Part II: Algorithms

6. Use of algorithms

We now take a rest from Turing machines, and examine the use of algorithms in general practice. There are many: quicksort, mergesort, treesort, selection and insertion sort, etc., are just some of the sorting algorithms in common use, and for most problems there is more than one algorithm. How to choose the right algorithm? There is no 'best' search algorithm (say): it depends on the application, the implementation, the environment, the frequency of use, etc, etc. Comparing algorithms is a subtle issue with many sides.

6.1. RUN TIME FUNCTION OF AN ALGORITHM

Consider some algorithm's implementation, in Modula_2 say. It takes inputs of various sizes. For an input of size n we want to assign a measure f(n) of its use of resources, and, usually, minimise it. Usually f(n) is the *time* taken by the algorithm on inputs of size n, in the worst or the average case (at our choice).

Calculating f(n) exactly can be problematical:

- The time taken to execute an individual Modula_2 instruction can vary even on a single machine, if the environment is shared. So the run time of even the same program for the same data on the same machine may vary.
- There can be a wide variation of use of resources over all the inputs of a fixed size n. The worst case may be rare in practice, and the average case unrepresentative.
- Some algorithms may run better on certain kinds of input. E.g., some text string searching algorithms prefer strings of English text (whose patterns can be utilised) to binary strings (which are essentially random).
- Often we do not understand the algorithm well enough to work out f.

So how to proceed? Much is known about some algorithms, and you can look up information. (We list some books at the end of the section; Sedgewick's is a good place to begin.) Other algorithms are still mysterious. Maybe you designed your own algorithm or improved someone else's, or

you have a new implementation on a new system. Then you have to analyse it yourself.

Often it's not worth the effort to do a detailed analysis: rough rules of thumb are good enough. Suggestions:

- Identify the *abstract operations* used by the algorithm (read, if-then, +, etc). To maximise machine-independence, base the analysis on the abstract operations, rather than on individual Modula_2 instructions.
- Most of the (abstract) instructions in the algorithm will be unimportant resource-wise. Some may only be used once, in initialisation. Generally the algorithm will have an 'inner loop'. Instructions in the inner loop will be executed by far the most often, and so will be the most important. Where is the inner loop? A 'profiling' compilation option to count the different instructions that get executed can help to find it.
- By counting instructions, find a good upper bound for the worst case run time, and if possible an average case figure. It will usually not be worth finding an exact value here.
- Repeatedly refine the analysis until you are happy. Improvements to the code may be suggested by the analysis: see later.

6.1.1. Typical time functions

You can usually obtain the run-time function of your algorithm by a *recurrence relation* (see below). Most often, you will get a run time function **f(n) = c.g(n) + smaller terms**, where n is the input size and:

- c is a constant (with c>0);
- the 'smaller terms' are significant only for small n or for sophisticated algorithms;
- g(n) is one of the following functions:
  - constant (algorithm is said to run in constant time);
  - log(n) (algorithm runs in log time);
  - n (algorithm runs in linear time);
  - n log(n) (algorithm runs in log linear time);
  - n^2 (algorithm runs in quadratic time);
  - n^3 (algorithm runs in cubic time);
  - 2^n (or k^n for some k>1) (algorithm runs in exponential time).

These functions are listed in order of growth: so n^2 grows faster than n log(n) as n increases. The graph in Fig. 6.1 shows some similar functions.

6.1.2. Why these functions?

Why do the functions above come up so often? Because algorithms typically work in one of a few standard ways:

- Simple stack pushes and pops will take constant time, assuming the data is of a small size (e.g., 32 bits for each entry in an integer stack). Algorithms running in low constant time are 'perfect'.
- An algorithm may work by dealing with one character of its input at a time, taking the same time for each. Thus (roughly, and up to a choice of time units) we get

  $f(n) = n$: linear time.

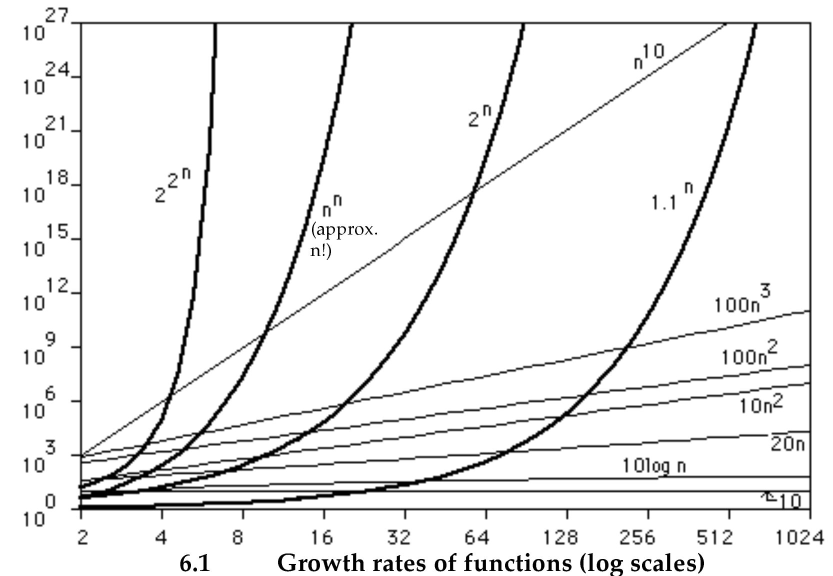
  Algorithms running in linear time are excellent.
- It may loop through the entire input to eliminate one item (e.g., the largest). In this case we'll have:

  $f(n) = n + f(n-1)$.

  Thus $f(n) = n + f(n-1) = n + (n-1 + f(n-2)) = \ldots = n + (n-1) + (n-2) + \ldots + 2 + f(1)$. This is an arithmetic progression, so we get $f(n) = n^2/2+n/2 +k$ for some constant k. The k and n/2 are small compared with n^2 for large n, so this algorithm runs in quadratic time. An algorithm that considers all n^2 pairs of characters of the n-character input also takes quadratic time. Algorithms running in quadratic time can be sluggish on larger inputs.
- Maybe the algorithm throws away half the input in each step, as in binary search, or heap accessing (§7.3.2). So

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

  (this is only an approximation if n is not a power of 2). Then letting $n = 2^x$, we get $f(2^x) = 1 + f(2^{x-1}) = 1+(1+f(2^{x-2})) = \ldots = x + f(2^0) = x+k$ for some constant k. As $f(2^x)$ is about x, f(n) is about $\log_2(n)$: this algorithm runs in log time. The log function grows very slowly and algorithms with this run-time are usually very good in practice.



6.1 Growth rates of functions (log scales)

- The algorithm might recursively divide the input into two halves, but make a pass through the entire input before, during or after splitting it. This is a very common 'divide and conquer' scheme, used in quicksort, mergesort, etc. We have $f(n) = n + 2f(n/2)$ roughly — n to pass through the whole input, plus f(n/2) to process each half. So, using a trick of dividing by the argument, and letting $n = 2^x$ again,

  $$\begin{aligned} f(2^x)/2^x &= 2^x/2^x + 2.f(2^x/2)/2^x \\ &= 1 + f(2^{x-1})/2^{x-1} \\ &= 1 + (1 + f(2^{x-2})/2^{x-2}) = \ldots \\ \ldots &= x + f(2^0)/2^0 = x+c. \end{aligned}$$

  So $f(2^x) = 2^x(x+c) = 2^x.x$ + smaller terms. Thus $f(n) = n.\log_2(n)$ + smaller terms. We have a log linear algorithm. Exercise: what if it divides the input into three?
- Maybe the input is a set of n positive and negative whole numbers, and the algorithm must find a subset of the numbers that add up to 0. If it does an exhaustive search, in the worst case it has to check all possible subsets — 2^n of them. This takes exponential time. If the problem is to find all anagrams of the n-letter input word, it might try all n! possible orderings of the n letters. The factorial function $n! = 1x2x3x\ldots xn$ grows at about the rate of n^n, even faster than 2^n.

6.1.3. The O-notation (recall from 1st year)

This helps us make precise the notion 'my algorithm runs in log time' etc. It lets us talk about functions f(n) for large n, and up to a constant factor.

6.1.3.1. Definition

1. Let f, g be real-valued functions on whole numbers (i.e., functions from {1,2,3,…} into the set of real numbers). We say that *f is O(g)* ('f is of the order of g') if there are numbers m and c such that $f(n) \le c.g(n)$ whenever $n \ge m$.
2. We say that *f is θ(g)* ('theta of g') if f is O(g) and g is O(f). This means that *f and g have the same order of growth.*

So f is of the order of g iff for all large enough n (i.e., $n \ge m$), f(n) is at most g(n) up to some constant factor, c. Taking logs, this means that for <u>all large enough n</u>, $\log(f(n)) \le c' + \log(g(n))$, where c' is a constant (= log(c)). I.e., log f(n) is *eventually* no more than a constant amount above log g(n).

