that it works, note first that `#`xs really is a recursion variant. The recursion is unusual in that the recursive calls work not on xs, but on ys and zs. However, we know that these are strictly shorter than $x{:}xs$. For instance, `#`$ys \leq$ (`#`ys `++` `#`zs) = `#`$xs <$ `#`$x{:}xs$ so we do have a recursion variant.

Proposition 12.1 `qsort` terminates and satisfies its specification.

Proof `qsort []` clearly works correctly.

Now consider `qsort` $x{:}xs$. The result, (`qsort` ys) `++ [`x`]++` (`qsort` zs), is sorted, because

- (`qsort` ys) is sorted (the recursive calls can be assumed to be satisfactory);
- (`qsort` ys) is a permutation of ys, and by the specification of `Partition` every element of ys is $\leq x$, so every element of (`qsort` ys) is $\leq x$;
- similarly, (`qsort` zs) is sorted and all its elements are $> x$.

Also, it is a permutation of ys`++[`x`]++`zs, hence of $x{:}(ys$`++`$zs)$, and hence (by specification of partition) of $x{:}xs$. □

12.3 Arrays as lists

Miranda is a much simpler notation than Modula-2 and it is often helpful to be able to reason first in terms of Miranda and then transfer the reasoning to Modula-2. The most important properties of an array are its elements, together with their order: in other words, abstractly, the list or sequence of elements. For example, suppose we have

```
A: ARRAY [La..Ha] OF REAL;
B: ARRAY [Lb..Hb] OF REAL;
```

Then

A represents the list $[A[La], A[La+1], \ldots, A[Ha]]$
B represents the list $[B[Lb], B[Lb+1], \ldots, B[Hb]]$
A`++`B represents the list $[A[La], \ldots, A[Ha], B[Lb], \ldots, B[Hb]]$

(Note how we can sensibly talk about the append A`++`B, even though in Modula-2 it is quite difficult to construct it.) Also, `hd`$(A) = A$`[`La`]`, `hd`$(B) = B$`[`Lb`]`.

For computing purposes, we must also know how the elements are subscripted: hence the need for the bounds in the declarations. But the numerical values of the subscripts may be quite irrelevant to our original problem, and are just a computational necessity forced on us by the way Modula-2 accesses arrays. Then it is better to try to reason without them as much as possible — in fact, specifications that put too great a reliance on subscripts are said to suffer from 'indexitis'.

That said, we can of course put a subscript structure onto a list and thus treat it as an array. The conventional way in both Modula-2 (for open array parameters) and Miranda (for the ! operation) is to say that the first element has subscript 0. Thus a Miranda list

$$t = [t!0, t!1, \ldots, t!(\#t - 1)]$$

can be understood as an array with bounds [0..(# $t-1$)]. (But of course you cannot assign to the elements in Miranda.)

Let us also introduce some notation — *not* part of Modula-2 — for sublists. Suppose A has been declared as $A : ARRAY[m..n]OF\ldots$. We write A[i to j] to mean, essentially, the list

$$[[A[i], A[i+1], \ldots, A[j]]$$

This is provided that $m \le i \le j \le n$. It is also useful to define A[i to j] to be empty if $j < i$. Recursively,

```
A[i to j] = [],                 if j<i
          = A[i]:A[i+1 to j],   otherwise
```

Some properties of this notation are

- As lists, $A = A[m \text{ to } n]$.
- $A[i \text{ to } j]$ is defined iff $(i > j) \vee (m \le i \le j \le n)$. Use induction on $j - i$.
- If $A[i \text{ to } j]$ is defined and non-empty, then its length is $j - i + 1$.
- If $m \le i \le j \le k \le n$ then $A[i \text{ to } k] = A[i \text{ to } j-1]$++$A[j \text{ to } k]$ $= A[i \text{ to } j]$++$A[j+1 \text{ to } k]$.
- $A[i \text{ to } i] = [A[i]]$.
- $A[0 \text{ to HIGH}(A)] = A$.

