible algorithms* to solve the problem, not just those which have been so far devised. Of course proofs of optimality are not always so easy, and may require a lot of ingenuity. There are many problems, such as matrix multiplication, for which it is not known whether the best algorithm so far discovered is in fact optimal.

2.3 Searching an ordered list

Problem 2.3.1 (Searching an ordered list) *Given an ordered list L of length $n \geq 1$ over a ordered data type $(D, \leq)$ and a value $x \in D$, if x is in L return k ($0 \leq k < n$) such that $L[k] = x$, else return "not found".*

This problem is clearly no harder than the previous one, so that LS will clearly solve it. However the fact that L is ordered means that we can stop once we find a value in the list which exceeds x:

Algorithm Modified Linear Search (MLS):
$k = 0$
while $k < n$:
  cond:
    $L[k] = x$: return k
    $L[k] > x$: return "not found"
    $L[k] < x$: $k = k + 1$
return "not found"

We use an obvious notation for the three-way conditional, which we regard as performing a single comparison. We have made an improvement, in that we can return "not found" before inspecting the entire list. This does not show up in the worst-case analysis, since $W(n)$ is still n, as we may have to inspect the whole list. It would show up in the average-case analysis. However we do not bother with this, as there is a different algorithm which improves MLS drastically, namely binary search (BS).

Notation 2.3.2 *Let x be a real number. The* floor *of x, denoted $\lfloor x \rfloor$, is the greatest integer $\leq x$ (if $x \geq 0$ this is the integer part of x). The* ceiling *of x, denoted $\lceil x \rceil$, is the least integer $\geq x$.*

Algorithm Binary Search (BS):
procedure BinSearch(left, right):
  # searches for x in L in the range L[left] to L[right]
  if left $>$ right:
    return "not found"
  else:
    mid $:= \lfloor$(left + right)$/2 \rfloor$
  cond
    $x = L$[mid]: return mid
    $x < L$[mid]: BinSearch(left, mid $-$ 1)
    $x > L$[mid]: BinSearch(mid + 1, right)

A convenient representation of the algorithm is by means of decision trees. A node in a decision tree represents a comparison, and its children represent the next step in

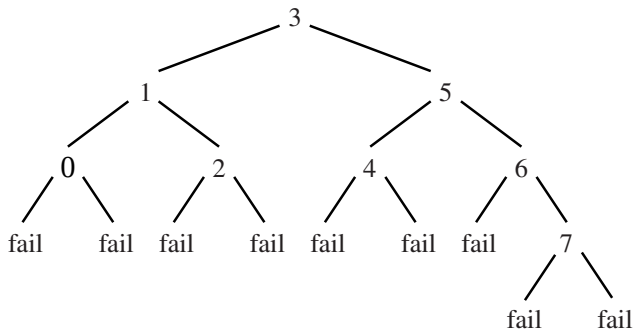


Figure 2.1: Decision tree for Binary Search ($n = 8$)

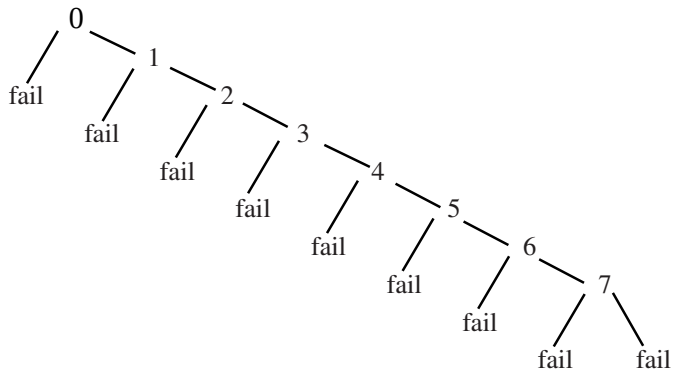


Figure 2.2: Decision tree for Modified Linear Search ($n = 8$)

the algorithm depending on the result of the comparison. We illustrate for $n = 8$ in Figure 2.1. The left-hand child of a node k corresponds to $x < L[k]$. The right hand child corresponds to $x > L[k]$. If $x = L[k]$ then the algorithm of course stops straight away. The tree shows us for instance that if $L[4] = x$ and $L[3] < x$, $L[5] > x$, then BS takes three comparisons to return the answer 4. It is immediate that the worst-case number of comparisons is 4 (when $x > L[6]$, so that $L[3]$, $L[5]$, $L[6]$ and $L[7]$ are inspected). So $W(8) = 4$.

We can draw a similar decision tree for every n, giving a complete description of how BS performs on any input of size n, and allowing us to find $W(n)$. Notice that the tree will contain n nodes, one for each of $0, \ldots, n-1$ (as well as nodes for fail).

As an illustration of how BS is better than MLS, Figure 2.2 shows a decision tree for MLS on $n = 8$. It has the same number of nodes, but greater depth, corresponding to higher worst-case complexity. BS has succeeded in reducing depth by "widening" the

tree. BS is in fact optimal, and the reason is that it does the best possible job of keeping the tree-depth low for a given number of nodes.

Before showing that BS is optimal, we calculate $W(n)$. Since BS is recursive, we cannot find $W(n)$ directly. However we can write down the following recurrence relation:

$$\begin{aligned} W(1) &= 1 \\ W(n) &= 1 + W(\lfloor n/2 \rfloor) \end{aligned}$$

Either we find $L[\text{mid}] = x$ immediately, or else we call the procedure on a portion of the array which has half the length. We solve for $W(n)$ by repeated expansion:

$$\begin{aligned} W(n) &= 1 + W(\lfloor n/2 \rfloor) \\ &= 1 + 1 + W(\lfloor n/4 \rfloor) \\ &\cdots \\ &= 1 + 1 + \cdots + 1 + W(1) \end{aligned}$$

The number of 1s is the number of times that we can divide n by 2. Suppose that $2^k \leq n < 2^{k+1}$. We can always find such a k. Then the number of 1s in the summation is k. If we denote the logarithm to base 2 of n by $\log n$ then $k = \lfloor \log n \rfloor$. Therefore (since $W(1) = 1$)

$$W(n) = \lfloor \log n \rfloor + 1$$

As a check, note that $\log 8 = 3$, so that $W(8) = 4$, which is the value we found from the decision tree.

Consider any algorithm A for searching an ordered list of length n. We can represent it by a binary decision tree. The tree will have at least n nodes since the algorithm must have the capability to inspect every member of the list (by an argument similar to that for unordered lists). Let the depth of the tree be d (not counting fail nodes, since fail is not a comparison). The worst-case performance of A will be $d+1$. We shall show that d has a minimum value.

Terminology 2.3.3 *A* binary tree *is a tree where each node has at most two children. Denote the depth of a tree T by* $\text{depth}(T)$*, the number of nodes in T by* $\#\text{nodes}(T)$*, and the root node of T by* $\text{root}(T)$.

Lemma 2.3.4 *Suppose that T is a binary tree with depth d. Then T has no more than $2^{d+1} - 1$ nodes.*

Proof. By induction.

Base case $d = 0$. Then T has at most 1 node (the root), and $1 \leq 2^{0+1} - 1$.

Induction step. Assume true for all numbers $\leq d$ and show for $d+1$. Let T be a tree of depth $d+1$. Suppose $\text{root}(T)$ has children T_1, T_2. Then $\text{depth}(T_1) \leq d$, $\text{depth}(T_2) \leq d$. So by induction T_1 and T_2 each have no more than $2^{d+1} - 1$ nodes. But

$$\#\text{nodes}(T) = 1 + \#\text{nodes}(T_1) + \#\text{nodes}(T_2)$$

Hence #nodes$(T) \leq 1+2(2^{d+1}-1) = 2^{(d+1)+1}-1$.

If root(T) has only one child then the argument is similar but easier.

From Lemma 2.3.4 we see that if T has n nodes and depth d then $n \leq 2^{d+1}-1$. So $d+1 \geq \lceil \log(n+1) \rceil$.

Lemma 2.3.5 $\lceil \log(n+1) \rceil = \lfloor \log n \rfloor + 1$.

Proof. Let $k = \lfloor \log n \rfloor$. Then $2^k \leq n < 2^{k+1}$. So $2^k < n+1 \leq 2^{k+1}$. Hence $k < \log(n+1) \leq k+1$ and the result follows.

Using Lemma2.3.5 we see that the worst-case behaviour of A (namely $d+1$) is at least $\lfloor \log n \rfloor + 1$. This gives us:

LOWER BOUND FOR SEARCHING. Any algorithm for searching an ordered list of length n for element x, and which only accesses the list by comparing x with entries in the list, must perform at least $\lfloor \log n \rfloor + 1$ comparisons in worst case.

But $W(n)$ for BS is $\lfloor \log n \rfloor + 1$. We conclude that BS is optimal.

2.4 Orders of complexity

Suppose that the worst-case complexity of some algorithm is $8n^2+300n+70$. In other words for input size n, it takes at most $8n^2+300n+70$ steps. The most important part of the polynomial is clearly the $8n^2$ term, since as n gets large, the $300n+70$ will make comparatively little difference. If we are interested in crude comparisons between algorithms, the constant factor 8 can also often be ignored, and we say that the algorithm is order n^2. A different algorithm with complexity say $2n^3+n2+4$ (order n^3) is clearly going to be slower for large n, even though it has smaller constant factors (and therefore may actually run faster for small n).

Another reason for ignoring constants is that we can get different constants depending on how we calculate the number of steps. As an example, suppose we wish to calculate the number of steps required to multiply two $n \times n$ matrices together. Each entry in the product takes n multiplications and $n-1$ additions by the obvious method. Since there are n^2 entries to be calculated, this gives a total of $n^2(2n-1) = 2n^3-n^2$ calculations. But it may be reasonable to regard multiplication as more costly than addition. In this case we might simply count the n^3 multiplications, or we might say that a multiplication counts as 10 additions. But when constants are ignored the order is n^3 by any of these methods.