Similarly, f is θ(g) iff there is a constant c such that for all large enough n, log f(n) and log g(n) differ by at most c.

So in the graph of Fig. 6.1, f is O(g) iff for large enough n, the line for f is at most a constant amount higher than that for g (it could be much lower, though!) And f is θ(g) if eventually (i.e., for large enough n) the lines for f and g are vertically separated by at most a fixed distance.

6.1.3.2. Definition

We can now say that an (implementation of an) algorithm *runs in log time* (or linear time) if its run-time function f(n) is θ(log n) (or θ(n), respectively). We define *runs in quadratic, log linear, exponential time*, etc, in the same way.

6.1.3.3. Exercises

1. Show that f is θ(g) iff there are m, c, d (with d>0, possibly a fraction) such that $d.g(n) \le f(n) \le c.g(n)$ for all $n \ge m$.

2. Show that if f is O(g) then there are c, d such that for all n, $f(n) \le \max(c, d.g(n))$. Is the converse true?

3. [Quite long.] Check that the functions in §6.1.1 are listed in increasing order of growth: if f is before g in the list then f is O(g) but not θ(g).

4. Let F be the set of all real-valued functions on whole numbers. Define a binary relation E on F by: E(f,g) holds iff f is θ(g). Show that E is an equivalence relation on F (see §7.1.1 if you've forgotten what an equivalence relation is). (Some define θ(f) to be the E-class of f. For mathematicians: 'f is O(g)' is a pre-order on F.)

5. Show that for any a, b, x>0, $\log_a(x) = \log_b(x).\log_a(b)$. Deduce that $\log_a(n)$ is $\theta(\log_b(n))$. Use Q4 to show that when we say an algorithm runs in log time, we don't need to say what base the log is taken to.

6.1.4. Merits of rough analysis

Note that the statement 'your algorithm runs in log time' (or whatever) will only be an accurate description of its actual performance <u>for large n</u> (so the smaller terms are insignificant), and <u>up to a constant factor</u> (c). A more detailed analysis can be done if these uncertainties are significant, but:

- A rough analysis is quicker to do, and often surprisingly accurate. E.g., most time will be spent in the inner loop ('90% of the time is spent in 10% of the code'), so you might even <u>ignore</u> the rest of your program.
- You may not know whether the algorithm will be run on a Cray or a Mac (or both). Each instruction runs faster by roughly a constant factor on a Cray than on a Mac, so we might prefer to keep the constant factor c above.
- You may not know how good the implementation of the algorithm is. If it uses more instructions than needed in the inner loop, this will increase its running time by roughly a constant factor.
- The run time of an algorithm will depend on the format of the data provided to it. E.g., hexadecimal addition may run faster than binary or decimal addition. So if you don't know the data format, or it may change, an uncertainty of a constant factor is again introduced.

6.1.5. Demerits of rough analysis

In the analysis it's often easy to show f is O(one of the functions g above). To prove f is θ(g) is harder. If you can only show the worst case run time function f to be O(g), so that $f(n) \le c.g(n)$ whenever $n \ge m$, then remember that g is an upper bound only.

In any case, whether you have an O- or a θ-estimate,

- the worst case may be rare;
- the constant c is unknown and could be large;
- the constant m is unknown and could be large.

This can be important in practice. For example, though for very large n we have $2n \log^2(n) < n^{3/2}$, in fact $2n \log^2(n) > n^{3/2}$ even for n=30,000. The moral is: although you should think twice before using an n^2 algorithm instead of an n log(n) one, nonetheless the n^2 algorithm may sometimes be the best choice.

6.1.6. The bottom line (almost…)

Generally, algorithms that run in linear or even log linear time are fine. Quadratic time algorithms are not so good for very large inputs, but algorithms with f(n) up to n^5 are of some use. Exponential time algorithms are hopeless even for quite small inputs, unless their average-case performance is much better.

6.1.7. Average case run time

The average case run time is harder to obtain and more machine-dependent. So your long, complex analyses may only be of any use on your current machine, and may not be worth the effort. Also the average case may not be easy to define mathematically or helpfully: what is 'average' English text? But average cases are useful in geometrical and sorting algorithms, etc.

6.2. CHOICE OF ALGORITHM

So how to choose, in the end? Don't ignore the run time function f(n). Much better algorithms may not be much harder to implement.

But don't idolise it either, as the run time will only be estimated by f(n) for large inputs, (and other caveats above). Moreover, programmers' time is money, so it may be best to keep things simple. The constant factors in f(n) may be unknown or wrong: a factor of 10 is easy to overlook in a rough calculation. So use empirical tests to check performance. But beware: empirical comparison of two algorithm implementations can be misleading unless done on similar machines, delays due to shared access are borne in mind, and equal attention has been paid to optimising the two implementations (e.g., by cutting redundant instructions and procedure calls in the inner loop).

Your choice of algorithm may also be influenced by the prevalent data structures (linked list, tree, etc.) in your programming environment, as some algorithms take advantage of certain structures.

6.3. IMPLEMENTATION

We've seen some of the factors involved in *choosing* an algorithm. But the same algorithm can be *implemented* in many different ways, even in the same language. What advice is there here?

6.3.1. Keep it simple

Go for simplicity first. A brute force solution may be OK if the algorithm is only going to be used infrequently, or for small inputs. So why waste your expert time? (Of course, the usage may change, so be ready to re-implement.) If the result is too slow, it's still a good check of correctness for more sophisticated algorithms or implementations. There are algorithms that are prey to bugs that merely slow up performance, maintaining correctness. A naïve program can be used for speed comparisons, showing up such bugs.

6.3.2. Optimisation

Only do this if it's worth it: if the implementation will be used a lot, or if you know it can be improved. If it is, improve it incrementally:

- Get a simple version going first. It may do as it is!
- Check any known maths for the algorithm against your simple implementation. E.g., if a supposedly linear time algorithm takes ages to run, something's wrong.
- *Find the inner loop and shorten it.* Use a profiling option to find the heavily-used instructions. Are they in what you think is the inner loop? Look at every instruction in the inner loop. Is it necessary, or inefficient? Remove procedure calls from the loop, or even (last resort!) implement it in assembler. But try to preserve robustness and machine-independence, or more programmers' time may be needed later.
- Check improvements by empirical testing at each stage — this helps to eliminate the bad improvements. Watch out for diminishing returns: your time may nowadays be more expensive than the computer's.

An improvement by a factor of even 4 or 5 might be obtained in the end. You may even end up improving the algorithm itself.

If you're building a *large system*, try to keep it amenable to improvements in the algorithms it uses, as they can be crucial in performance.

6.3.3. Useful books

Robert Sedgewick, *Algorithms,* Addison-Wesley, 2nd ed., 1988.
A practical guide to many useful algorithms and their implementation.

A.V. Aho, J.E. Hopcroft, J.D. Ullman, *The design and analysis of algorithms,* Addison-Wesley, 1975. For asymptotic worst-case performance.

D.E, Knuth, *The art of computer programming,* 3 volumes, Addison-Wesley. Does more full average-case analysis, and a full reference for particular algorithms.

G.H. Gonnet, *Handbook of algorithms and data structures,* Addison-Wesley, 1984. Worst- and average-case analysis, & covers more recent algorithms.

The last three are listed in Sedgewick. Maybe that's why all four are from the same publisher.

6.4. SUMMARY OF SECTION

We examined some practical issues and advice to do with choice and implementation of algorithms. We introduced the run time function of an algorithm, in worst case or average case form. It is often most sensible to

make do with a rough calculation of the run time function, obtaining it only for large input size (n) and up to a constant factor (c). In practice, more detailed calculations may be needed. The O-notation helps to compare functions in these terms. We saw how some common algorithm designs give rise to certain run time functions (n, log n, n log n, n^2); these are calculated using recursive equations by considering the 'inner loop' of the algorithm.

7. Graph algorithms

We will now examine some useful algorithms. We concentrate on algorithms to handle graphs, as they are useful, quite challenging, easy to visualise, and will be needed in Part III.

7.1. GRAPHS: THE BASICS

7.1.1. Relations (revision)

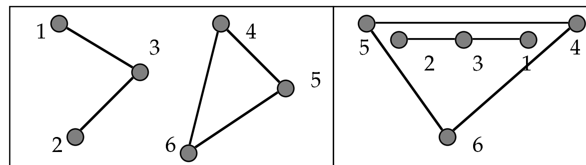
Recall that a *binary relation* R(x,y) on a set X is a subset of XxX. We usually write 'R(x,y) holds', or just 'R(x,y)', rather than '(x,y)∈R'. R is said to be:

- *reflexive*, if R(x,x) holds for all x in X
- *irreflexive* if R(x,x) holds for no element x in X
- *symmetric*, if whenever R(x,y) holds then so does R(y,x)
- *transitive*, if whenever R(x,y) and R(y,z) hold then so does R(x,z)
- *An equivalence relation* if it is reflexive, symmetric and transitive.