12.4 Quick sort in Modula-2

A Miranda version would waste space by creating lots of new lists all the time. In Modula-2, with arrays (of INTEGERs, say), we can instead try for an *in-place* sort, rearranging the elements within the original array. The recursive calls of `qsort` will now work on regions within the original array, so the procedure must have extra parameters to specify the region. Let us say that `QuickSort`(*A, Start, Rest*) is to sort *A*[*Start* to *Rest*-1]. (The -1 lets us specify empty regions by taking *Start*=*Rest*, even if *Start* = 0.)

```
PROCEDURE QuickSort (VAR A: ARRAY OF INTEGER; Start,Rest: CARDINAL);
(*pre: 0<= Start <= Rest <= HIGH(A)+1
 *post: Perm(A, A_0)
 *      & A[0 to Start-1] = A_0[0 to Start-1]
 *      & A[Rest to HIGH(A)] = A_0[Rest to HIGH(A)]
 *      & Sorted(A[Start to Rest-1])
 *)
```

`Partition` will also work in-place:

```
PROCEDURE Partition (VAR A: ARRAY OF INTEGER;
            Start, Rest: CARDINAL;
                    K: INTEGER): CARDINAL;
(*pre: 0<= Start <= Rest <= HIGH(A)+1
 *post: Start <= result <= Rest
 *      & Perm(A, A_0)
 *      & A[0 to Start-1] = A_0[0 to Start-1]
 *      & A[Rest to HIGH(A)] = A_0[Rest to HIGH(A)]
 *      & (A[Start to result-1], A[result to Rest-1])
 *        satisfies the Miranda specification for
 *       (partition K   A_0[Start to Rest-1])
 *)
```

Let us leave the implementation of `Partition` until after the Dutch national flag problem; but note that we do not need `Partition` to compute the same function as `partition` (which, in any event, we have not defined yet), only to satisfy its specification.

The functional `qsort` can now be translated into Modula-2, using a call of `Partition`:

```
PROCEDURE QuickSort (* specified above *);
(* recursion variant = Rest-Start *)
 VAR n: CARDINAL;
     x: INTEGER;
BEGIN
  IF Start < Rest
  (* region is nonempty,
   *(qsort A_0[Start to Rest-1]) =
   *       (qsort ys)++[A_0[Start]]++(qsort zs)
   * where (ys,zs) =
   *       partition A_0[Start] A_0[Start+1 to Rest-1]-from Miranda
   *)
  THEN
    n:=Partition(A, Start+1, Rest, A[Start]);
    Swap(A[Start], A[n-1]);
    (*(A[Start to n-2], A[n to Rest-1])
     *is a valid result for
     *(partition A_0[Start] A_0[Start+1 to Rest-1])
     *)
    QuickSort(A, Start, n-1);
    QuickSort(A, n, Rest)
(*ELSE Start = Rest, region is empty. *)
  END
END QuickSort;
```

This really is just a translation of the Miranda `qsort`, though you might not think it at first glance. (That is why we suggest that it is a good idea to see the algorithm clearly in Miranda first.)

Think of the call `qsort`*(x:xs)*. In the Modula-2 context, x is A_0`[`*Start*`]`, and xs is A_0`[`*Start+1* to *Rest-1*`]`. After `n := Partition(...)` we have a satisfactory result for `Partition` x xs, namely (A`[`*Start+1* to n-1`]`, A`[`n to *Rest-1*`]`) (= (ys_1,zs_1), say). However, this is not quite yet the *(ys,zs)* that is used for the recursive call of `qsort`, the reason being that we want x in the middle instead of at the left-hand end where it is at the moment. What we do next is to swap A`[`*Start*`]` with A`[`n-1`]`, so that instead of having the equivalent of `[`x`]++` ys_1 `++` zs_1 we have ys`++ [`x`]++` zs where ys is a permutation of ys_1 — its last element has been moved to the head — and zs is just zs_1 renamed. *(ys,zs)* is still a satisfactory result for `Partition` x xs, so if we apply the sorting algorithm recursively in place to ys and to zs we obtain (`qsort` ys) `++ [`x`] ++` (`qsort` zs), as required.

12.5 Dutch national flag

This algorithm, due to the Dutch computer scientist Dijkstra, solves the sorting problem in the very simple case where there are only three possible values of the elements to be sorted.