So using orders give us a way of comparing complexity as n grows large. It also allows us to ignore less important details.

The various possible orders of polynomials are 1 (constant), n (linear), n^2 (quadratic), n^3, ... Above this come the exponentials 2^n, 3^n, ... We are also interested in logarithmic orders. Recall that linear search is order n (linear), while binary search is order $\log n$.

This gives us an expanded list of orders

$$1 \quad \log n \quad n \quad n\log n \quad n^2 \quad \dots$$

An algorithm which is order $n\log n$ is said to be *log linear*. We now give the precise definitions.

Definition 2.4.1 *Let* $\mathbf{R}^+$ *be the real numbers* ≥ 0. *Let* $f, g : \mathbf{N} \to \mathbf{R}^+$.

1. f *is* $O(g)$ *("*f *is big Oh of* g*") iff there are* $m \in \mathbf{N}$, $c \in \mathbf{R}^+$ *such that for all* $n \geq m$ *we have* $f(n) \leq c.g(n)$.
2. f *is* $\Theta(g)$ *("*f *is order* g*") iff* f *is* $O(g)$ *and* g *is* $O(f)$.

We may read "f is $O(g)$" as saying that the order of $f \leq$ the order of g. "f is $\Theta(g)$" then says that f has the same order as g.

Example 2.4.2 $8n^2 + 300n + 70$ *is* $\Theta(n^2)$.

Proof: First show $8n^2 + 300n + 70$ is $O(n^2)$. Take $c = 16$. We need to find m such that for all $n \geq m$, $8n^2 + 300n + 70 \leq 16n^2$, i.e. $300n + 70 \leq 8n^2$. $m = 40$ will do, or if we wished we could take $m = 106$, or any other large number. Also clearly n^2 is $O(8n^2 + 300n + 70)$. Hence $8n^2 + 300n + 70$ is $\Theta(n^2)$.

In practice the official Θ notation is seldom used, and "f is $O(n^3)$" is often pronounced as "f is order n^3", and really means the stronger statement "f is $\Theta(n^3)$". But of course there are times when the true complexity of the problem is unknown. For instance we have just seen that there is a matrix multiplication algorithm which is $\Theta(n^3)$. But there might be (and indeed there is) a faster algorithm. So without further analysis, the most we can say about the problem of matrix multiplication is that it is $O(n^3)$. Its true order might be as low as n^2 (the best lower bound currently known).

Over the years, aymptotically better algorithms have been obtained:

- Strassen 1969: $O(n^{\log 7}) = O(n^{2.807})$
- Coppersmith-Winograd 1987: $O(n^{2.376})$
- Stothers 2010: $O(n^{2.373})$
- Williams 2012: $O(n^{2.3729})$
- Le Gall 2014: $O(n^{2.3728639})$
- Alman, Williams 2021: $O(n^{2.3728596})$
- Daun, Wu, Zhou 2022: $O(n^{2.37188})$

2.5 Strassen's Algorithm

We now describe Strassen's Algorithm for matrix multiplication. Assume we are multiplying two $n \times n$ matrices:

$$AB = C$$

Start with $n = 2$.

Then

$$C = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

This takes 8 multiplications (and 4 additions).

Strassen's insight was that we can do the $n = 2$ case in only 7 multiplications (and 18 additions).

$$C = \begin{pmatrix} x_1 + x_4 - x_5 + x_7 & x_3 + x_5 \\ x_2 + x_4 & x_1 + x_3 - x_2 + x_6 \end{pmatrix}$$

where

$$\begin{array}{ll} x_1 = (a_{11} + a_{22}) * (b_{11} + b_{22}) & x_5 = (a_{11} + a_{12}) * b_{22} \\ x_2 = (a_{21} + a_{22}) * b_{11} & x_6 = (a_{21} - a_{11}) * (b_{11} + b_{12}) \\ x_3 = a_{11} * (b_{12} - b_{22}) & x_7 = (a_{12} - a_{22}) * (b_{21} + b_{22}) \\ x_4 = a_{22} * (b_{21} - b_{11}) & \end{array}$$

Note that commutativity of multiplication is not used. Hence we can generalise to matrices.

Suppose that $n = 2^k$. Divide up the matrices into four quadrants, each $n/2 \times n/2$:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Compute C_{ij} using the formulas for c_{ij}, and recursively compute each multiplication by further subdivision until bottom out at $n = 2$. We need to add/subtract 18 matrices of dimension $n/2 \times n/2$, and recursively perform 7 multiplications of $n/2 \times n/2$ matrices.

The number of arithmetic operations $A(k)$ for $n = 2^k$ is given by the recurrence relation:

$$\begin{aligned} A(0) &= 1 \\ A(k) &= 7A(k-1) + 18(n/2)^2 \end{aligned}$$

We can solve this by repeated expansion and then summing the resulting geometric progression. The solution is:

$$A(k) = 7 \cdot 7^k - 6 \cdot 4^k = 7 \cdot 7^{\log n} - 6n^2 = 7 \cdot n^{\log 7} - 6n^2 \approx 7 \cdot n^{2.807} - 6n^2$$

So Strassen's matrix multiplication algorithm is $\Theta(n^{2.807})$.

So far we have required $n = 2^k$. If $n \neq 2^k$, we can add an extra dummy row and column to keep the dimension even to allow subdivision.

Strassen's algorithm is an example of a *divide and conquer* algorithm.

- *Divide* problem into a subproblems of size n/b (here $a = 7$ and $b = 2$):
  May take work to set up the subproblems (here matrix addition).
- Solve each subproblem recursively.
- Then *combine* to get the result.
  Again, this may take work (here matrix addition).

We shall see further examples: MergeSort, QuickSort.

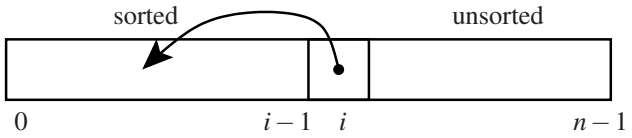
2.6 Sorting

Problem 2.6.1 *Sort a list L of length n over a totally ordered data type D.*

We look at algorithms which use comparisons, i.e. take any two members of the list and see which is below the other. There are other methods which can be employed if D is of certain special kinds. We wish to discover what are the most efficient sorting algorithms, and whether there are optimal algorithms. Just as with searching, we will do two things:

- measure the time complexity of known algorithms
- obtain a lower bound for the amount of work required.

We start by considering a simple sorting algorithm and measuring its worst-case complexity. Insertion Sort works as follows. Suppose that $L[0..i-1]$ is known to be sorted (where $1 \leq i < n$). Then insert $L[i]$ into $L[0..i-1]$ in its correct position. When this is done, $L[0..i]$ will be sorted.



Algorithm Insertion Sort:
$i = 1$
while $i < n$:
  # insert $L[i]$ into $L[0..i-1]$
  $j = i$
  while $L[j-1] > L[j]$ and $j > 0$:
    Swap$(j-1, j)$
    # swaps $L[j-1]$ and $L[j]$. $L[j]$ will always be the value to be inserted
    $j = j-1$
  $i = i+1$

How many comparisons does Insertion Sort take? The insertion can be done by comparing $L[i]$ successively with $L[i-1]$, $L[i-2]$, ... until we find the first $L[j]$ such that $L[i] \geq L[j]$. Once this is found $L[i]$ should be inserted between $L[j]$ and $L[j+1]$. In the best case, $L[i] \geq L[i-1]$ (so that $L[0..i]$ is already sorted). Only one comparison is needed. However in worst case we need to perform i comparisons. This will happen if $L[i] < L[0]$. Therefore in worst case we need to perform i comparisons for $i = 1, 2, \ldots, n-1$. So for Insertion Sort:

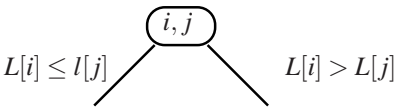
$$W(n) = \sum_{i=1}^{n-1} i \ = \ \frac{n(n-1)}{2}$$

The worst case arises when sorting a list which is in reverse order. So we have a sorting algorithm which is $\Theta(n^2)$. We shall see that there are faster algorithms. First we establish a lower bound for sorting.

2.6.1 Lower Bound for Sorting by Comparison

Terminology 2.6.2 *Say that a node of a tree is a* leaf *if it has no successor nodes. Otherwise it is* internal.

Any algorithm for sorting by comparison can be expressed as a decision tree. A decision tree for sorting a list L of length n is a binary tree with the following properties. Every internal node of the tree consists of a pair i, j where $i, j < n$. This represents comparing $L[i]$ with $L[j]$. The left- hand path is followed if $L[i] \leq L[j]$ and the right-hand path if $L[i] > L[j]$.



Every leaf node represents a possible outcome expressed as a permutation of $[0, 1, ..., n-1]$. For example if $n = 4$ a possible leaf will be $[1, 0, 3, 2]$. This means that the original list can be sorted by placing $L[0]$, $L[1]$, $L[2]$, $L[3]$ in the order $L[1]$, $L[0]$, $L[3]$, $L[2]$. Thus, once a permutation is given the list is in effect sorted. Of course the actual algorithm may well rearrange L during the stage when the comparisons are carried out.

Example 2.6.3 *The algorithm* 3*-sort for sorting a list of length* 3 *is described by the decision tree in Figure2.3. We note the following:*

- *The worst-case number of comparisons is* 3*, and the best is* 2*.*
- *There are* 6 *leaf nodes, one for each of the* 6 *possible permutations of* $[0, 1, 2]$.