7.1.2. Graphs

A *graph* is a pair (V,E) where V is a non-empty set of *vertices* or *nodes,* and E is a symmetric, irreflexive binary relation on V.

We can represent a graph by drawing the nodes as little circles, and putting a line ('edge') between nodes x, y iff E(x,y) holds. In Fig. 7.1, the graph is ({1,2,3,4,5,6}, {(1,3),(3,1), (2,3),(3,2), (4,5),(5,4), (5,6),(6,5), (4,6),(6,4)}).



7.1 Two drawings of the same graph (6 nodes, 5 edges)

If G = (V,E) is a graph, and x, y ∈ V, we say that *there's an edge from x to y* iff E(x,y) holds. We think of (x,y) and (y,x) as representing the same edge, so the number of edges in G is half the number of (x,y) for which E holds. So the graph above has 5 edges, not 10. We'll usually write n for the number of nodes, and e for the number of edges.

7.1.2.1. Exercise

Show that any graph with n nodes has at most n(n-1)/2 edges.

7.1.3. Discussion

There are many examples of graphs, and many problems can be represented in terms of graphs. The London tube stations form the nodes of a graph whose edges are the stations (s,s') that are one stop apart. The world's airports form the nodes of a graph whose edge pairs consist of airports that one can fly between by Aer Lingus without changing planes. The problem of pairing university places to students can be considered as a graph: we have a node for each place and each student, and we ask for every student to be connected to a unique place by an edge. Electrical circuits: the wires form the edges between the nodes (transistors etc).

There are more 'advanced' graphs. *Directed graphs* do not require that E is symmetric. In effect, we allow *arrows* on edges, giving each edge a direction. There may now be an edge from x to y, but no edge from y to x. The nodes could represent tasks, and an arrow from a to b could say that we should do a before b. They might be players in a tournament: an arrow from Merlin to Galdalf means Merlin won. *Weighted graphs* (see §8) have edges labelled with numbers, called *weights* or *lengths.* If the nodes represent cities, the weights might represent distances between them, or costs or times of travel.

Graph theory is an old and difficult area of mathematics. Many useful algorithms for dealing with graphs are known, but as we will see, some are not easy. For example, no fast way to tell whether a graph can be drawn on paper without edges crossing one another was known until 1974, when R.E. Tarjan developed an ingenious linear time algorithm. We'll soon see graph problems with no known efficient algorithmic solution.

7.2. REPRESENTING GRAPHS

How to input a graph (V,E) into a computer? First rename the vertices so that they are called 1,2,…,n for some n. (Maybe use a hashing technique to do this if the vertices originally have names, like London tube stations.) Typically you'll then input the number n of vertices, followed by a delimiter *, followed by a list of all pairs (x,y) ∈ E — perhaps omitting the dual pair (y,x) unless the graph is directed. If the graph is weighted, the weight of an

edge can be added after the edge. This format can be input to a Turing machine if all numbers are input in binary (say) and there's a terminating ∧.

How to represent the graph in a computer? We could just use an nxn Boolean array, where there are n nodes (see Sedgewick's book). But it's often better to use a *linked list,* especially if the graph has relatively few edges. Finding edges is then faster. The graph in Fig 7.1 would be represented by:

1 → 3
2 → 3
3 → 1 → 2
4 → 5 → 6
5 → 4 → 6
6 → 5 → 4

Each line begins with a header vertex (1–6), and lists all vertices connected to it by an edge. There's redundancy (e.g., the edge (1,3) shows up in lines 1 and 3), but this is useful for queries such as 'which vertices are connected to x?' If operations such as deleting a node from a graph are important, it can help to add pointers from the head of each line of the list to the corresponding entries in bodies of the other lines. Here, 1 would get a special pointer to the second entry of line 3. The overhead cost of doing this should be borne in mind.

A linked list can represent directed graphs: the entries in a line headed by x are those nodes y such that there's an arrow from x to y. The weights in a weighted graph can be held in an integer field attached to each non-header entry in the list.

7.3. ALGORITHM FOR SEARCHING A GRAPH

We want to devise a general purpose algorithm that will rapidly visit every node of a graph, travelling only along graph edges. Such an algorithm will be useful for graph searches, measurements, etc. For example, from Dublin (or anywhere reachable from Dublin, for that matter) it would trace out all airports reachable by Aer Lingus, even with plane changes.

7.3.1. The search strategy

The general idea will be this. At any stage of the search, some graph vertices will have been visited. Others will be accessible from the already-visited vertices in one step, by a single edge. They are on the *fringe* of the visited nodes, and are ripe for visiting next. The other vertices will be *far away,* neither visited nor (yet) on the fringe. We will repeatedly:

- choose a fringe vertex,
- visit it and so promote it to 'visited' status

- replace it on the fringe by its immediate (but unvisited) neighbours: i.e., those unvisited nodes that are connected to it by an edge.

At each stage we must decide which fringe vertex to visit next. The choice depends on what we're trying to do.

7.3.1.1. Depth-first and breadth-first search

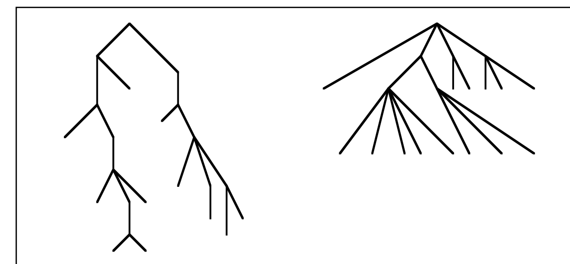
Two common choices are:

(depth first) Visit the newest fringe vertex next: the one that most recently became a fringe vertex. (Last in, first out: implementation could use a *stack*.)

(breadth first) Visit the oldest fringe vertex next. (First in, first out: implementation could use a *queue*.)

In the breadth first approach, all neighbours of the start node get visited first, then the next nearest, and so on. This strategy would be good for a group of people searching for something in a maze. In contrast, the neighbours of the starting node tend to be visited later in the depth first approach. As the next place to visit is usually close by, this approach is good for a single person searching a maze.

Compared with depth first search (heavy arrows in Fig 7.4 below), the edges traced out in breadth first search (Fig. 7.5) tend to form a squatter and bushier pattern, with many short branches. Fig 7.2 shows the kind of shapes to expect. Note that unlike when searching a tree (e.g., in implementing Prolog), the difference between breadth first and depth first search in a graph is not just the order in which vertices are visited. The path actually taken also differs.



7.2 Depth first and (right) breadth first search trees

7.3.1.2. *More general priority schemes*

There's a more general way of choosing a fringe vertex to visit next. Whenever we add a vertex to the fringe, we assign it a *priority.* At each stage, the fringe vertex with the highest priority will be visited next. If there are several fringe vertices with equal priorities, we can choose any of them; the algorithm is non-deterministic.

We can choose any scheme to assign priority. If we let highest priority = newest, we get depth first search; if we let highest priority = oldest, we get breadth first search. So both breadth first and depth first search can be done using priorities. We'll see the effects of other priority schemes in §8.3.

7.3.2. The data structure: priority queue

There is a data structure called a *priority queue* for implementing general, user-chosen priorities. It generalises stacks and queues. It's often implemented as a *heap,* and any access typically takes log time (with stacks and queues, access takes constant time).

For our purposes, we'll assume the priority queue has the following specification.

7.3.2.1. *Specification of priority queue*

The priority queue consists of triples, (x,y,p), where:

- x is an *entry;*
- y is a *label* of x (it can be any extra information we want);
- p is the *priority* of x.

It is x that's in the queue (with label y and priority p). So at most one x-entry is allowed in the queue at any time.

1 We can 'push' onto the priority queue any entry x, with any label y, and any priority p.

2 The push has no effect if x is already an entry in the queue with higher or equal priority than p.
   I.e. if the queue contains a triple (x,z,q), where z is any label and q is a priority higher than or equal to p, the push doesn't do anything.

3 Any x-entry already in the queue but with lower priority than p is removed.
   I.e. if the queue contains a triple (x,z,q), with any label z, and q a lower priority than p, then the push replaces it with (x,y,p).

4 A 'pop' operation always removes from the queue (and returns) an entry (x,y,p) with highest possible priority.

7.3.3. The algorithm in detail

In 'pseudo-code', our algorithm is as follows.

```
1       Visited(n): global Boolean array, initially all false; x: integer
```



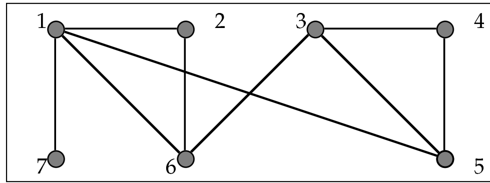
```
2       repeat with x=1 to n
3           if not visited(x) then visit(x)
4       end repeat
5       procedure visit(x)
6           x,y,z: integer                    --- to represent vertices
7           empty the fringe (priority queue)
8           push x into fringe, with label *, and any priority
9           repeat until fringe is empty
10              pop (x,y,p) from fringe  [So x was the queue entry; y was its
                       ---  label; and p was the (highest possible) priority.]
11              visited(x) := true
                    [Anything else you want to do to the new current node, x,
                    such as printing it, do it here!
                    Note: the label y tells us the edge (y,x) used to get to x.]
12              repeat for all nodes z connected to x by an edge
13                  if not visited(z) then
14                      push z into fringe, with label x, and chosen priority
15                  end if
16              end repeat
17          end repeat
18      end visit
```

The nodes of the graph are represented by the numbers 1–n. Note that in line 10, there could be several fringe nodes of equal highest priority. The priority queue non-deterministically pops any such node. The repeat in line 12 does not need to test all nodes k of the graph: it just examines the body of line j of the linked list (§7.2).