The Dutch national flag is a tricolour, red (at the top), white and blue (Figure 12.2). For the problem imagine that (a computer representation of)

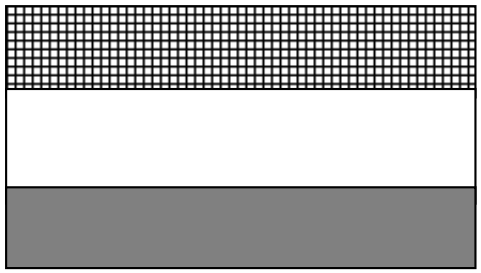

Figure 12.2

the flag gets scrambled (Figure 12.3), the stripes being cut up horizontally and rearranged: It is desired to correct this in one pass, that is, inspecting each stripelet once only. We are not told whether the three stripes are cut into the same number of stripelets. The only permitted way of rearranging stripelets is by swapping them, two at a time:

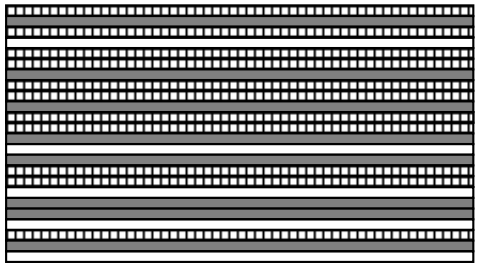

Figure 12.3

```
TYPE Colour = (red, white, blue);

PROCEDURE Restore (VAR A: ARRAY OF Colour);
(*pre: none
 *post:  Perm(A, A_0) & Sorted(A)
 *)
```

The idea is to track through the stripelets and put each one in 'the right area'. Part way through, let us have the stripelets arranged as in Figure 12.4. We shall need to keep pointers to the boundaries between the four areas.

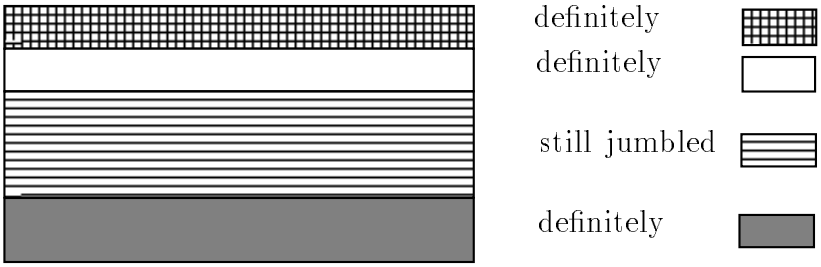

Figure 12.4

At each iteration, we inspect the first, that is, the top, grey (uninspected) stripelet. If it is *white*, then it is already in the right place and we can move on. If it is *red*, then we swap it with the first white and move on. If it is *blue*, then we swap it with the last grey before the blues but do not move on because we have now fetched another grey to inspect. Finally, when there are no greys left, then the stripelets are in the right order.

If we invent names for the pointers, then we can improve the diagram (Figure 12.5). We have adopted a convention here: there are three boundaries to be marked (red–white, white–grey and grey–blue), and the corresponding variable is always the index of the element just *after* the boundary. If two adjacent markers are equal, it shows that that region is empty. In particular, when *GreyStart*=*BlueStart* then there are no greys left and the flag is in order.

This diagram is essentially the loop invariant. At the appropriate points in the computation, we can imagine freezing the computer, inspecting the

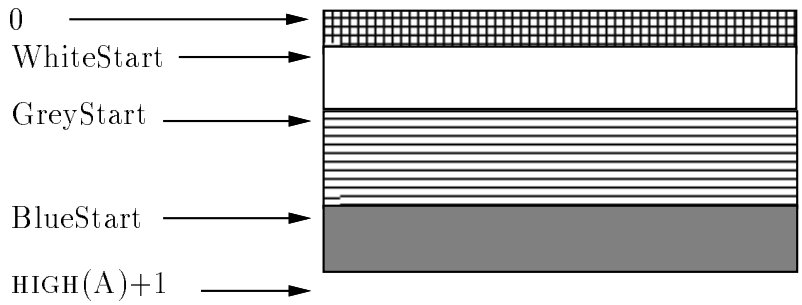

Figure 12.5

variables and the array, and asking whether the stripelets from, for instance *WhiteStart* to *GreyStart*−1, are indeed all white, as the diagram suggests. In other words, the diagram suggests a statement about the computer's state, and our next task is to translate this into logic as the invariant. You will see this in the implementation.