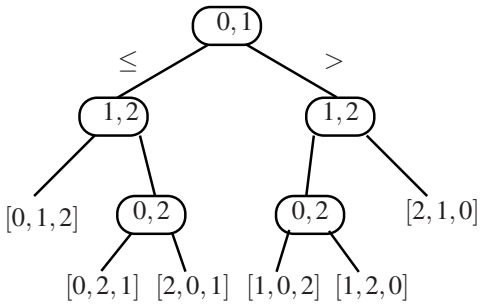


Figure 2.3: Decision tree for 3-sort

Now we argue that any sorting algorithm for $n = 3$ must do at least 3 comparisons in worst case: First note that the algorithm should have every possible permutation of $[0,1,2]$ as a leaf in its decision tree. If it did not have a certain permutation, then it could not correctly deal with a list which was jumbled in that particular way. So its tree must have at least 6 leaves. Now the worst-case number of comparisons is equal to the depth of the tree. If a binary tree has depth 2 then it can have no more than 4 leaves. So the tree must have depth at least 3, and the algorithm must do at least 3 comparisons in worst case.

Therefore 3-sort is optimal.

Now we generalise what we have just done to obtain a lower bound for all values of n.

Proposition 2.6.4 *If a binary tree has depth d then it has $\leq 2^d$ leaves.*

Proof. By induction on d.

Base case $d = 0$. Then no more than $1 = 2^0$ leaves.

Induction step. Assume true for all trees of depth $\leq d$. Suppose the tree has depth $d+1$. The root has one or two successors, which have depth $\leq d$. By the inductive hypothesis, the successors have at most 2^d leaves each. So the total number of leaves in the tree $\leq 2^d + 2^d = 2^{d+1}$.

Consider a decision tree for sorting a list of length n. It must have at least one leaf for every permutation of $[0,...,n-1]$. Now there are $n!$ such permutations. So the tree needs at least $n!$ leaves. Now the worst-case performance of the algorithm is exactly the depth d of the tree. By the proposition, d must satisfy $n! \leq 2^d$. Hence $d \geq \log(n!)$. Since d is an integer we can improve this to $d \geq \lceil \log(n!) \rceil$. Hence:

LOWER BOUND FOR SORTING IN WORST CASE. Any algorithm for sorting a list of length n by comparisons must perform at least $\lceil \log(n!) \rceil$ comparisons in worst case.

We now give a table showing the lower bounds for $n = 1$ to 10 compared with the

worst-case performance of Insertion Sort:

n	1	2	3	4	5	6	7	8	9	10
$\lceil \log(n!) \rceil$	0	1	3	5	7	10	13	16	19	22
$n(n-1)/2$	0	1	3	6	10	15	21	28	36	45

Although Insertion Sort is optimal for $n \leq 3$, it seems that there is a big difference as n gets larger. This means that there is room for improvement, either by finding a faster sorting algorithm, or possibly by finding a better lower bound argument. We shall see how the algorithm Mergesort improves on Insertion Sort.

However first we consider the lower bound for average-case analysis. One might expect this to be significantly less than the worst-case bound, but we shall find that it is almost the same.

Let us return to the example 3-sort. There are 6 possible rearrangements of a list of length 3. From the tree (Figure 2.3) we see that $[0,1,2]$ and $[2,1,0]$ can be sorted in 2 comparisons, while the other four take 3 comparisons. It is reasonable to assume that each of the 6 is equally likely. Therefore the average number of comparisons is

$$(2+2+3+3+3+3)/6 = 2\,2/3$$

which is scarcely less than the worst-case number 3.

We must now generalise to arbitrary decision trees. Call the sum of all the depths of the leaf nodes in a tree its *total path length*. This is closely related to the average number of comparisons. Suppose that the tree has $n!$ leaves and total path length b. Then the average number of comparisons is $b/n!$. So for a given number of leaves ($n!$) we want to find a lower bound on the total path length.

Now notice that the tree for 3-sort has depth 3, and all its leaves have depth 3 or 2.

Definition 2.6.5 *A binary tree of depth d is* balanced *if every leaf node is of depth d or $d-1$.*

Proposition 2.6.6 *If a binary tree is unbalanced then we can find a balanced tree with the same number of leaves without increasing the total path length.*

Proof. Suppose a tree of depth d is unbalanced. Then it has a leaf of depth d, and a leaf of depth $d' \leq d-2$. In this case we can construct another tree with the same number of leaves, which is "more balanced" in a sense that we do not make precise, and where the total path length has not increased.

Take a node of depth d. There are two cases:

1. If this node has no sibling (i.e. it is the only child of its parent) then remove it. An example is shown on the left-hand side of Figure 2.4. The parent becomes a leaf node of depth $d-1$. The total path length has reduced by 1. The depth of the tree may or may not have been reduced. The number of leaves is unchanged. The tree is "more balanced".

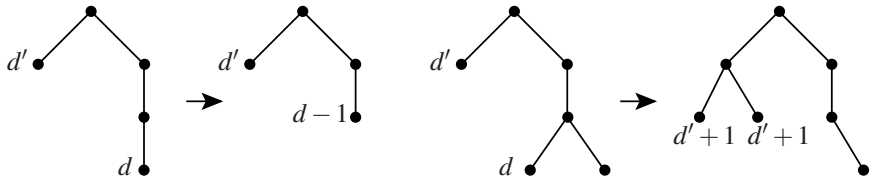


Figure 2.4: Balancing binary trees

2. If the node of depth d has no sibling, remove it and suspend two new leaf nodes below the leaf of depth d'. An example is shown on the right-hand side of Figure 2.4. The old two leaves contributed $d+d'$ to the total path length, while their replacements contribute $2(d'+1)$. So the total path length has not increased (and will have been reduced if $d' < d-2$). The number of leaves is unchanged. Again the tree is "more balanced".

We repeat the above procedure until we can no longer find a pair of nodes of depth d and $d' \leq d-2$. The tree is now balanced. The number of leaves is unchanged and the path length has not increased (and may have reduced).

In view of the proposition, if we are trying to find the minimum total path length of a tree with $n!$ leaf nodes, we can restrict our attention to balanced trees. But in this case the average depth of a leaf node must be only slightly less than the depth of the tree. In fact if the tree has depth d, then the average leaf depth must be between $d-1$ and d.

We conclude that the lower bound for average-case is almost the same as for worst-case.

LOWER BOUND FOR SORTING IN AVERAGE CASE. Any algorithm for sorting a list of length n by comparisons must perform at least $\lfloor \log(n!) \rfloor$ comparisons in average case.

Remark 2.6.7 *We have not mentioned the possibility that the list to be sorted will contain repeats. This will of course reduce the number of possible different permutations. However it does not alter the work we have done above in any essential way.*

2.6.2 Mergesort

We would now like to see how various well-known algorithms match up against the lower bounds we have calculated. In this section we consider Mergesort. Mergesort splits the list into two roughly equal halves, and sorts each of these recursively before merging the two sorted sublists. When sorting a list L of length n, the following procedure will be called initially with left $= 0$, right $= n-1$.

Algorithm Mergesort:

```
procedure Mergesort(left, right):
if left < right:
   mid = ⌊(left + right)/2⌋
   # left ≤ mid < right
   Mergesort(left, mid)
   Mergesort(mid + 1, right)
   Merge(left, mid, right)
```

The procedure Merge(left, mid, right) merges the two sorted sublists L[left..mid] and L[mid + 1..right] to form a new sorted list, and puts the result in the space occupied by those two lists. It works by writing the merged list into an auxiliary array and then copying it back into L when the merging is over. We require left $\leq$ mid $<$ right as a precondition for calling Merge.

We now see how many comparisons Merge performs. Of course this depends on how Merge works, but we shall assume that it follows the following obvious procedure: Generate the merged list in ascending order by repeatedly removing the current least value from the two lists to be merged. Since the lists to be merged are sorted, it is enough to compare the leftmost elements of each list until one of the lists is exhausted, after which the remainder of the other list is transferred automatically. We see that each element is placed in the merged list at the expense of at most one comparison, apart from the last element, which can always be transferred automatically. So the number of comparisons is one less than the length of the merged list, i.e. right $-$ left. It can be shown that this form of merge is optimal in worst case.

We now obtain the following recurrence relation for the worst-case behaviour of Mergesort. The "$n-1$" is for the number of comparisons to perform the merge.

$$\begin{aligned} W(1) &= 0 \\ W(n) &= n-1+W(\lceil n/2\rceil)+W(\lfloor n/2\rfloor) \end{aligned}$$

This allows us to calculate $W(n)$ for $n = 1,\ldots,10$, and to compare this with the lower bounds we obtained earlier.

n	1	2	3	4	5	6	7	8	9	10
$\lceil \log(n!)\rceil$	0	1	3	5	7	10	13	16	19	22
$W(n)$	0	1	3	5	8	11	14	17	21	25

It is apparent that there is a fairly tight, though not exact fit. In particular, we can be certain that Mergesort is optimal for $n \leq 4$. We would like to discover whether Mergesort continues to be close to the lower bound as n gets large. We therefore solve the recurrence relation for $W(n)$ by repeated expansion. To simplify matters we assume $n = 2^k$ ($k \geq 0$).

$$\begin{aligned} W(n) &= n-1+2W(n/2) \\ &= (n-1)+2(n/2-1)+2^2W(n/2^2) \\ &= n+n-(1+2)+2^2W(n/2^2) \\ &\cdots \\ &= n+n+\cdots+n-(1+2+2^2+\ldots+2^{k-1})+2^kW(n/2^k) \end{aligned}$$

Now there are k ns in the sum. Also $1+2+\ldots+2^{k-1} = 2^k - 1 = n-1$. Hence

$$W(n) = k.n - (n-1) + 0\ .$$

Now $k = \log n$ and so we have:

$$W(n) = n\log(n) - n + 1$$

This is of course for n a power of 2. It can be shown that for general n the solution is

$$W(n) = n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1$$

We now wish to compare $W(n)$ for Mergesort with the lower bound we obtained earlier. On the face of it there should be little connection, since $n\log(n) - n + 1$ and $\log(n!)$ seem quite different. However the two functions are of the same order. Notice that

$$\log(n!) = \sum_{i=1}^{n} \log i$$

Moreover this is roughly the area under the curve $y = \log x$ between 1 and n, so that

$$\sum_{i=1}^{n} \log i \approx \int_1^n \log x\ dx$$

(as n gets large). Now $\log x$ is a constant multiple of $\ln x$ (logarithm to base e), and

$$\int_1^n \ln x\ dx = [x\ln(x) - x]_1^n = n\ln(n) - n + 1$$

We conclude that Mergesort is of the same order as the lower bound we obtained.