The label y in line 10 is useful because it tells us how we got to the current node, x. We usually want to know the route we took when searching the graph, as well as which nodes we visited and in what order. Knowing the order that we visited the nodes in is not enough to determine the route: see the example below.

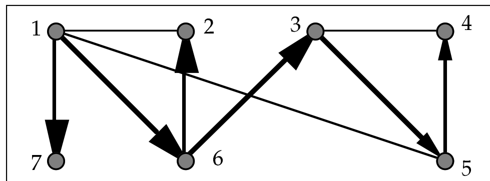
7.3.4. Depth first search: example

Let's see how the algorithm runs on an example graph. Visiting the newest fringe vertex first conducts a *depth first search* of the graph, only moving along edges. Running visit(1) on the graph in Fig 7.3, represented by the linked list below, visits nodes in the order 1,7,6,3,5,4,2:



7.3 Another graph

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



7.4 Depth first search

The 'tree' produced (heavy arrows) tends to have only a few branches, which tend to be long. Cf. Fig 7.2 (left).

7.3.4.1. Execution

We show the execution as a table. Initially, the fringe consists of (1,*,0), where the label * indicates we've just started, 1 is the starting node, and 0 is the (arbitrary) priority. We pop it from the fringe. The immediate neighbours of 1 are numbered 2, 5, 6 and 7. Assume we push them in this order. The fringe becomes (2,1,1), (5,1,2), (6,1,3), (7,1,4), in the format of §7.3.2.1; the third figure is the (increasing) priority.

fringe	pop	visited	print	push	comments
(1,*,0)	(1,*,0)	1		(2,1,1) (5,1,2) (6,1,3) (7,1,4)	
(2,1,1) (5,1,2) (6,1,3) (7,1,4)	(7,1,4)	7	edge '1,7'	-	
(2,1,1) (5,1,2) (6,1,3)	(6,1,3)	6	edge '1,6'	(2,6,5) (3,6,6)	'Backtrack' to visit 6 from 1. Push of 2 has better priority than the current fringe entry, which is replaced.
(5,1,2) (2,6,5) (3,6,6)	(3,6,6)	3	edge '6,3'	(4,3,7) (5,3,8)	The view of 5 from 3 replaces the older view from 1.
(2,6,5) (4,3,7) (5,3,8)	(5,3,8)	5	edge '3,5'	(4,5,9)	Again, this push involves updating priority of node 4.
(2,6,5) (4,5,9)	(4,5,9)	4	edge '5,4'	-	No unseen nbrs. of 4, so no push.
(2,6,5)	(2,6,5)	2	edge '6,2'	-	Another backtrack! No un-visited nbrs of 2, so no pushes.
empty					terminate call of visit(1). Return.

7.3.4.1. WARNING: Significance of the label, y

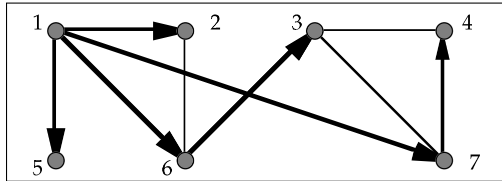
Notice that the nodes were visited in the order 1,7,6,3,5,4,2, but that this does not determine the route. Did we go to 2 from 1 or from 6? Did we arrive at 4 from 3, or from 5? That's why we have to keep the label y in the queue, so that we can tell how we got to each node. This will be even more important in §8, where the route taken is what we're actually looking for.

7.3.4.2. Exercises

1. What route do we get if we start at 2, or 4, instead of 1?
2. What alterations to the code would be needed to implement the fringe with an ordinary stack?
3. Work out how to deduce the path taken in depth-first search, if you know only (a) the graph, and (b) the order in which its nodes were visited. (The main problem, of course, is to handle backtracking.) Can you do the same for breadth-first search (see below)?

7.3.5. Breadth first search: example

Visiting the oldest fringe vertex first conducts a *breadth first search* of the graph, only moving along edges. Running visit(1) on the graph above leads to the sequence 1,2,5,6,7,3,4 of visits:



7.5 Breadth first search

Note the squatter tree. Exercise: work out the execution sequence and try it from different starting nodes.

7.3.6. Run time of the algorithm

Let's simplify by approximating, and only counting data accesses. The fringe is administered by a priority queue that stores nodes in priority order. As we said, this is often implemented by a *heap:* a kind of binary data structure. If a heap contains m entries, accessing it (read or write) takes time log m in the worst case (cf. binary search, §6.1.2).

Suppose the graph has n nodes and e edges. Clearly the fringe never contains more than n entries, so let's assume each fringe access takes time log n (worst case). We count only 1 for emptying the fringe and 1 for the first push in line 8, however. Obviously, each *visited* array access takes constant time: independent of n and e.

- Initialisation of the n elements of the *visited* array to false (line 1), the n reads from it in line 3, and of the fringe (lines 7–8): total = 2n+2.
- Every node is removed from the fringe exactly once (line 10). This involves n accesses, each taking time ≤ log n. Total: n log n.
- For each node j visited, every neighbour k is obtained from the linked list (only count 1 for each access to this, since we just follow the links) and checked to see if it's been visited (lines 12–16; count 1). As each graph edge (j,k) gets checked twice in this way, once from j and once from k, the total time cost here is 2x2e = 4e.
- Not all k's connected to j get written to the fringe, because of the test in line 13. If k is put on the fringe when at j, j will not be put on the fringe later, when at k, as j will by then have been visited. So each edge results in at most one fringe-write. Hence the fringe is written to at most e times. Each write takes log n. Total: e log n.

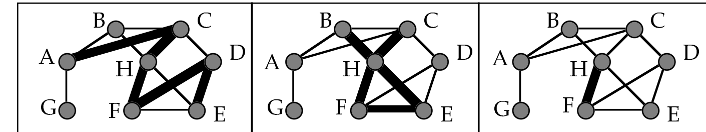
Grand total: 2n + 2 + n log n + 4e + e log n. This is satisfactorily low, and the algorithm is useful in practice. Neglecting smaller terms, we conclude:

The algorithm takes time O((n+e) log n) (i.e., log linear time) in the worst case.

The performance of graph algorithms is often stated as f(n,e), not just f(n).

7.4. Paths and Connectedness

If x, y are nodes in a graph, a *path* from x to y in the graph is a sequence $v_0, v_1, \ldots, v_k$ of nodes, such that k>0, $v_0 = x$, $v_k = y$, and (v_i, v_{i+1}) is an edge for each i with $0 \le i < k$. The *length* of the path is k — i.e., the number of edges in it. The path is *non-backtracking* if the v_i are all different.

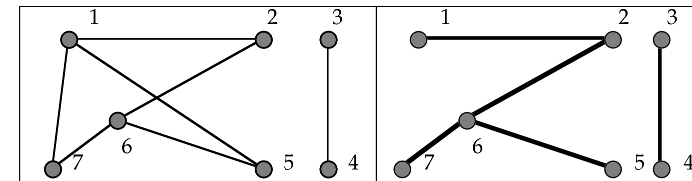


7.6 Paths

In Fig 7.6 (left), the heavy lines show the path ACHFDE from A to E. (They also represents the path EDFHCA from E to A — we can't tell the direction from the figure.) This path is non-backtracking. In the centre, BHFEHC (or BHEFHC ?) is a path from B to C, but it's a backtracking path because H comes up twice. On the right, the heavy line is an attempt to represent the path HFH, which again is backtracking.

7.4.1. Connectedness

The graph of Fig 7.3 is *connected*: there's a path along edges between any two distinct (= different) vertices. In contrast, the graph on the left of Fig. 7.7 is *disconnected.* There's no path from 1 to 3.



7.7 A disconnected graph; a depth first search tree for it

What if we run the algorithm on a disconnected graph like this? In depth first mode, it traces out the heavy lines on the right of Fig. 7.7. *Visit(1)* starts at 1 and worms its way round to 2,6,5 and 7. But then it terminates, and

visit(3) is called (line 3). Whatever priority scheme we adopt, *visit(1)* will only visit all nodes reachable from 1.