The variant, which is a measure of the amount of work left to be done, is the size of the jumbled (grey) area: *BlueStart*−*GreyStart*. Progress is made by reducing it:

```
PROCEDURE Restore(VAR A: ARRAY OF Colour);
VAR WhiteStart,GreyStart,BlueStart: CARDINAL;
BEGIN
  WhiteStart := 0; GreyStart := 0;
  BlueStart := HIGH(A)+1;
(* loop invariant:
 *   Perm(A,A_0)
 *   & WhiteStart <= GreyStart <= BlueStart <= HIGH(A)+1
 *   & (A)i:nat. ((0 <= i < WhiteStart -> A[i] = red)
 *   & (WhiteStart <= i < GreyStart -> A[i] = white)
 *   & (BlueStart <= i <= HIGH(A) -> A[i] = blue))
 * variant = BlueStart-GreyStart
 *)
   WHILE GreyStart < BlueStart DO
     CASE A[GreyStart] OF
        red:   Swap(A[WhiteStart],A[GreyStart]);
               WhiteStart := WhiteStart+1;
               GreyStart := GreyStart+1
       |white: GreyStart := GreyStart+1
       |blue:  Swap(A[GreyStart],A[BlueStart-1]);
               BlueStart := BlueStart-1
      END
    END
END Restore;
```

Sample reasoning

Let us us look at just two examples of how to verify parts of the procedure. First, why is `Perm` (A, A_0) always true? This is because all we ever do to the array is swap pairs of its elements, and a sequence of swaps is a permutation.

Next, why does the *red* part of the `CASE` statement reestablish the invariant? Let us write WS, GS, BS and A_1 for the values of *WhiteStart*, *GreyStart*, *BlueStart* and A when the label *red* is reached.

We know that $0 \leq WS \leq GS < BS \leq \texttt{HIGH}(A) + 1$, so after the update, when $WhiteStart = WS + 1$, $GreyStart = GS + 1$ and $BlueStart = BS$, we have $0 \leq WhiteStart \leq GreyStart \leq BlueStart \leq \texttt{HIGH}(A) + 1$, as required. To check that the colours are correct after the update, let i be a natural number.

If $0 \leq i < WhiteStart$, then $0 \leq i \leq WS$. We must show that $A\texttt{[}i\texttt{]}$ is *red*. If $i = WS$, this follows from the specification of `Swap`:

- $A\texttt{[}WS\texttt{]} = A_1\texttt{[}GS\texttt{]} = red$ by the `CASE` switch.
- If $i < WS$, which is $\leq GS$, then i is neither WS nor GS. Hence $A\texttt{[}i\texttt{]}$ was unaffected by the `Swap`, so $A\texttt{[}i\texttt{]} = A_1\texttt{[}i\texttt{]} = red$ by the loop invariant.

Next, suppose $WhiteStart \leq i < GreyStart$, that is, $WS + 1 \leq i \leq GS$. Note in this case that $A_1\texttt{[}WS\texttt{]} = white$ by the loop invariant, for $WS < WS + 1 \leq GS$. (The point is that the situation where $WS = GS$, and so $A_1\texttt{[}WS\texttt{]}$ is *grey*, is impossible given the i that we are considering.) Hence for $i = GS$, the specification of `Swap` tells us that $A\texttt{[}GS\texttt{]} = A_1\texttt{[}WS\texttt{]} = white$.

If $i < GS$, then i is neither WS nor GS. Hence from the specification of *Swap*, $A\texttt{[}i\texttt{]}$ is unchanged, and by the loop invariant it was white. For the third case, take $BlueStart \leq i \leq \texttt{HIGH}(A)$. Again, $A\texttt{[}i\texttt{]}$ is unchanged, and by the loop invariant it was *blue*.

12.6 Partitions by the Dutch national flag algorithm

Suppose, given a key integer K, you think of all integers as being coloured:
integers $< K$ are *red*
K is *white*
integers $> K$ are *blue*
Then the Dutch national flag algorithm, applied to an integer array, can do a crude sort. The *white* region is likely to be small or non-existent, so it is reasonable to merge it with the *red* region to make *pink*. The two regions correspond to those that `Partition` discovers: one for $\leq K$, one for $> K$.