2.6.3 Master Theorem

Here are three recurrence relations we have already obtained.

Worst case comparisons for Binary Search:

$$W(1) = 1$$
$$W(n) = W(n/2) + 1$$

Worst case comparisons for MergeSort:

$$W(1) = 0$$
$$W(n) = 2W(n/2) + (n-1)$$

Number of arithmetic operations for Strassen's Algorithm:

$$A(1) = 1$$
$$A(n) = 7A(n/2) + 18(n/2)^2$$

All three algorithms are examples of Divide and Conquer algorithms.

The general form for Divide and Conquer:

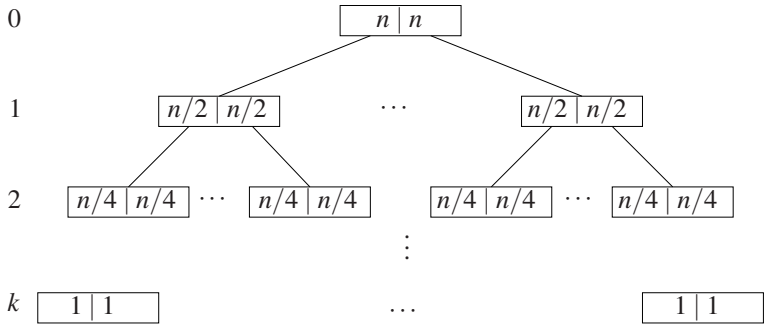


Figure 2.5: Recursion tree.

- Work $T(n)$ for input size n
- Split into a sub-problems of size n/b.
- Non-recursive work (split and combine) is $f(n)$.

The recurrence relation for $T(n)$ is:

$$T(n) = aT(n/b) + f(n)$$

(plus base cases)

We shall show how to solve such a recurrence relation up to Θ. It is helpful to associate *recursion trees* with our recurrence relations. As an example, consider:

$$T(1) = 1$$
$$T(n) = aT(n/2) + n$$

(this is like MergeSort if $a = 2$)

Create a *recursion tree* as follows:

- Start with input of size n;
- Each unfolding of the recursion takes us down one level to a subproblems each with size halved;
- Each node of the tree records the size and the non-recursive work done.

Suppose $n = 2^k$. See Figure 2.5 for the recursion tree.

The total work done is the sum of the work at each of the $k+1$ levels.

$$n + a(n/2) + a^2(n/2^2) + \cdots + a^{k-1}n/(2^{k-1}) + a^k$$
$$= n + (a/2)n + (a/2)^2 n + \cdots + (a/2)^{k-1}n + a^k$$

This is a geometric series.

Proposition 2.6.8

$$\sum_{i=o}^{k} ar^i = \frac{a(r^{k+1}-1)}{r-1}$$

provided $r \neq 1$.

Clearly the sum is $(k+1)a$ if $r = 1$.

Corollary 2.6.9 *Let $t(n)$ be the largest term in the geometric progression*

$$a, ar, ar^2, \ldots, ar^k$$

where r is non-negative, $r \neq 1$ and r does not depend on n (though a and k can depend on n). Then

$$\sum_{i=o}^{k} ar^i = \Theta(t(n))$$

Clearly $t(n)$ is either a (if $r < 1$) or ar^k (if $r > 1$).

We can apply the Corollary to

$$n + (a/2)n + (a/2)^2 n + \cdots + (a/2)^{k-1} n + a^k$$

Here $r = a/2$.

There are three cases depending on a.

- If $a < 2$ the greatest term is n.

  $$T(n) = \Theta(n)$$

  Here the non-recursive work at level 0 dominates.

- If $a = 2$ then

  $$T(n) = (k+1)n = \Theta(n \log n)$$

  Here the work is (roughly) evenly spread at all levels. (cf. MergeSort)

- If $a > 2$ the greatest term is $a^k = a^{\log n} = n^{\log a}$.

  $$T(n) = \Theta(n^{\log a})$$

  Here the base cases (the leaves of the recursion tree) dominate.

We now work towards the general case. Consider

$$T(n) = aT(n/b) + f(n)$$

(plus base cases). The recursion tree will have $1 + \log_b n$ levels. See Figure 2.6.

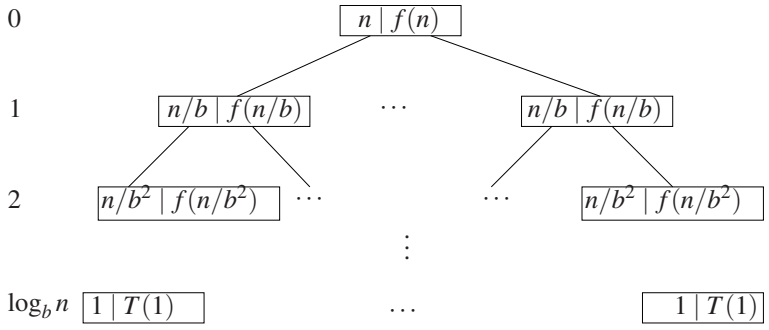


Figure 2.6: Recursion tree.

Level 0: work $f(n)$
Level 1: work $af(n/b)$
Level 2: work $a^2 f(n/b^2)$
$\vdots$
Level $\log_b n$: work $\Theta(a^{\log_b n})$

For the bottom level we just need to know the number of leaves, as the work for each is constant.

Suppose that $f(n) = n^c$. Then the ratio is $r = a/b^c$. Let the *critical exponent* be

$$E = \log_b a = \log a / \log b\,.$$

Then $r > 1$ iff $a > b^c$ iff $\log_b a > c$ iff $E > c$. There are three cases:

- $E < c$:
$$T(n) = \Theta(f(n))$$
- $E = c$:
$$T(n) = \Theta(f(n)\log_b n) = \Theta(f(n)\log n)$$
- $E > c$:
$$T(n) = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a}) = \Theta(n^E)$$

Theorem 2.6.10 *(Master Theorem)*

$$T(n) = aT(n/b) + f(n)$$

has solutions as follows, where $E = \log a / \log b$ is the critical exponent:

1. *If $n^{E+\varepsilon} = O(f(n))$ for some $\varepsilon > 0$ then $T(n) = \Theta(f(n))$.*
2. *If $f(n) = \Theta(n^E)$ then $T(n) = \Theta(f(n)\log n)$.*

3. *If* $f(n) = O(n^{E-\varepsilon})$ *for some* $\varepsilon > 0$ *then* $T(n) = \Theta(n^E)$.

We look at some examples, starting with the worst case comparisons for Binary Search:

$$W(n) = W(n/2) + 1$$

Here $a = 1$ and $b = 2$ and $f(n) = \Theta(n^0)$.

Then $E = \log a / \log b = 0$. So

$$W(n) = \Theta(n^0 \log n) = \Theta(\log n)$$

Next, the worst case comparisons for MergeSort:

$$W(n) = 2W(n/2) + (n-1)$$

Here $a = 2$ and $b = 2$ and $f(n) = \Theta(n^1)$.

Then $E = \log a / \log b = 1$. So

$$W(n) = \Theta(n \log n)$$

Finally, the number of arithmetic operations for Strassen's Algorithm:

$$A(n) = 7A(n/2) + 18(n/2)^2$$

Here $a = 7$ and $b = 2$, $f(n) = \Theta(n^2)$.

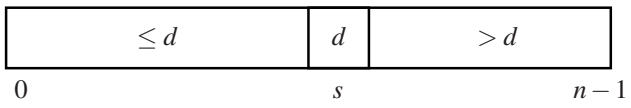
Then $E = \log a / \log b = \log 7 > 2$. So

$$A(n) = \Theta(n^{\log 7})$$

Note that any improvement to $f(n)$ here will not help with the order of $A(n)$.

2.6.4 Quicksort

Quicksort works by "splitting" the list L around a member of L, which we shall take to be $L[0] = d$. This involves comparing every $L[k]$ ($i \neq 0$) with d, and placing it before or after d depending on whether $L[k] \leq d$ or $L[k] > d$, as in the following diagram (s is the new index of the element d):



The portions before and after d are then sorted recursively, after which the whole list is sorted.

Algorithm Quicksort:
procedure Quicksort(left, right):
if left $<$ right:
  s = Split(left, right)
  # left $\leq s \leq$ right
  # $L[s]$ is now in correct position
  Quicksort(left, $s-1$)
  Quicksort($s+1$, right)

Split(left, right) splits L[left..right] around $d = L$[left], and returns the index s of the splitpoint. The result is a reordering of L[left..right] with left $\leq s \leq$ right and

- left $\leq k \leq s$ implies $L[k] \leq d$
- $s < k \leq$ right implies $L[k] > d$

Algorithm Split(left,right):
pre-condition: left $<$ right
$d = L$[left] # pivot
$i =$ left $+ 1$
$j =$ right
# Invariant:
# left $< i \leq j+1$
# $j \leq$ right
# if left $\leq k < i$ then $L[k] \leq d$
# if $j < k \leq$ right then $L[k] > d$
while $i \leq j$:
  if $L[i] \leq d$:
    $i = i+1$
  else:
    Swap(i, j)
    $j = j-1$
# $i = j+1$
Swap(left, j)
return j

It should be clear that applying Split to a list of length n takes $n-1$ comparisons.