7.4.2. Connected components

The nodes reachable from a given node of a graph form a *connected component* of the graph. Any graph divides up into disjoint connected components; it's connected iff there's only one connected component. On any graph, a call of *visit(x)* visits all the nodes of the connected component containing x. *Visit* can't jump between connected components by using edges, so we have to set it off again on each component. Line 3 of the code does this: it will be executed once for each connected component. The number of times *visit* is called counts the connected components of the graph.

7.4.3. Exercises

1. Try the algorithm on Fig. 7.7, starting at 4, in depth- and breadth-first modes. How often is *visit* called in each case?

2. Let G = (V,E) be a graph. Define a binary relation ~ on V by x~y if x = y or there is a path from x to y. Check that ~ is an equivalence relation on V. (The equivalence classes are the connected components of G.)

7.5. TREES, SPANNING TREES

We've already seen in the examples that our algorithm traces out a tree-like graph: the heavy lines in Figs 7.4–7.5 and 7.7. We can say quite a lot about trees.

7.5.1. Trees

A *tree* is a special kind of graph:

7.5.1.1. Definition (very important!)

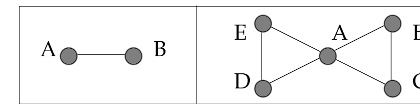
A *tree* is a connected graph with no cycles.

But what's a cycle?

7.5.1.2. Definition

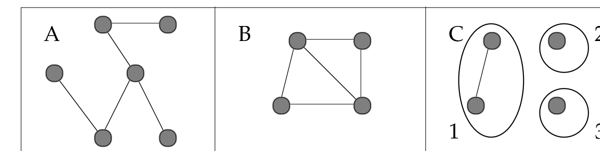
A cycle in a graph is a path from a node back to itself without using a node or edge twice.

So paths of the form ABA, and figures-of-eight such as ABCAEDA (see Fig. 7.8), are not cycles.



7.8 Not cycles!

In Fig. 7.7, 1,2,6,5,1 is a cycle.



7.9 A (tree), B, C (not trees)

In Fig. 7.9, A is a tree. B has a cycle (several in fact), so isn't a tree. C has no cycles but isn't connected, so isn't a tree. It splits into three connected components, the ringed 1,2 and 3, which are trees. Such a 'disconnected tree' is called a *forest.*

7.5.2. Spanning trees

A call of the *visit* procedure always traces out a tree. For as it always visits new (unvisited) nodes, it never traces out a cycle. Moreover, if the graph is connected, a single *visit* call visits all the nodes, so the whole algorithm's trace is a tree. As it contains all the nodes, it's called a *spanning tree* for the graph. If the graph is disconnected we get a spanning tree for each connected component.

7.5.2.1. Definition (very important!)

A tree containing all the nodes of a graph (and only using edges from the graph) is called a *spanning tree* for the graph.

Only a connected graph can have a spanning tree, but it can have more than one spanning tree. The breadth first and depth first searches above gave different spanning trees (Figs 7.4 and 7.5).

A spanning tree is the quickest way to visit all the nodes of a (connected) graph from a starting node, as no edge is wasted. The algorithm starts with the initial vertex and no edges. Every step adds a single node and edge, so the number of nodes visited is always one ahead of the number of edges. Because of this, the number of edges in the final spanning tree is one less than the number of nodes.

If we run the algorithm on a tree, it will trace out the entire tree. (A tree T is connected, so the algorithm generates a spanning tree T' of T. Every edge

of T is in T'. For if e = (x,y) were not in T', then as x and y are in T', there's a (non-backtracking) path from x to y in T'; and this path, plus e, gives a cycle in the original tree T — impossible. So T'=T.)

Thus we see:

7.5.2.2. Result

Any tree with n vertices has n-1 edges.

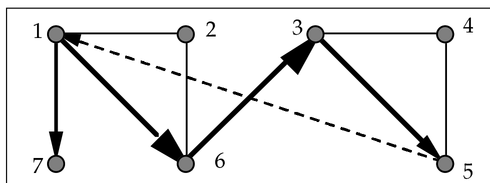
7.5.3. Testing for cycles

If the original connected graph (with n nodes) has ≥n edges, it must have a cycle. For any edge not in a spanning tree must connect two nodes that are already joined by a path in the tree. Adding the extra edge to this path gives a cycle.

We can use this to find out if a connected graph has a cycle. Just count its vertices and edges. There's a cycle iff no. of edges ≥ no. of vertices. If the graph is disconnected, we could do this for every connected component in turn.

Another way is to modify the algorithm (§7.3.3) to check, each time round the main loop of lines 9–17, whether the test in line 13 is failed more than once. If this ever happens, there's a cycle, because the algorithm has found two 'visited' neighbours z of the current node x. One such z is the node one step higher in the tree than the current node (if any). The other z indicates a cycle.

Example: when at node 5 during the depth first search of Fig. 7.4, nodes 1 and 3 were rejected as fringe contenders because they had been visited earlier. Node 3 is the previous node in the search tree; but node 1's visibility from 5 indicates the presence of a cycle, as we can travel from 5 to 1 directly and then return to 5 via the tree (via 6 and 3). See Fig. 7.12.



7.12 Cycle 5163

7.5.3.1. Exercises

1. Let U = (S,L) where S is the set of London tube stations, and (x,y) ∈ L iff x is exactly one stop away from y. Is U a connected graph? Is it a tree?

2. Show that any two distinct nodes x, y of a tree are connected by a unique non-backtracking path.

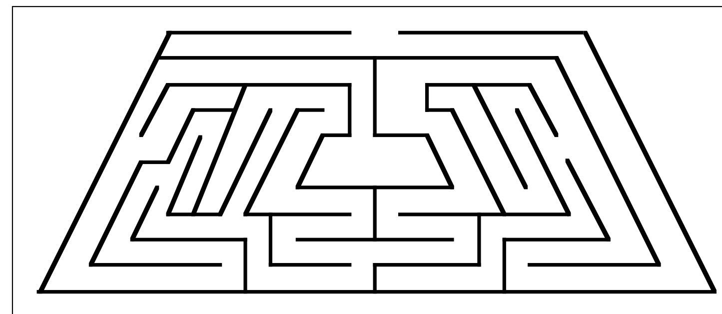
3. Show that any (even disconnected) graph with at least as many edges as vertices must contain a cycle. [Hint: add some edges between components.]

4. Show that no graph with 10 nodes and 8 edges is connected.

5. Show that any connected graph with n nodes and n-1 edges (for some n≥1) is a tree. Find a graph with n nodes and n-1 edges (for some n) that's not a tree.

6. Let G = (V,E) be a graph, and T = (V,P) a subgraph (so every edge of T is an edge of G). Show that T is a spanning tree of G iff it's connected and has n-1 edges, where n is the number of nodes.

7. Fig. 7.13 is a picture of the maze at Hampton Court, on the river west of London, made by Messrs. Henry Wise and George London in 1692.


7.13 Hampton Court maze

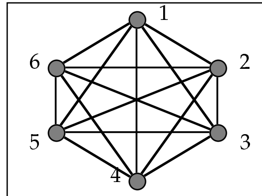
Draw a graph for this maze. Put nodes at the entrance, the centre, and at all 'choice points' and dead ends. Join two nodes with an edge if you can walk directly between them without going through another node. Is the graph connected? Is it a tree? What information about the maze is not represented in the graph?

7.6. COMPLETE GRAPHS

Graphs with the maximum possible number of edges are called *complete graphs*. So (V,E) is complete iff (x,y) ∈ E for all x ≠ y in V.

7.6.1. Exercises

1. What spanning trees are obtained by depth first and breadth first search in the following complete graph? How many writes to the fringe are there in each case?

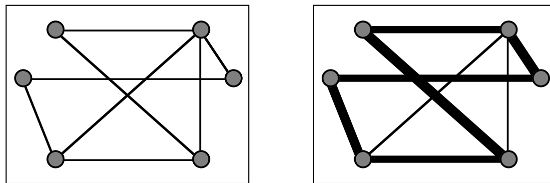


7.14 The complete graph on 6 vertices

2. How many edges does a complete graph on n vertices have?

7.7. HAMILTONIAN CIRCUIT PROBLEM (HCP)

A Hamiltonian circuit of a graph is a cycle containing all the nodes of the graph.



7.15 Graph (left) with a Hamiltonian circuit (right)

The Hamiltonian Circuit Problem (HCP) asks: does a given graph have a Hamiltonian circuit?

Warning: HCP is much harder than the previous problems. Our search algorithm is no use here: we want a 'spanning cycle', not a spanning tree. An algorithm could check every possible ordered list of the nodes in turn, stopping if one of them is a Hamiltonian circuit. If the graph has n nodes, there are essentially at most (n-1)!/2 such lists: n! ways of ordering the nodes, but we don't care which is the start node (so divide by n), or which way we go round (so divide by 2). Whether a given combination is a Hamiltonian circuit can be checked quickly, so the (n-1)! part dominates the time function of this algorithm. But (n-1)! is not $O(n^k)$ for any number k. It is not even $O(2^n)$ (exponential time). There is no known *polynomial time solution* to this

problem: one with time function $O(n^k)$ for some k. We will look at it again later, as it is one of the important class of NP-complete problems.