We can therefore implement `Partition` by simplifying the Dutch national flag algorithm to cope with the flag of the Royal College of Midwives (pink and blue stripes). `QuickSort` will then look as in Figure 12.6, with recursive calls to sort the pink and blue regions. To implement `Partition` by adapting

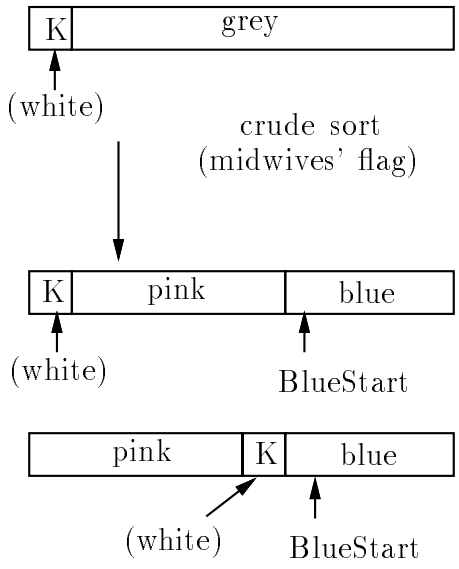

Figure 12.6

the `Partition` from the Dutch national flag, we must:

1. Simplify `Restore` to do the Midwives' sort (drop the 'red' case and *WhiteStart*; we can also turn the `CASE` statement to an `IF` statement).
2. Return the final *BlueStart* as a result in order to show the boundary of the partition.
3. Convert the colours to arithmetic inequalities ($\leq$ or $>$ the key K).
4. Allow for partitioning regions, rather than the whole array.

There should be no need to reason that the implementation is correct because we have done all the reasoning for `Restore`. But the loop invariant allows us to check, in case of doubt:

```
PROCEDURE Partition(VAR A: ARRAY OF INTEGER; Start, Rest: CARDINAL;
                    K: INTEGER): CARDINAL;
(*specification as before *)
VAR GreyStart, BlueStart:  CARDINAL;
  x: INTEGER;
BEGIN
  GreyStart := Start; (* no pinks *)
  BlueStart := Rest; (* no blues *)
(* loop invariant:
 * Perm(A, A_0)
 * & Start <= GreyStart <= BlueStart <= Rest
 * & (A)i:nat.
 * ((Start <=i< GreyStart -> A[i]<=K)
```

```
 * &(BlueStart <=i< Rest -> A[i]>K))
 * variant = BlueStart-GreyStart
 *)
  WHILE GreyStart < BlueStart DO
    IF A[GreyStart]<=K(*pink*)
    THEN GreyStart := GreyStart+1
    ELSE
      x:=A[GreyStart];
      A[GreyStart] := A[BlueStart-1];
      A[BlueStart-1] := x;
      BlueStart := BlueStart-1
    END
  END;
  RETURN BlueStart
END Partition;
```

12.7 Summary

- Functional definitions can be useful reasoning tools even if the final implementation is to be imperative.
- Sometimes a diagram is the real loop invariant.
- The method of introducing logical constants to name the values of computer variables is often (as in `Restore`) indispensable when you show that the loop body reestablishes the invariant.

12.8 Exercises

1. For the Dutch national flag algorithm show the following:
   (a) the invariant is established by the initialization;
   (b) the invariant is reestablished by each iteration (that is, do the *blue* and *white* cases corresponding to the *red* case above);
   (c) when looping stops, the post-condition has been set up;
   (d) the variant strictly decreases on each iteration, but never goes negative;
   (e) for every array access or `Swap`, the subscripts are within bounds (that is, $\leq$ `HIGH`(A)).
2. Consider the following idea for the Dutch national flag problem. The

white stripelets are to be put at the other end of the grey area:

```
[ Red     |Grey      |White      |Blue ]
          ↑          ↑           ↑
          GreyStart  WhiteStart  BlueStart
```

(a) Show that this is unsatisfactory for two reasons:

- on average, more swaps are done than are necessary;
- this method can give wrong answers.

(b) Two other sequences of two swaps are possible; is either of them correct?