Quicksort is harder to analyse than Mergesort because the list can be split into sublists of varying lengths. The split might be roughly in the middle, or it might be near to one end. In order to minimise the number of times that Quicksort is called recursively, it is desirable for the split to occur at the midpoint of the list. The worst case is when

the split occurs at one end. This gives us the following recurrence relation for the worst-case:

$$\begin{aligned} W(1) &= 0 \\ W(n) &= n-1+W(n-1) \end{aligned}$$

The solution to this is

$$W(n) = \sum_{i=0}^{n-1} i \;=\; \frac{n(n-1)}{2}\;.$$

This is no better than Insertion Sort. Ironically the worst case arises in particular when L is already sorted.

However Quicksort performs well in practice. The reason is that the split is unlikely to be near one of the ends, and if it is near the middle then we get performance similar to that of Mergesort. Earlier we said that average-case complexity is often of the same order as worst-case complexity. Quicksort is an exception, with a significant improvement for average-case complexity.

We now obtain a recurrence relation for $A(n)$, the average number of comparisons to sort a list of length n. When Quicksort$(0,n-1)$ is invoked it will split the list at some position s. We assume that s is equally likely to take each of the possible values $0,\ldots,n-1$. Each of these values therefore has a probability of $1/n$. If the split is at s, then Quicksort$(0,s-1)$ and Quicksort$(s+1,n-1)$ will be invoked, taking $A(s)$ and $A(n-s-1)$ comparisons respectively. Hence

$$\begin{aligned} A(0) &= 0 \\ A(1) &= 0 \\ A(n) &= n-1+\frac{1}{n}\sum_{s=0}^{n-1}(A(s)+A(n-s-1)) \end{aligned}$$

This can be simplified to

$$\begin{aligned} A(1) &= 0 \\ A(n) &= n-1+\frac{2}{n}\sum_{i=2}^{n-1}A(i) \end{aligned}$$

This is not easy to solve. However we have the following:

Proposition 2.6.11 *$A(n)$ is $\Theta(n\log n)$.*

Proof. We show that $A(n) \leq 2n\ln n$. This is enough to show $A(n)$ is $O(n\log n)$, since logarithms to different bases differ by a constant factor. We then deduce that $A(n)$ is $\Theta(n\log n)$, using the fact that the lower bound for sorting in average case is $\Theta(n\log n)$.

By induction. Clearly holds for $n=1$. Assume that for all $i<n$ we have $A(i)\leq 2i\ln i$.

$$\begin{aligned}
A(n) &\leq n-1+\frac{2}{n}\sum_{i=2}^{n-1}A(i) \\
&\leq n+\frac{2}{n}\sum_{i=2}^{n-1}2i\ln i \quad \text{(by Induction Hypothesis)} \\
&\leq n+\frac{2}{n}\int_2^n 2x\ln x\,dx \quad \text{(since } x\ln x \text{ is monotonically increasing)} \\
&\leq n+\frac{2}{n}[x^2(\ln x-\frac{1}{2})]_2^n \\
&\leq n+\frac{2}{n}[n^2(\ln n-\frac{1}{2})-2^2(\ln 2-\frac{1}{2})] \\
&\leq n+2n(\ln n-\tfrac{1}{2}) \quad (\text{since } \ln 2>\tfrac{1}{2}) \\
&\leq 2n\ln n
\end{aligned}$$

The following table shows (to two decimal places) the lower bounds for the average case and the values of $A(n)$ for Quicksort. The figures are very similar to those given for the worst-case lower bounds and $W(n)$ for Mergesort. It is clear that Quicksort gives good performance in average case.

n	1	2	3	4	5	6	7	8	9	10
Lower bound	0	1	2.67	4.67	6.93	9.58	12.37	15.37	18.56	21.84
$A(n)$	0	1	2.67	4.83	7.40	10.30	13.49	16.92	20.58	24.44

Remark 2.6.12 *It might seem that Mergesort should be preferred to Quicksort, since $W(n)$ for Mergesort is almost as good as $A(n)$ for Quicksort. However Quicksort has the advantage that it can be performed "in place" without using extra space, whereas Mergesort needs extra space to perform the merges. Also, our values for $A(n)$ may be a little pessimistic, since it is often possible to increase the chances of a favourable split (that is, one near the middle of the list).*

2.6.5 Heapsort

Our final sorting algorithm is Heapsort, devised in 1964 by J.W.J. Williams, who also invented the binary heap data structure.

We start by defining min and max binary heaps.

A *heap structure* is a *left-complete* binary tree. Left-complete means that if the tree has depth d then

- all nodes are present at depth $0, 1, \ldots, d-1$
- and at depth d no node is missing to the left of a node which is present.

See Figure 2.7 for an example. We call the rightmost node at depth d the *last* node.

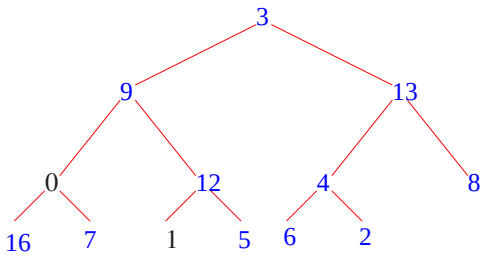


Figure 2.7: Example of a heap structure.

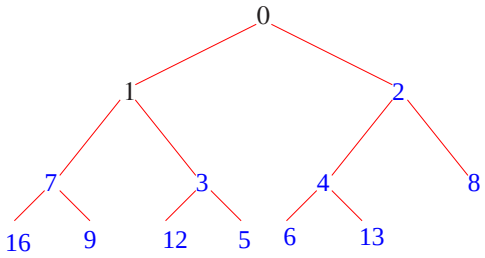


Figure 2.8: Example of a min heap.

We say that a tree T is a *minimising partial order tree* if the key at any node $\leq$ the keys at each child node (if any).

Definition 2.6.13 *A* min heap *is a heap structure with the minimising partial order tree property.*

See Figure 2.8 for an example. Note that for any node of a min heap, the left and right subtrees below the node are also min heaps.

A tree T is a *maximising partial order tree* if the key at any node $\geq$ the keys at each child node (if any).

Definition 2.6.14 *A* max heap *is a heap structure with the max partial order tree property.*

See Figure 2.9 for an example. Again note that for any node of a max heap, the left and right subtrees below the node are also max heaps. Clearly the largest key is at the root of the tree, which allows us to use max heaps to perform sorting.

The scheme is as follows:

Algorithm Heapsort scheme:
Build max heap H out of an array E of elements

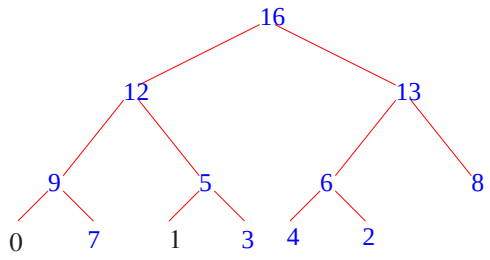


Figure 2.9: Example of a max heap.

for $i = n$ to 1:
  max = getMax(H)
  deleteMax(H)
  $E[i] =$ max

We repeatedly find the maximum element in the heap and add it to the sorted array E starting at the righthand end and filling the array to the left.

Here getMax(H) just reads the root node of H. We have to do more work for deleteMax(H), which removes the maximum element and then takes the remainder and turns it into a heap again (with one fewer element).

Algorithm deleteMax(H) scheme:
copy element at last node into root node
remove last node
fixMaxHeap(H)

Note that copying the last element into the root and removing the last node both preserve the property that the left and right subtrees of the root are max heaps.

We use fixMaxHeap(H) to restore the heap property. Starting from a heap structure H where the left and right subtrees of the root are heaps, this makes H into a heap (again).

Algorithm fixMaxHeap(H) scheme:
if H not a leaf:
  largerSubHeap = the left or right subheap with the larger root
  if root(H).key $<$ root(largerSubHeap).key:
    swap elements at root(H) and root(largerSubHeap)
    fixMaxHeap(largerSubHeap)

If the heap has depth d we see that fixMaxHeap takes at most $2d$ comparisons. Since a heap with n elements has depth $\lfloor \log n \rfloor$, fixMaxHeap takes $O(\log n)$ comparisons.

Before starting heapsort proper, we need to build the initial max heap. Here is a scheme whereby, starting from a heap structure H that does not necessarily have the partial order property, we build a max heap.

Algorithm buildMaxHeap(H) scheme:
if H not a leaf:
  buildMaxHeap(left subtree of H)
  buildMaxHeap(right subtree of H)
  fixMaxHeap(H)

Note that this is an example of a divide and conquer algorithm.

Analysis of buildMaxHeap: Suppose for simplicity $n = 2^k - 1$ so that the heap structure is a complete binary tree with depth $k-1$. Let $W(n)$ be the worst-case number of comparisons for buildMaxHeap.

$$W(n) = 2W((n-1)/2) + 2\log n$$

Apply the Master Theorem with $a = 2$, $b = 2$ and $f(n) = 2\log n$. The critical exponent $E = 1$, and so

$$W(n) = \Theta(n^E) = \Theta(n)$$

So we can build the heap in linear time.

We can now analyse Heapsort overall:

Algorithm Heapsort scheme:
Build max heap H out of an array E of elements $O(n)$
for $i = n$ to 1:
  max = getMax(H) $O(1)$
  deleteMax(H) $O(\log n)$
  $E[i] =$ max

This gives overall $O(n\log n)$ comparisons, the same as MergeSort. Thus Heapsort is an optimal algorithm for sorting by comparisons.

Heapsort has an advantage over MergeSort, in that it can be carried out entirely in place, like Quicksort, if we implement heaps using arrays.

The idea is to store the heap level by level in an array starting at index 1. Starting at index 1 rather than 0 simplifies the arithmetic. Then the left and right children of node at index i are at positions $2i$ and $2i+1$, respectively. The parent node of the node at index i is at $\lfloor i/2 \rfloor$. There is no need for pointers. See Figure 2.10.