7.7.1. Puzzle

Consider the squares on a chess-board as the nodes of a graph, and let two squares (nodes) be connected by an edge iff a knight can move from one square to the other in one move. Find a Hamiltonian circuit for this graph.

7.8. SUMMARY OF SECTION

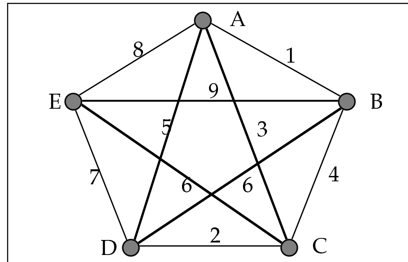
We examined some examples of graphs, and wrote a general purpose graph searching algorithm, which chooses the next node to examine according to its priority. Different priorities gave us depth first and breadth first search. We saw that it traced out a spanning tree of each connected component of the graph. We can use it to count or find the connected components, or to check for cycles. It runs in time O((n+e) log n) at worst (on a graph with n nodes and e edges). We defined a complete graph, and briefly looked at the (hard) Hamiltonian circuit problem.

8. Weighted graphs

Now we'll consider the more exotic but still useful *weighted graph*. We'll examine some weighted graph problems and algorithms to solve them. Sedgewick's book has more details.

8.1. EXAMPLE OF WEIGHTED GRAPH

Imagine the nodes A–E in Fig. 8.1 are towns. An oil company wants to build a network of pipes to supply all the towns from a refinery at A. The numbers on the edges represent the cost of building an oil pipeline from one town to another: e.g., from A to D it's £5 million. The problem is to find the cheapest network.



8.1 A weighted graph

8.1.1. What is a weighted graph?

We can represent the map above by a weighted graph. The nodes are the towns A–E, all edges are present, and the weight on each edge is the cost of building a pipe between the towns it connects. Formally, a *weighted graph* is a triple (V,E,w) where:

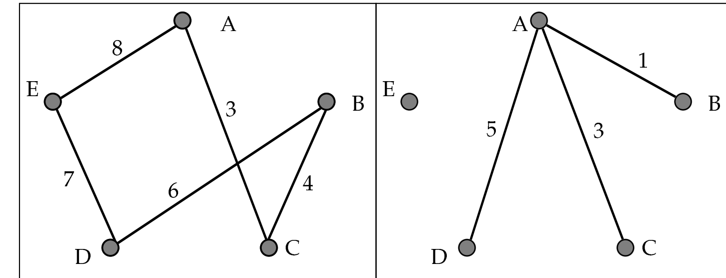
- (V,E) is a graph
- w : E → {1,2,…} a map providing a number (the *weight*) for each vertex pair ('edge'). We require that w(x,y) = w(y,x) for all (x,y) ∈ E (so that each edge gets a well-defined weight).

We'll usually assume that weighted graphs (V,E,w) are connected (this means that (V,E) is connected).

So a weighted graph is just a graph with a number attached to each edge. The numbers might be distances, travel times or costs, electrical resistances, etc. Often, depending on the problem, (V,E) will be a complete graph, as we can easily represent a 'non-edge' by a very large (or small) weight. Fig. 8.1 represents a weighted complete graph, as all edges are present.

8.2. MINIMAL SPANNING TREE

A proposed network of pipes can be represented by a graph (V,P). V is the set of towns as above, and P is the set of proposed pipelines. We put an edge (x,y) in P iff a pipe is to be built directly between x and y. Two possible pipe networks are given in Fig. 8.2.



8.2 Two possible pipelines

Clearly, the cheapest network will have only one pipeline route from any town to any other. For if there were two different ways of getting oil from A to B (e.g., via C or via E and D, as on the left of Fig 8.2), it would be cheaper to cut out one of the pipes (say the expensive one from A to E). The remaining network would still link all the towns, but would be cheaper. So:

- the pipes the company builds should form a tree.

The right hand pipeline network in Fig 8.2 does not connect all the towns. As every town should lie on the network,

- the tree should be a spanning tree (of the complete graph with vertices A–E).
- And its total cost should be least possible.

8.2.1. Definition

A *minimal spanning tree* (MST) of a (connected) weighted graph (V,E,w) is a graph (V,P) such that:

- (V,P) is connected
- P ⊆ E
- the sum of the weights of the edges in P is as small as possible, subject to the two previous constraints.

A minimal spanning tree (V,P) will be a tree, for (as above) we could delete an edge from any cycle, leaving edges still connecting all the nodes but with smaller total weight. Because (V,P) must be connected, it must be a spanning tree of (V,E).

A MST will give the oil company a cheapest network. There may be more than one such tree: e.g., if all weights in (V,E,w) are equal, any spanning tree of (V,E) will do. Though one might find a MST in the graph of Fig 8.1 by inspection, this will be harder if there are 100 towns, say. We need an algorithm to find a MST.

8.3. Prim's algorithm to find a MST

Our search algorithm gave a <u>spanning tree</u>; can we modify it to give a <u>minimal</u> one in a weighted graph? Let's try the following: when we push a node (town) onto the fringe, its priority will be the length of the edge joining it to the current node. A short length will mean high priority for popping, a long one low priority. See below for an example of this algorithm in action.

8.3.1. Proving correctness of Prim's algorithm

This idea seems intuitively correct. The graph will be explored using the shortest edges first, so the spanning tree produced <u>has a good chance</u> of being minimal. But how can we be <u>sure</u> that it always delivers a MST in any weighted graph? After all, the algorithm operates 'locally', working out from a start node; whereas a MST is defined 'globally', as a spanning tree of least weight. Maybe the best edges to use are at one side of the graph, but if we start the algorithm at the other side, it'll only find them at the end, when it's too late. (See §8.5.1 below for an apparently similar case, where these difficulties seem fatal.)

So we should <u>prove</u> its correctness. This is not so hard if we know the following property of MSTs. (If we don't, it can be seriously nasty!)

8.3.1.1. Useful 'separation' property

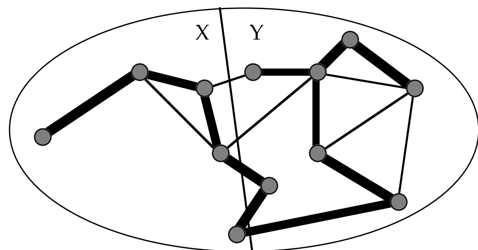
Let (V,E,w) be any connected weighted graph, and T = (V,P) be any spanning tree. We say that T has the 'separation property' if:

(*) Given <u>any</u> division of the nodes in V into two sets, the MST T contains one of the shortest (lowest weight) edges connecting a node in one set to a node in the other.

I used to call this the X–Y property.

8.3.1.2. Example

Fig. 8.3 shows a weighted graph, the weight w(x,y) being the distance from x to y.



8.3 Does this tree have the separation property? (weight ≈ distance)

The bold lines form a spanning tree; the light lines are the graph edges not in the tree. We've chosen an arbitrary division of the nodes into sets X, Y. If the spanning tree has the separation property, no graph edge from X to Y should be shorter than the three heavy tree edges crossing the X–Y division.

8.3.1.3. .i.Warning; — what the separation property is not.

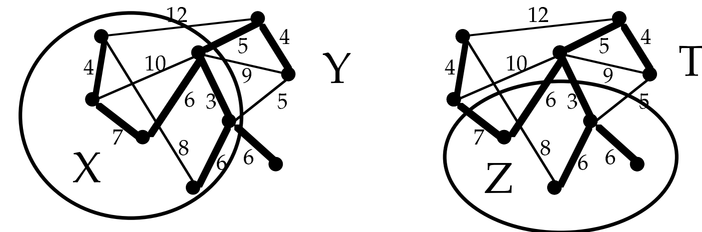
There might be <u>more than one</u> shortest edge from X to Y. (They'll all be equal length, of course. For example, this happens if all graph edges have the same length!) The separation property says that <u>at least one</u> of them is in the spanning tree.

The separation property is talking about shortest X–Y <u>edges</u>, not shortest paths. <u>It is false</u> in general that the shortest path between any node of X and any node of Y is the path through the tree. Look for yourself. The top two nodes in figure 8.4 below are connected by an edge of length 12. But the path between them in the tree shown has length 4+7+6+5=22 — and the tree does in fact have the separation property.

8.3.1.4. Separation property for MSTs?

What's all this got to do with Prim's algorithm? Well, we will show that <u>any MST has the separation property</u>. Let's see an example first.

8.3.1.5. Example with numerical weights



8.4 The MST has one of the least weight X–Y (and Z–T) edges

The heavy edges form an MST. On the left, X is the set of nodes in the circle, and Y is the rest. There are two least X–Y edges (of weight 5), and one of them is indeed in the MST shown, as the separation property says.