3. Can the Dutch national flag method be generalized to work with more than three colours?
4. Implement `partition` in Miranda.
5. Modify the Miranda `partition` and `qsort` so that the order relation used does not have to be $\leq$, but is supplied as a parameter *lte*, a 'comparison function' which takes two elements as arguments and gives a Boolean result:

```
partition1::(*->*-> bool)->* ->[*]->([*],[*])
qsort1::(*->*-> bool)->[*]->[*]
```

(The comparison function can be thought of as a two-place predicate, or as a relation.) Give implementations for these, ensuring that `qsort` and `partition` are (`qsort1` ($\leq$)) and (`partition1` ($\leq$)). To obtain a *downward* ordered list, you would use (`qsort1`($\geq$)).

Chapter 13

Warshall's algorithm

Warshall's algorithm is an example of an algorithm that is difficult to understand at all without some kind of reasoning based on a loop invariant. The problem is to find the transitive closure of a relation. We shall first look at an algorithm that is relatively clear, and then go on to one (Warshall's algorithm) that is clever, and more efficient, but more difficult to understand.

13.1 Transitive closure

Warshall's algorithm computes transitive closures, a notion that comes from the theory of *relations*. To keep the discussion here simple, we shall explain this in terms of graphs, such as the one in Figure 13.1. A graph has a

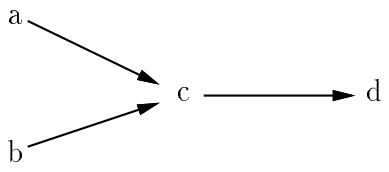

Figure 13.1

number of *nodes* (a, b, c and d here), and some *edges* (the arrows). In the sort of graph that we shall be using, for any pair (x, y) of nodes, there will be at most one edge from x to y (but possibly also one from y to x). Let us write "$x \to y$" if there is an edge from x to y. In our example,

$$a \to c,\ b \to c \text{ and } c \to d$$

but not $a \to b,\ a \to a,\ c \to b$, nor $a \to d$.

We shall interest ourselves in the problem of finding composite *paths* through the graph, made by joining edges up, head to tail, like elephants on parade.

Let us write "$x \to^+ y$" if there is a path from x to y; so here we have

$$a \to^+ c,\ b \to^+ c,\ c \to^+ d,\ a \to^+ d \text{ and } b \to^+ d$$

but not $a \to^+ b$, $a \to^+ a$ or $c \to^+ b$.

Formally, $x \to^+ y$ iff we can find a sequence $z_1, \ldots, z_n$ with

$$x \to z_1 \to \ldots \to z_n = y$$

$\to^+$ is the *transitive closure* of $\to$.

The *length* of the path is the number of *edges*, which is n here. We write $x \to^r y$ if there is a path of length r from x to y. Then

- $x \to^+ y$ iff $\exists n$:nat. $(1 \leq n \wedge x \to^n y)$
- $x \to y$ iff $x \to^1 y$

The following are some applications of finding the transitive closure:

- Suppose the nodes and edges represent airports and direct air flights. The paths are composite trips that can be made by plane alone.
- Suppose that nodes represent procedures in some program, and an edge from a to b means that a *calls* b. Then a path from a to b means that a calls b, though possibly indirectly (via some other procedures). A path from a to *itself* shows that a is potentially recursive. It may be useful for a compiler to be able to discover this because non-recursive procedures can be optimized to store return addresses, parameters and local variables in fixed locations instead of on a stack.

Computer representation

The graph can also be thought of as a *matrix*, or *array*, and this is the basis of the computer representation. If you give each node a number, then the whereabouts of the edges can be described by a square array of Boolean values:

$$Edge[a,b] = \begin{cases} \text{TRUE} & \text{if there is an edge from } a \text{ to } b, \text{ that is, } a \to b \\ \text{FALSE} & \text{otherwise} \end{cases}$$

This array, or matrix, is called the *adjacency matrix* of the graph. The transitive closure can be described the same way:

$$Path[a,b] = \begin{cases} \text{TRUE} & \text{if there is a path from } a \text{ to } b, \text{ that is, } a \to^+ b \\ \text{FALSE} & \text{otherwise} \end{cases}$$