Next we present Heapsort on an array:

Algorithm Heapsort(E, n):

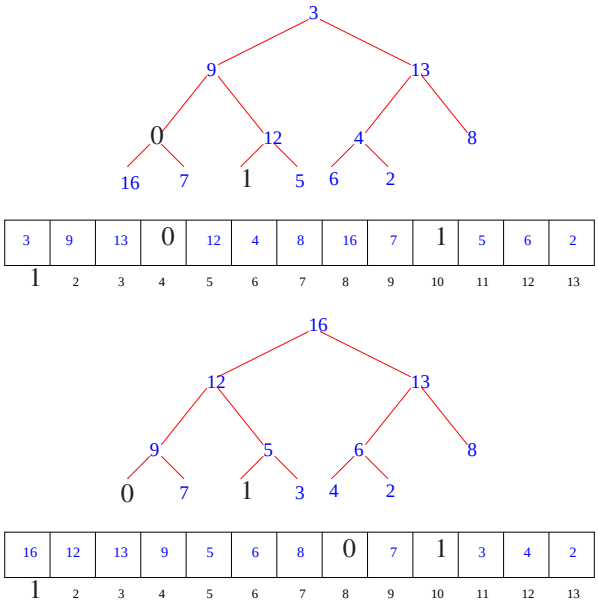


Figure 2.10: Examples of (1) a heap structure and the corresponding array (2) a max heap and the corresponding array.

```
# perform Heapsort on elements 1..n of an array E of elements
heapsize = n
buildMaxHeap(1,heapsize)
# Invariant:
# max heap in E[1..heapsize] of first heapsize-many elements
# of sorted list
# elements in heapsize+1 to n are correctly sorted
while heapsize> 1:
   swap(1,heapsize)
   heapsize = heapsize−1
   fixMaxHeap(1,heapsize)
```

```
Algorithm buildMaxHeap(root,heapsize):
left = 2∗root
right = 2∗root+1
if left ≤ heapsize:
   # root is not a leaf
   buildMaxHeap(left,heapsize)
   if right ≤ heapsize:
      # there is a right subtree
```

```
    buildMaxHeap(right,heapsize)
  fixMaxHeap(root,heapsize)
```

Algorithm fixMaxHeap(root,heapsize):

```
left = 2*root
right = 2*root+1
if left ≤ heapsize:
  # root is not a leaf
  if left = heapsize:
    # no right subheap
    largerSubHeap = left
  elif E[left].key > E[right].key:
    # favours right subheap if equal
    largerSubHeap = left
  else:
    largerSubHeap = right
  if E[root].key < E[largerSubHeap].key:
    swap(root,largerSubHeap)
    fixMaxHeap(largerSubHeap,heapsize)
```

As we said earlier, priority queues can be implemented as binary heaps. Let us recall the Min PQ as used for Prim's algorithm:

- Each item x of the queue has a priority key[x]
- Items removed lowest key first.

The operations are:

- Q = PQcreate()
- isEmpty(Q)
- insert(Q,x)
- getMin(Q)
- deleteMin(Q)
- decreaseKey(Q,x,newkey) — updates key[x] = newkey

Plainly we should use a min binary heap. We sketch how to perform the PQ operations using an array implementation of min binary heaps. Suppose we have an array E plus the heapsize parameter with the heap in $E[1..\text{heapsize}]$ (inclusive).

Some operations are simple to implement:

- *Q = PQcreate()*
  Create an empty array E of a suitable size with heapsize = 0.
- *isEmpty(Q)*
  Check if heapsize = 0. Time $O(1)$.
- *getMin(Q)*
  Return $E[1]$. Time $O(1)$.
- *deleteMin(Q)*
  $E[1] = E$[heapsize]
  decrement heapsize
  fixMinHeap(1,heapsize)
  Time $O(\log n)$.

The insert operation:

Algorithm insert(Q,x):
heapsize = heapsize+1
E[heapsize] = x
# allow x to percolate towards the root until heap property is restored
percolateup(heapsize)

Algorithm procedure percolateup(c)::
if $c > 1$:
  parent $= \lfloor c/2 \rfloor$
  if $E[c]$.key $< E$[parent].key:
    swap(c,parent)
    percolateup(parent)

Time taken is $O(\log n)$.

Remark 2.6.15 *It may be possible to build the queue in one go (time $O(n)$) rather than inserting elements individually (time $O(n \log n)$).*

The decrease key operation decreaseKey(Q,x,newkey) causes more difficulty. If we know the location c of x in the heap then we can change its key to newkey and use percolateup(c) to restore the heap. The problem is to locate x efficiently – in time $O(\log n)$ rather than $O(n)$.

One possible solution is as follows: Suppose that each element has an identifier id. Suppose that identifiers are integers in a compact range [1..maxid]. Use a supplementary array xref to store the location of id: so xref[id] $= k$ means that element is at location k in heap. We need to add code to percolateup to keep xref up to date as swaps occur.

2.7 Dynamic Programming

We have seen several examples of dynamic programming (Floyd, Warshall, Bellman-Held-Karp). We consider an example to illustrate:

- *top-down* versus bottom-up solutions
- *memoisation*

We take as our example the following word break problem:

> Given a string of characters s, can s be split into words occurring in a dictionary?

As an example, $s =$ 'windown' can be split as 'win down' (or 'wind own'). But 'trcarlenz' cannot be split into (English) words. Looking at all possible splits would take too long, as there are exponentially many.

We start with a 'top-down' recursive solution:

Algorithm procedure wb1(s):
```
if len(s) == 0:
   return true
else:
   for i = 0 to len(s) − 1:
      if indict(s[i :]):
         if wb1(s[: i]):
            return true
return false
```

Here indict checks if a string is a word in the dictionary. We shall use the number of indict lookups as a measure of running time. We use a Python-like slice notation:

- $s[i : j]$ is string s from index i to index $j-1$
- $s[: i]$ is string s from start index 0 to index $i-1$
- $s[i :]$ is string s from index i to the end index $\text{len}(s)-1$

The recurrence relation for the worst case on strings of length n is as follows:

$$\begin{aligned} W_1(0) &= 0 \\ W_1(n) &= n + W_1(0) + \cdots + W_1(n-1) \quad (n \geq 1) \end{aligned}$$

This has solution

$$W_1(n) = 2^n - 1$$

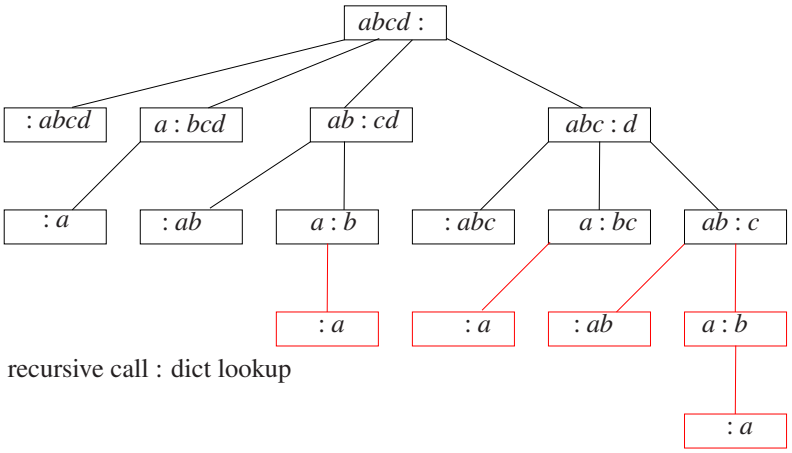


Figure 2.11: Recursion tree for wb1 on a string of length four.

This is exponential! There is inefficiency from computing wb1($s[:i]$) repeatedly. We show the recursion tree for a word of length four in Figure 2.11. The nodes of the tree represent recursive calls and dictionary lookups. We evaluate in depth-first order. The red nodes are duplicates of nodes which have already been evaluated.

We solve the problem of repeated evaluation of recursive calls using what is called *memoisation*. We store previously computed results of recursive calls in an associative array memo, which is initially empty.

```
memo = {} # empty associative array
```

Algorithm procedure wb2(s):

```
if len(s) == 0:
    return true
else:
    for i = 0 to len(s) − 1:
        if indict(s[i :]):
            if memo[s[: i]] undefined:
                memo[s[: i]] = wb2(s[: i])
            if memo[s[: i]]:
                return true
return false
```

Note that the transformation is generic. The recursion tree is now cut off at depth two (the red nodes have been removed). At level 1 we have n nodes, and at level 2 there are $\sum_{i=0}^{n-1} i$ nodes. Adding, we see that the worst case on strings of length n is $\sum_{i=0}^{n} i = n(n+1)/2$. Thus the complexity is now $O(n^2)$. This is a dramatic improvement, taking

us from exponential to polynomial time.

We next develop a 'bottom-up' non-recursive solution: The idea is to solve increasing sub-problems culminating in the main problem:

$$s[:0], s[:1], \ldots, s[:n] = s$$

Note that these problems overlap, which is typical of dynamic programming, in contrast to divide and conquer algorithms where subproblems do not overlap.

We store the result for $s[:i]$ in array wb[i].

Algorithm wb3(s):
n = len(s)
wb[0] = true
if $n > 0$:
  for i = 1 to n:
    wb[i] = false
    for j = 0 to $i-1$:
      if wb[j] and indict($s[j:i]$):
        wb[i] = true
        break
return wb[n]

The complexity is again $O(n^2)$.

We compare the advantages of the two styles. A top-down solution with memoisation is perhaps easier to develop, and may be faster if not all subproblems need computing. A bottom-up non-recursive solution avoids overheads due to recursion.

We conclude with a few remarks. Dynamic programming typically involves taking a problem with exponentially large solution space and finding a path to a solution in polynomial time. Typical problems are to find whether a solution exists (Hamiltonian Circuit, Word Break) or (very often) find the *best* solution (TSP). A dynamic programming solution will tend to involve evaluating overlapping subproblems (in contrast with divide and conquer algorithms). It will also require storing the results of computations of subproblems (requiring more space/memory usage).