On the right, I used a different division, Z–T, of the same weighted graph. The shortest Z–T edge is of length 3 — and again, it's in the MST.

So the separation property might just hold for this MST, if we checked all sets X, Y. But in general? In fact, any MST has the separation property. But we can't establish this by checking all possible MSTs — there are infinitely many MSTs, and we wouldn't have the time. We will have to prove it.

8.3.1.6. Any MST has the separation property.

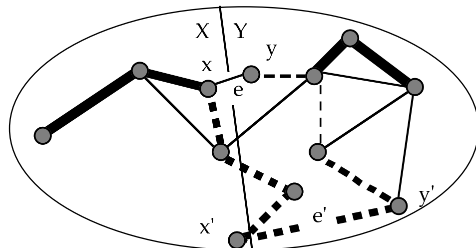
Proof. We will show that any spanning tree that does not have the separation property is not an MST.

Suppose then that:

- T is a spanning tree of the weighted graph (V,E,w).
- there's a division of V into two sets X, Y
- there's an edge e = (x,y) ∈ XxY that's shorter than any edge of T connecting X and Y.

We'll show that T is not a MST.

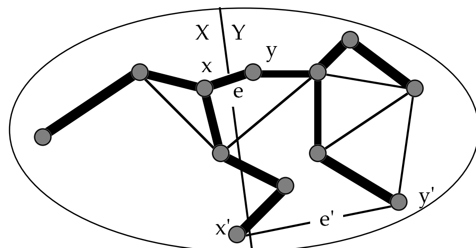
As T is a spanning tree, there's a unique path in T connecting x to y (the dotted line in Fig. 8.5). This path must cross over from X to Y by some edge e' = (x',y') ∈ E. We choose any cross-over edge if there's more than one.



8.5 A short edge e from X to Y

<u>Let's replace e' by e in T</u>. We get T* = (V,(P∪{e})\{e'}) (see Fig. 8.6). Then:

- As e is shorter than e', T* has <u>smaller total weight</u> than T.
- T* has <u>no cycles</u>. Although adding e to T produces a unique cycle, taking e' out destroys it again.
- T* is <u>connected</u>. For if z, t ∈ V are different nodes, there was a path from z to t in T. If this path didn't use the edge e', it's still a path in T*. If it did use e', then the path needed to get from x' to y'. But we can get from x' to y' in T*, by going via e. So we can still get from z to t in T*.



8.6 New spanning tree T* (e' replaced by e)

So T* is a spanning tree.

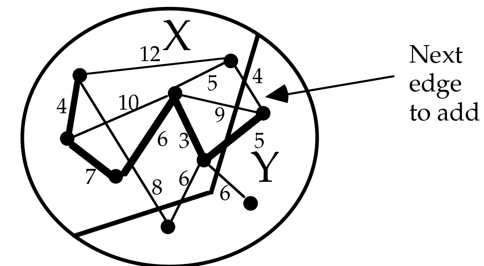
But T* is a spanning tree with smaller weight than T. So T was not a MST. The separation property is proved. QED.

We're not finished yet. We showed any MST has the separation property; but we still have to show our algorithm builds a MST.

8.3.2. Proof of correctness of algorithm

Assume for simplicity that all graph edges have different weights (lengths). (The algorithm finds a MST even if they don't: proving this is a tutorial exercise.) Let T be any MST. At each stage, our proposed algorithm adds to its half-built tree X the shortest possible edge connecting the nodes of X with the remaining nodes Y. By the ‘separation’ property, 8.3.1.1, the MST also includes this edge. So every edge of the tree built by the algorithm is in T.

For example, here's Prim's algorithm half way through building a MST for the graph in Fig. 8.4.



8.7 Prim's algorithm in progress

X is the half-built tree — the nodes already visited. Y is the rest. The algorithm will add the edge shown in the next step, as it has highest priority on the fringe at the moment (check this!) But this edge is <u>the</u> shortest X–Y edge. So it is also in any MST — it's in the one shown in fig. 8.4.

But all spanning trees have the same number of edges (n-1, where the whole graph has n nodes; see §7.5.2.2). We know the algorithm always builds a spanning tree — so it chooses n-1 edges. But T is a MST, so also has n-1 edges. So the tree built by the algorithm <u>is</u> T. This is true even with fractional or real-number weights.

8.3.2.1. Challenging exercises

1. Deduce that if all edges in a weighted graph have different weights, then it has a unique MST. Must this still be true if some edges have equal weight? Can it still be true?
2. Is it true that any spanning tree (of a weighted graph) that has the separation property is a MST of that graph? (This, 'separation property ⇒ MST', is the converse of §8.3.1.6.)
3. Here's a proposed algorithm to find a MST of a connected weighted graph G.

1. Start with any spanning tree T of G
2. Pick any X–Y division of the nodes of G
3. If T doesn't have a shortest X –Y edge, replace an X –Y edge of T with a shorter one [as in the proof, §8.3.2, especially Figs 8.5, 8.6]
4. Repeat 2,3 until T doesn't change any more, whichever X, Y are picked.

a) Does this terminate?
b) If it does, is the resulting tree T a MST of G?
c) If so, would you recommend this algorithm? Why?

8.3.2.2. Warning

If we run the algorithm on the graph in Fig. 8.1, starting from node A, we get a MST — we just proved this. If we start it from node D, we also get a MST — we proved that the algorithm always gives a MST. So wherever we start it from, it delivers a MST. Of course, we may not always get the same one. But if all edges had different weights, we would get the same MST wherever we started it from (by previous exercise).

So to get an MST, there is no need to run the algorithm from each node in turn, and take the smallest tree found. It gives an MST wherever we start it from.

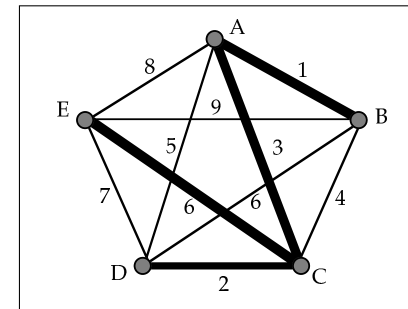
Try the algorithm on Fig. 8.1, starting from each node in turn. What is the total weight of the tree found in each case? (They should all be the same!) Do you get the same tree?

8.3.3. Implementation and execution of Prim's algorithm

We can use the algorithm of §7.3.3. When we push (x,y) onto the fringe (priority queue) we give it priority w(x,y), where low weight = high priority for popping. (We can write this as 'push (x,y,w(x,y)) onto fringe'.) E.g., if edge (A,C) has weight 4 we push (A,C,4) onto the fringe. When we pop an edge (x,y) we pop the one with highest priority — i.e., lowest weight.

A run of MST(A) for the weighted graph in Fig. 8.1 looks like this. First, push (A,*,0) into queue first. The run is then as shown in the table. The MST we get is 'AB, AC, CD, CE', of total length 12:

fringe	pop	visited	print	push	comments
(A,*,0)	(A,*,0)	A		(B,A,1) (C,A,3) (D,A,5) (E,A,8)	
(B,A,1) (C,A,3) (D,A,5) (E,A,8)	(B,A,1)	B	edge 'A,B'	(C,B,4) (D,B,6) (E,B,9)	C, D and E are already in the fringe with better priority, so the pushes have no effect.
(C,A,3) (D,A,5) (E,A,8)	(C,A,3)	C	edge 'A,C'	(D,C,2) (E,C,6)	Both pushes have better priority than the current fringe entries, which are replaced.
(D,C,2) (E,C,6)	(D,C,2)	D	edge 'C,D'	(E,D,7)	This has lower priority than current entry for E, so no dice.
(E,C,6)	(E,C,6)	E	edge 'C,E'	-	
empty					terminate



8.8 the MST found from Fig. 8.1

8.3.4. Run time of Prim's algorithm

On graphs with few edges, the algorithm runs in time O((n+e)log n), where there are n nodes and e edges. Cf. §7.3.6. Another algorithm (due to Kruskal) runs in O(e log e).

8.4. SHORTEST PATH

Suppose in a weighted graph we want to find the path of least possible length from node x to node y. We use the algorithm to build a spanning tree, starting at x. For each node z added to the tree, we keep a tally of its distance d(z) from x through the tree as built so far, and add to the fringe all neighbours t of z with priority d(z)+w(z,t). We stop when y gets into the tree. The shortest path from x to y is then the unique path from x to y through the tree. This algorithm is essentially due to Dijkstra (a big cheese). Exercise: try it on an example. And prove it correct!!

8.5. TRAVELLING SALESMAN PROBLEM (TSP)

Example: Consider Fig. 8.1 again. Suppose the numbers on the edges represent road distances between the towns[1]. A salesperson lives in A and wants to make a round trip, visiting each city just once and returning home at the end. The manager conjures a figure, d, out of the air. If the whole trip is more than d miles, no expenses can be claimed.

Problem: Is there a route of length ≤d?

We can formalise this problem using weighted graphs.