Let us give some suitable declarations, and also specify the transitive closure procedure:

```
CONST Size = ...;(*number of nodes*)

TYPE
  Node = 1..Size;
  AdjMatrix = ARRAY Node,Node OF BOOLEAN;

PROCEDURE TransClos(Edge: AdjMatrix; VAR Path: AdjMatrix);
(*pre: none
 *post: Path represents transitive closure of Edge
 *)
```

You might decide to have `Edge` a `VAR` parameter, to avoid any possible copying. Then you would need a pre-condition to say that `Edge` and `Path` are different arrays, and an extra post-condition to say that `Edge` = $\texttt{Edge}_0$.

13.2 First algorithm

We shall look at three algorithms, and all of them will use the same basic idea. Some paths are more complicated than others; the simplest ones are the single edges, and they can be put together to make more complicated ones. The loop invariant will always say 'the TRUE entries in *Path* all represent paths, and all paths up to a certain degree of complication have been registered as TRUEs in *Path*'. More formally,

$$\forall a, b : Node.\ ((Path[a,b] \rightarrow (a \rightarrow^+ b))$$
$$\wedge((a \rightarrow^+ b) \text{ by a path of degree of complication} \leq N \rightarrow Path[a,b]))$$

The invariant will always be established initially by copying *Edge* to *Path* (thus registering the simplest paths), and each algorithm terminates when the degree of complication is sufficient to cover all possible paths. One difference between the algorithms lies in the measure of complication.

For the first two algorithms, we equate complication of a path with its *length.*

Suppose *Path* has registered all the paths of length $\leq n$, and we now want to find all paths of length $\leq n+1$: the new ones that we must find are those of length exactly $n+1$. But such a path from a to b splits up as a path of length n (from a to c_n, say), which is already registered in *Path*, and then an edge from c_n to b. Hence we shall be able to recognize it by the fact that $Path[a, c_n] = Edge[c_n, b] = \text{TRUE}$. Our method is to look at all possible combinations for a, b and c, and assign TRUE to $Path[a,b]$ if either it was TRUE already or we have $Path[a,c] = Edge[c,b] = \text{TRUE}$.

Paths can be of arbitrary length, so we must find a way of stopping. Actually, we can stop when we have registered all paths of length $\leq Size$, for longer ones do not tell us anything new. To see this, suppose we have a path

from a to b of length $n > Size$:

$$c_0 \to c_1 \to \ldots \to c_{n-1} \to c_n \quad \text{where } c_0 = a \text{ and } c_n = b$$

Consider $c_0, \ldots, c_n$. There are at least $Size + 1$ of these symbols, but there are only $Size$ possible nodes. Therefore, one node appears twice – $c_i = c_j$ where $i < j$. But this path can now be collapsed to a shorter path from a to b:

$$a \to c_1 \to \ldots \to c_i = c_j \to \ldots \to c_n = b$$

(See Exercise 1 for a more rigorous induction proof.)

Detailed reasoning

initialization: This follows because $a \to^1 b$ iff $Edge[a, b]$.

finalization: This follows because at the end $N = Size$, and $a \to^+ b$ iff $a \to^r b$ for some $r \leq Size$, as reasoned above.

reestablishing the invariant: Let us split the invariant I_1 into two parts:

$$I_1 1 \equiv_{\text{def}} \forall a, b : Node.\ (Path[a, b] \to (a \to^+ b))$$

$$I_1 2 \equiv_{\text{def}} \forall a, b : Node.\ (\forall r : nat.\ (a \to^r b) \wedge 1 \leq r \leq N \to Path[a, b])$$

The first thing to notice is that nothing ever spoils the truth of $I_1 1$. In particular, suppose it holds just before the assignment in the `FOR` loop. The only possible change is if $Path[i, j]$ becomes TRUE because we already have $Path[i, k]$ and $Edge[k, j]$; but then from $I_1 1$ we know $(i \to^+ k)$ and $(k \to j)$, so $(i \to^+ j)$, as required, and $I_1 1$ still holds afterwards. Hence $I_1 1$ holds right through the program.