Remark 2.7.1 *The name 'dynamic programming' is historic. Here* 'programming' *means planning a suitable order/plan of computation. In many cases the subproblems to consider will depend on the previously obtained results (hence* 'dynamic'*).*

Chapter 3

Introduction to Complexity

3.1 Tractable problems and P

We wish to identify which problems are *tractable/feasible*, or *efficiently computable*, i.e. those problems whose solutions can be computed in a reasonable amount of time. We focus on worst-case analysis, and we want the worst-case complexity $W(n)$ to be not too large, where n is the input size.

Example 3.1.1 *Sorting a list by comparisons.*

We have seen $W(n) = n \log n$ comparisons where n is the length of the list (using Merge-Sort). We can agree that sorting is tractable, since comparing two elements in a list can be done easily.

Example 3.1.2 *The* EULERPATH *problem is: Given an undirected graph G, does G have a Euler path?*

Suppose G has n nodes and m arcs. Input size $|G|$ depends on the representation of G:

- adjacency matrix has size n^2
- linked list has size $n+m$

G has an Euler path iff G has 0 or 2 nodes of odd degree. This is tractable, since we can count the odd degree nodes by making a single pass through either representation, keeping track of the degree count for the current node and the number of odd degree nodes found so far.

By contrast:

Example 3.1.3 *The* HAMPATH *problem is: Given an undirected graph G, does G have a Hamiltonian path (i.e. a path which visits every node exactly once)?*

To see whether a graph with n nodes has a Hamiltonian path seems to require looking at $n!$ different paths - this takes too long. Thus HAMPATH is apparently intractable.

Decision problems are those problems which have a yes/no answer, such as EULERPATH, HAMPATH.

Definition 3.1.4

- *A decision problem D[1] is* decided *by an algorithm A if for any input x, A returns 'yes' or 'no' depending on whether $D(x)$ (and in particular A always terminates).*
- *A decision problem D is* decidable in polynomial time *iff it is decided by some algorithm A which runs within polynomial time, i.e. on all inputs of size n, A takes $\leq p(n)$ steps for some $p(n)$.*

We abbreviate polynomial time to poly time or p-time.

Thus EULERPATH is decidable in p-time, but HAMPATH is (apparently) not.

Thesis 3.1.5 (Cook-Karp Thesis) *A problem is tractable if it can be computed within polynomially many steps in worst case ($W(n) \leq p(n)$ for some polynomial $p(n)$).*

More succinctly we have the slogan: 'tractable = polynomial time'.

According to the thesis, sorting a list is tractable, as is EULERPATH, but HAMPATH is (apparently) not. We use the word 'thesis', rather than 'theorem', since the thesis cannot be proved, since it relates an intuitive concept (tractability) with the notion of polynomial time, which can be made completely precise once we have a specific model of computation.

We have been talking about $p(n)$ steps in a computation on input size n. Clearly the notions of input size and computation step have different meanings depending on the model we are using.

- For sorting a list, we took input size to be the number of list items, and we counted comparisons, ignoring other computation steps (swaps, copying, recursive procedure calls, etc.).
- For EULERPATH we measured input size either using adjacency matrices or adjacency lists, and computation steps would involve inspecting the input and incrementing counters.

[1] We use D rather than P to range over decision problems as P will be the class of polynomial-time problems.

However it turns out that all reasonable models of computation and measures of input size give essentially the same results. If a problem can be solved in polynomial time $p(n)$ in some model, then if we change the model the problem can still be solved in p-time $q(n)$. We may get a different polynomial, but the concept of p-time is robust.

Thesis 3.1.6 (Polynomial invariance thesis) *If a problem can be solved in p-time in some reasonable model of computation, then it can be solved in p-time in any other reasonable model of computation.*

Thus for sorting a list, a different model would be to take the sizes of the items to be sorted into account in measuring input length, and to take into account the fact that comparisons can take different amounts of time depending on the sizes of the items and the length of the list.

However we would still get p-many steps but not necessarily $O(n \log n)$.

Definition 3.1.7 *A decision problem $D(x)$ is in the complexity class* P *(polynomial time) if it can be decided within time $p(n)$ in some reasonable model of computation.*

By the Invariance Thesis this definition is model-independent.

Note that sorting a list does not belong to the class P, since it is not a decision problem. It is technically convenient to define complexity classes for decision problems only, at least to start with.

We referred to 'reasonable' models in the Invariance Thesis. What models would be unreasonable? We give two unreasonable models; the first one relates to computation steps, whereas the second one relates to measuring input size.

1. Superpolynomial parallelism is unreasonable. If we could carry out more than polynomially many operations in parallel in a single step, then we might be able to solve exponential time problems in p-time. So this model is unreasonable.
2. Unary numbers (writing e.g. 11111 for 5) are unreasonable. When dealing with numbers we do not allow unary representation (use base 2 or greater). This is because unary gives input size which is exponentially larger than binary, and so an exponential time algorithm can appear to be p-time.

Example 3.1.8 *With a unary representation we can check whether a number n is prime by looking at all $m < n$ and seeing whether m divides n. This takes n divisions. This is p-time if input size is n.*

However the true input size is actually $\log n$ *and so we have an exp-time algorithm. There are of course far better algorithms for determining whether a number is prime.*

Remark 3.1.9 *The usual arithmetical operations (addition, subtraction, multiplication, division) can be shown to be p-time using the usual binary or decimal representations, i.e. they are polynomial in* $\log n$*, rather than in n.*

The next result shows that polynomial time is well-behaved. It will be useful when discussing reduction and NP-completeness.

Proposition 3.1.10 *Suppose that f and g are functions which are p-time computable. Then the composition $g \circ f$ is also p-time computable.*

Proof. Suppose that $f(x)$ is computed by algorithm A within time $p(n)$ where $n = |x|$, while $g(y)$ is computed by algorithm B within time $q(m)$ where $m = |y|$.

Take input x with $|x| = n$. We compute $g(f(x))$ by first running A on x to get $f(x)$ and then running B on $f(x)$ to get $g(f(x))$. Running A on x takes $\leq p(n)$ steps.

To see how long running B takes, we need a bound on the size of the input $f(x)$. But A runs for $\leq p(n)$ steps to build $f(x)$. So $|f(x)|$ must be polynomially bounded in n — there is no time to build a larger output. Say $|f(x)| \leq p'(n)$ for some polynomial $p'(n)$. Then B runs within $q(p'(n))$ steps.

The total running time (A followed by B) is $p(n) + q(p'(n))$. This is polynomial in n. Hence result.

3.2 The complexity class NP

Consider the HAMPATH problem. Given a graph G, if we guess a list π then it is easy to check whether π is a Hamiltonian path of G.

- check that the items of π are a permutation of nodes(G);
- check that successive nodes of π are adjacent in G.

It is pretty clear that these checks can be carried out in p-time. Thus HAMPATH becomes easy (p-time) if we guess the path. The Hamiltonian path π acts as a certificate that HamPath(G).

Of course if we guess π and we discover that π is not a Hamiltonian path of G, then we are none the wiser, since it might be that G has a (different) Hamiltonian path, or that G has no Hamiltonian path. Nevertheless, it remains the case that if G has a Hamiltonian path then some guess will prove correct.

To make this more precise, let us define the associated verification problem which we call VER-HAMPATH: Given a graph G and a list π, is π a Hamiltonian path of G? Note that VER-HAMPATH(G,π) is in P. Also, clearly

$$\text{HAMPATH}(G) \text{ iff } \exists\pi.\text{VER-HAMPATH}(G,\pi) \ .$$

Definition 3.2.1 *A decision problem $D(x)$ is in* NP *(non-deterministic polynomial time) if there is a problem $E(x,y)$ in* P *and a polynomial $p(n)$ such that*

- $D(x)$ *iff* $\exists y.E(x,y)$
- *if* $E(x,y)$ *then* $|y| \leq p(|x|)$ *(*E *is* polynomially balanced*)*

We require that the certificate y is polynomially bounded in x since otherwise it would take too long to guess y.

Clearly the guess for the Hamiltonian path can be p-bounded in the size of G. So we have checked that HAMPATH $\in$ NP according to the definition of NP.

Remark 3.2.2 *The term 'non-deterministic' is used in the definition of* NP *because some formulations of the definition use an unrealistic notion of computation where the certificate is guessed by the computation making various non-deterministic choices. For instance we can guess a bit-string by non-deterministically choosing to write down either* 0 *or* 1 *on successive steps.*

To sum up the difference between P and NP, we have the slogan:

- P class of decision problems which can be efficiently *solved*
- NP class of decision problems which can be efficiently *verified*

We now introduce a famous decision problem from logic: Boolean satisfiability.

A formula ϕ of propositional logic is in *conjunctive normal form* (CNF) if it is of the form

$$\bigwedge_i (\bigvee_j a_{ij})$$

where each a_{ij} is either a variable x or its negation $\neg x$.

- Terms a_{ij} are called *literals*
- Terms $\bigvee_j a_{ij}$ are called *clauses*

The SAT (satisfiability) problem is as follows: Given a formula ϕ in CNF, is ϕ satisfiable (is there an assignment v to the variables of ϕ which makes ϕ true)?

It seems that SAT is not decidable in p-time: we have to try all possible truth assignments. If ϕ has m variables there are 2^m assignments — exponentially many.

We can let $|\phi|$ be the number of symbols in ϕ and $|v|$ be m (size of the domain of v). Notice that m can be of similar size to $|\phi|$ - every literal could be a different variable.

However SAT does belong to NP, as we can see using the guess and verify method. Given a formula ϕ:

- guess a truth assignment v

- verify in p-time that v satisfies ϕ

As we did with HAMPATH, we define the associated verification problem VER-SAT: VER-SAT(ϕ, v) iff ϕ is in CNF and v satisfies ϕ.