Problem (TSP): Given a weighted complete graph (V,E,w), and a number d, is there a Hamiltonian circuit of (V,E) of total length at most d?

As the graph is complete, there will be many Hamiltonian circuits, but they may all be longer than d.

This is not a toy problem: variants arise in network design, integrated circuit design, robotics, etc. See Harel's book, p.153. TSP is another hard problem. The exhaustive search algorithm for HCP also works for TSP. There are (n-1)! possible routes to consider. For each route, we find its length (this can be done in time O(n)) and compare it with d. As for HCP, this algorithm runs even slower than exponential time. There is no known polynomial time solution to TSP. Some heuristics and sub-optimal solutions in special cases are known.

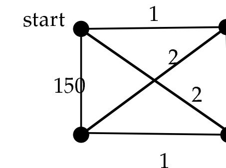
8.5.1 Nearest neighbour heuristic for TSP

One might hope that the following algorithm would find the shortest Hamiltonian circuit in any weighted graph (V,E,w):

```
start by letting current_node be any node of V
repeat until all nodes have been visited
    go to the nearest node to current_node
        [the node x such that w(x,current_node)_ is least]
end repeat
```

[1] Important: roads between the towns may not be straight! I.e., there may be three towns, x, y, znd z, with w(x,z) > w(x,y)+w(y,z). (There is a p-time solution to TSP on graphs where this never happens.)

This is called the *nearest neighbour heuristic.* It works locally, choosing the nearest neighbour to the current node every time. It's the nearest algorithm to Prim's algorithm for finding a MST (§8.3), and it is similarly fast. But while Prim's algorithm is correct, actually delivering a MST, the performance of the nearest neighbour heuristic is absolutely diabolical in many cases — it's one of the worst TSP heuristics of all. The energetic will find seriously incriminating evidence in Rayward-Smith's book; the rest of us may just try the heuristic on the following graph.



8.9 A bad case for nearest neighbour

One might easily think that the nearest neighbour heuristic was 'intuitively a correct solution' to TSP. It takes the best edge at each step, yes? But in fact, it is far from being correct. Intuition is surely very valuable. Here we have an awful warning against relying on it uncritically.

8.6. POLYNOMIAL TIME REDUCTION

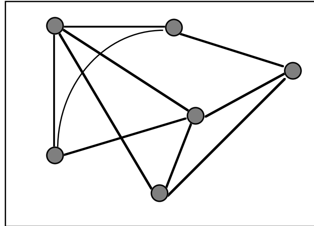
Though similar to TSP, HCP seems rather easier. We can formalise this using the *reduction* of §5, with the new feature that now we want the reduction to be fast. Suppose we have a fast method F of transforming a graph G into a complete weighted graph G* plus a number d, so that G has a Hamiltonian circuit iff G* has a round trip of length ≤d. That is:

- G is an instance of HCP,
- F(G) = <G*,d> is an instance of TSP,
- the answers (yes or no) are the same for G as for <G*,d>.

Then any fast method M of solving TSP could also be used to solve HCP quickly. For given an instance G of HCP, we transform it quickly into F(G) and apply M, which is also fast. Whatever answer M gives to F(G) (yes or no) will also be the correct answer to G. By *fast* we mean *takes polynomial time* (see §7.7) *in the worst case.* This technique is called *polynomial time (p-time) reduction.* See part III.

8.6.1. Example: p-time reduction of HCP to TSP

Suppose that we have an instance of HCP: a graph such as:

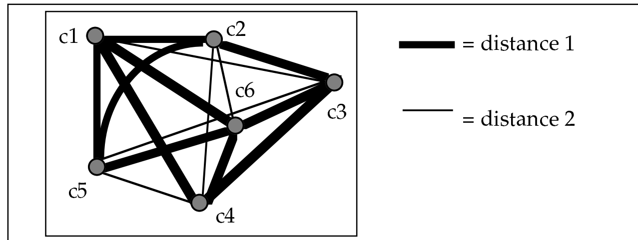


8.10 An instance G of HCP … but is it a yes-instance?

We can turn it into an instance of TSP by:

- defining the distance between nodes x and y to be 1, if x is joined to y in the graph,
- defining the distance from x to y to be 2, if x is not joined to y,
- defining the bound 'd' to be the number of nodes.

We get:



8.11 The instance F(G) of TSP; d = 6

This conversion takes time about n^2 if there are n nodes, so is p-time. Then

- any Hamiltonian circuit in the original graph yields a round trip of length n in the weighted graph.
- Conversely, any round trip in the weighted graph must obviously contain n edges; if it is of length ≤n then all its edges must have length 1. So they must be real edges of the original graph.

So the original graph has a Hamiltonian circuit iff there's a route of length ≤n in the corresponding weighted graph.

E.g., in Figure 8.11, the route (c1,c2,c3,c4,c6,c5,c1) has length 6.

8.7. NP-COMPLETENESS TASTER

So HCP is 'no harder' than TSP. In fact they are about the same difficulty: one can also reduce TSP to HCP in p-time, though this is more tricky. Both TSP and HCP are examples of *NP-complete problems.* Around 1,000 problems are now known to be NP-complete, and they all reduce to each other in p-time. In practice, NP-complete problems are *intractable:* currently, even moderately large instances of them can't be handled in a reasonable time, and most people believe that no fast solution exists. We'll examine NP-completeness in Part III.

8.8. SUMMARY OF SECTION

We discussed weighted graphs and applications. Using a 'short edge = high priority' fringe popping strategy, we found an algorithm for finding a minimal spanning tree (MST) in a weighted graph, and proved it correct. There's a unique MST if all edges have different weights. We gave an algorithm to find the shortest path between two nodes of a weighted graph. We mentioned the (hard) travelling salesman problem, and showed that any fast solution to it would provide a fast solution to the Hamiltonian circuit problem (§7.7). No polynomial time solution to either of these is known.

Part II in a nutshell

§6: When choosing an algorithm it helps to know roughly the time f(n) that it'll take to run on an input of a given size n. As many inputs have size n, we usually consider the worst or the average case. Worst case run time estimates are easier to find. They can often be calculated by a recurrence relation. Because many algorithms use the same techniques, run time functions often have the form f(n) = constant, log(n), n, n log(n), n^2, or 2^n. Usually a rough estimate will do: we get f(n) = c.g(n) + smaller terms. (If f(n) ≤ c.g(n) for large enough n, we say that f is O(g). If f is O(g) and g is O(f) we say that f is θ(g).) The uncertainties in the estimate may be important and should be borne in mind. Implementation should use careful experiments and may involve optimising the code. *Keep it simple* is a sound rule.

§7: A *graph* is a collection of vertices or *nodes*, some of which are connected by *edges*. Many problems can be represented as problems about graphs. A common graph searching algorithm proceeds from a start node through the graph along edges to other nodes. At each point, the immediate

neighbours of the current node are added to the 'fringe' of vertices to visit next. Which fringe vertex is taken depends on its priority, which can be assigned in any way. E.g., giving top priority to the most recent fringe entrant (stack), or the oldest (queue), leads to *depth first* and *breadth first search*, respectively.

Each call visits an entire *connected component* of the graph: those nodes accessible from the start node by going along edges. The algorithm traces out a *tree* made of graph edges and including every vertex (a *spanning tree*). If the graph is not *connected* (has >1 connected component), the algorithm will have to be called more than once. So we can use it to count connected components. If it ever examines a node that was visited earlier (not counting the immediately previous node), the graph has a *cycle*. On a graph with n nodes and e edges, the algorithm runs in worst case time O((n+e)log n).

We saw that running the algorithm on a tree gives the whole tree, which therefore has 1 less edge than the number of nodes. Depth-first search of a tree will list its nodes in some order; if we travel between them in that order and return to start, we cover each edge exactly twice.

A *complete* graph is one with all possible edges. A *Hamiltonian circuit* in a graph is a cycle visiting all nodes. The problem of whether a given graph has such a circuit (Hamiltonian circuit problem, or HCP) is hard. Exhaustive search can be used; there is no known polynomial time algorithm (i.e., one running in time $O(n^k)$ in the worst case, for some k) to detect whether a graph has such a cycle.

§8: In a weighted graph we attach a positive whole number (a weight, or length) to each edge. A common problem is to find a spanning tree of least possible total weight (a minimal spanning tree, or MST). We showed that the algorithm above will produce a MST if at each stage we always choose the fringe node closest to the visited nodes. We can use a similar method to find the shortest path between two nodes.

Given a weighted graph and a bound d, the travelling salesman problem (TSP) asks if there's a Hamiltonian circuit in the graph of total weight ≤d. This problem has many applications, but is hard. The position is similar to HCP.

We can reduce HCP to TSP rapidly (with worst case time function of the order of a polynomial). Any putative fast solution to TSP could then be used to give a fast solution to HCP. In fact one can also reduce TSP to HCP in p-time, so HCP and TSP are about equally hard. They are 'NP-complete' (see part III). Currently they are intractable, and most expect them to remain so.