Then:

- SAT(ϕ) iff $\exists v$.VER-SAT(ϕ, v)
- if VER-SAT(ϕ, v) then $|v| \leq |\phi|$ (VER-SAT is p-balanced)

So we have confirmed that SAT $\in$ NP.

There is a simple relationship between classes P and NP:

Proposition 3.2.3 *If a decision problem is in* P *then it is in* NP*, i.e.* P $\subseteq$ NP.

Proof. Suppose that problem D is in P.

Idea: to verify that $D(x)$ holds we don't need to guess a certificate y — we can decide $D(x)$ directly.

More formally, we define $E(x,y)$ iff $D(x)$ and $y = \varepsilon$ (the empty string — a dummy guess). Then clearly

$$D(x) \text{ iff } \exists y.E(x,y) \text{ and } |y| \leq p(|x|)$$

It remains unknown whether P = NP despite many researchers' attempts. This is the most important open problem in computer science. It is arguably one of the most important open problems in mathematics. The Clay Mathematics Institute has offered a prize of one million dollars for a solution (either equal or not equal). It is fair to say that most researchers believe that P $\neq$ NP.

3.3 Problem reduction

Since P $\subseteq$ NP, if we show that a problem such as HAMPATH belongs to NP we do not know whether it is tractable (in P) or not.

We want to identify the hard (high complexity) problems in NP.

We start by saying what it means for one problem to be harder than another, using the concept of *reduction*.

Suppose that D and D' are two decision problems. We say that D *(many-one) reduces* to D' ($D \leq D'$) if there is a p-time computable function f such that

$$D(x) \text{ iff } D'(f(x))$$

Note that f can be a many-one function (hence the name).

The idea is that we reduce a question about D (the easier problem) to a question about D' (the harder problem).

Suppose that we have an algorithm A' which decides D' in time $p'(n)$. Then if $D \leq D'$ via reduction function f running in time $p(n)$ we have an algorithm A to decide D:

Algorithm A (input x):
1. Compute $f(x)$
2. Run A' on input $f(x)$ and return the answer (yes/no)

Now A runs in p-time, as can be seen using the same argument as when composing p-time functions: Step 1 takes $p(n)$ steps. $|f(x)| \leq q(n)$ for some polynomial q. Step 2 takes $p'(q(n))$ steps.

Hence:

Proposition 3.3.1 *Suppose $D \leq D'$ and $D' \in \mathsf{P}$. Then $D \in \mathsf{P}$.*

There is a similar result for NP:

Proposition 3.3.2 *Suppose $D \leq D'$ and $D' \in \mathsf{NP}$. Then $D \in \mathsf{NP}$.*

Proof. Assume that $D \leq D'$ via reduction function f, and $D' \in \mathsf{NP}$. Then there is $E'(x,y) \in \mathsf{P}$ and a polynomial $p'(n)$ such that $D'(x)$ iff $\exists y.E'(x,y)$ and if $E'(x,y)$ then $|y| \leq p'(|x|)$. Also we have $D(x)$ iff $D'(f(x))$ (property of reduction). Combining: $D(x)$ iff $\exists y.E'(f(x),y)$

Define $E(x,y)$ iff $E'(f(x),y)$. Then $D(x)$ iff $\exists y.E(x,y)$. Also $E \in \mathsf{P}$ (composition of p-time functions/relations).

We check that E is p-balanced: Suppose $E(x,y)$. Then $E'(f(x),y)$, so that $|y| \leq p'(|f(x)|)$ As before we have $|f(x)| \leq q(n)$. Hence $|y| \leq p'(q(|x|))$.

Hence $D \in \mathsf{NP}$.

The reduction order is reflexive and transitive:

- $D \leq D$
- if $D \leq D' \leq D''$ then $D \leq D''$

The proofs are left as an exercise.

If both $D \leq D'$ and $D' \leq D$ we write $D \sim D'$. Here D and D' are as hard as each other.

3.4 NP-completeness

We are interested in identifying problems in NP which are unlikely to be in P.

Definition 3.4.1 *A decision problem D is* NP-hard *if for all problems $D' \in \mathsf{NP}$ we have $D' \leq D$.*

Thus NP-hard problems are at least as hard as all NP problems. Note that NP-hard problems do not necessarily belong to NP. They could be harder.

It is easy to see that if D is NP-hard and $D \leq D'$ then D' is also NP-hard (consequence of transitivity of reduction).

Definition 3.4.2 *A decision problem D is* NP-complete *(NPC) if*

1. $D \in \mathsf{NP}$
2. *D is* NP*-hard*

NP-complete problems are the hardest problems in NP.

It is not clear from the definition that NPC problems exist. However:

Theorem 3.4.3 (Cook-Levin 1971) SAT *is* NP*-complete.*

We omit the proof.

Many other problems have been shown to be NPC. Rather than proving this directly as for SAT, we use reduction as follows:

Method 3.4.4 *To see that problem D is NPC show:*

1. $D \in \mathsf{NP}$ *(typically using guess and verify)*
2. *$D' \leq D$ for some known NPC problem D'*

It is clear that item 2 establishes that D is NP-hard, since D' is NP-hard.

As an example take HAMPATH. We have already seen that HAMPATH $\in$ NP by guessing and verifying in p-time. If we can show SAT $\leq$ HAMPATH then we can conclude that HAMPATH is NPC. It is indeed possible to show SAT $\leq$ HAMPATH but we omit the reduction as it is long and difficult. So HAMPATH is NP-complete.

3.4.1 Intractability via NP-completeness

Our original interest was in deciding which problems are tractable and which are intractable. We have a definition for 'tractable', and we have confirmed that problems such as sorting are indeed tractable. What we have not done is show that a problem is intractable. We now see how NP-completeness can help with this.

Proposition 3.4.5 *Suppose* $\mathsf{P} \neq \mathsf{NP}$. *If a problem D is* NP*-hard then $D \notin \mathsf{P}$.*

Proof. Assume $\mathsf{P} \neq \mathsf{NP}$ and D is NP-hard.

Suppose for a contradiction that $D \in \mathsf{P}$. We show that $\mathsf{NP} \subseteq \mathsf{P}$. Take $D' \in \mathsf{NP}$. Since D is NP-hard, we have $D' \leq D$. Hence $D' \in \mathsf{P}$. We have shown that $\mathsf{NP} \subseteq \mathsf{P}$.

But we know $\mathsf{P} \subseteq \mathsf{NP}$. Hence $\mathsf{P} = \mathsf{NP}$ which contradicts our assumption.

Thus if we can show that a problem is NPC, we know that it is intractable (assuming that $\mathsf{P} \neq \mathsf{NP}$, as is generally believed).

Recall the Travelling Salesman Problem TSP: Given a (complete) weighted graph (G,W), find a tour of G *of minimum weight* which visits each node exactly once and returns to the start node.

TSP is an optimisation problem. We first define a decision version of TSP which we call TSP(D): Given a weighted graph (G,W) and a bound B, is there a tour of G with total weight $\leq B$? We can think of B is being a budget or travel allowance. In the decision version, we ask whether there is a tour that does not exceed the budget.

We show that TSP(D) is NP-complete using Method 3.4.4:

1. TSP(D) $\in$ NP:

   If we guess a path p, we can check in p-time that p is a Hamiltonian circuit of G and that $W(p) \leq B$. Clearly $|p| \leq |G|$.

   More formally define VER-TSP(D)$((G,W),B,p)$ iff p is a Hamiltonian circuit of (G,W) and $W(p) \leq B$.

   Then TSP(D)$((G,W),B)$ iff $\exists p.$VER-TSP(D)$((G,W),B,p)$.

   Also if VER-TSP(D)$((G,W),B,p)$ then $|p| \leq |G|$ under reasonable definitions of size.

2. $D' \leq$ TSP(D) for some known NPC problem D':

   We choose HAMPATH as the known NPC problem and show HAMPATH $\leq$ TSP(D).

   We need to define a p-time function f which transforms a graph G into a weighted graph (G',W) together with a bound B so that HAMPATH(G) iff TSP(D)$((G',W),B)$.

   Given G we construct (G',W) as follows: Set nodes(G') = nodes(G). Given any two distinct nodes x,y of G:

   - If (x,y) is an arc of G then (x,y) is also an arc of G', with $W(x,y) = 1$.
   - If (x,y) is not an arc of G then (x,y) is an arc of G', with $W(x,y) = 2$.

   Thus we add in the missing arcs of G but we give them a higher weight. Finally we let $B = n+1$ where G has n nodes.

   It is not hard to see that $f(G) = ((G',W),B)$ is p-time — easiest to see using the adjacency matrix representation.

   We now check that f is a reduction: HAMPATH(G) iff TSP(D)$((G',W),B)$: Suppose G has a Hamiltonian path π with endpoints x and y. The same path in G' has weight $n-1$ (all arcs have weight 1). We get a Travelling Salesman

tour by adding in arc (x,y) with $W(x,y) \leq 2$. Thus we have a tour of weight $\leq n+1 = B$.

Conversely, suppose (G',W) has a tour of weight $\leq B = n+1$. This has n arcs. So at most one arc can have weight 2. Suppose this arc has endpoints x,y. Then omitting arc (x,y) gives us a Hamiltonian path in G.

We conclude that $\mathrm{TSP(D)}$ is NP-complete.

Finally we can show that TSP is intractable (assuming $\mathsf{P} \neq \mathsf{NP}$). Suppose that TSP can be solved by a p-time algorithm. We compute the optimal value O in p-time. Then we can also solve $\mathrm{TSP(D)}$ in p-time — we simply check whether $O \leq B$. So $\mathrm{TSP(D)} \in \mathsf{P}$. But this is impossible by Proposition 3.4.5 since $\mathrm{TSP(D)}$ is NP-complete and we assume $\mathsf{P} \neq \mathsf{NP}$.