Turning to $I_1 2$, this involves N so we must take care to allow for the increment $N := N + 1$. Let us write N_1 for the old value of N; after the increment, $N = N_1 + 1$. Before the `FOR` loops, $I2$ told us that if $a \to^r b$ with $r \leq N_1$ then $Path[a, b]$ and this much is never spoiled because $Path[a, b]$ never changes from TRUE to FALSE. Now suppose afterwards that $a \to^{N_1+1} b$, so there is a path of length $N_1 + 1$ from a to b. The last step of this path goes from c (say) to b, so we know $a \to^{N_1} c$ and $Edge[c, b]$; by the previous invariant we know $Path[a, c]$. Now consider the `FOR` loop iteration when $i = a$, $j = b$ and $k = c$: because $Path[a, c] = Edge[c, b] =$ TRUE, this sets $Path[a, b]$ to TRUE and it stays TRUE for ever, as required.

This is a good example of the reasoning style for `FOR` loops that was suggested in Section 10.6

Implementation

```
PROCEDURE TransClos(Edge: AdjMatrix; VAR Path: AdjMatrix);
(*pre: none
 *post:     Path represents transitive closure of Edge
 *notation: write - a-> b  iff Edge[a,b] = true
 *          (there is an edge from a to b)
 *          a-> +b  iff a is related to b by the transitive closure
 *          of Edge (there is a path from a to b);
 *          a-> ^n   b  iff there is a path from a to b  of length n
 *)
 VAR N: CARDINAL;
   i,j,k: Node;
 BEGIN
   CopyAdjMatrix(Edge,Path);
   N:=1;
(*loop invariant - call it I1:
 *N<= Size
 * & (A)a,b:Node.
 *   ((Path[a,b]-> (a-> +b))
 * & (A)r:nat. ((a-> ^r b) & 1<=r<=N -> Path[a,b]))
 * variant = Size-N
 *)
   WHILE N < Size DO
     FOR i:=1 TO Size DO
       FOR j:=1 TO Size DO
         FOR k:=1 TO Size DO
           Path[i,j] := Path[i,j] OR (Path[i,k] AND Edge[k,j])
          (*NB Path[a,b] never changes from true to false *)
         END
       END
     END;
     N:=N+1
   END
 END TransClos;

PROCEDURE CopyAdjMatrix(From: AdjMatrix; VAR To: AdjMatrix);
(*pre: none
 *post:  From = To
 *)
 BEGIN
 (*exercise*)
 END CopyAdjMatrix;
```

Efficiency

There are four nested loops, controlled by N, i, j and k. Each is executed roughly *Size* times.

($Size - 1$ times for N, *Size* each for i, j, k. $Total = Size^4 - Size^3$.) Hence, the total number of iterations is of the order of $Size^4$. (For large graphs the $Size^3$ term is insignificant compared with $Size^4$.)

This measures the *complexity* of the algorithm. *Size* measures how big the problem is: so the execution time increases roughly as the fourth power of the size of the problem. Thus big problems (lots of nodes) will really take quite a long time. Can we improve on this?

The first improvement is obvious but good. Suppose all paths of length N or less are recorded in *Path*. Then any path of length $2 * N$ or less can be decomposed into two parts, each of length N or less: if $a \rightarrow^r b$ with $r \leq 2 * N$, then we can write $r = s + t$ with $s, t \leq N$, and $a \rightarrow^s c \rightarrow^t b$ for some node c. Therefore, we have already registered $Path[a, c] = Path[c, b] =$ TRUE.

By this means, we can double N at each stage (that is, replace the assignment $N := N + 1$ by $N := 2 * N$) by using the innermost statement

```
Path[i,j] := Path[i,j] OR (Path[i,k] AND Path[k,j])
```

The outermost (N) loop is now executed approximately $log_2 Size$ times, so the total number of iterations is of the order of $log_2 Size \times Size^3$. This is good. $log_2 Size$ increases much more slowly than *Size*. Can we do better still?

13.3 Warshall's algorithm

The path relation that we are building up is *transitive:*

$$\forall a, b, c : Node.\ ((a \rightarrow^+ c) \wedge (c \rightarrow^+ b) \rightarrow (a \rightarrow^+ b))$$

(This is proved by joining paths together.) One way of understanding Warshall's algorithm is through the idea that part way through the calculation, *Path* will not be completely transitive but will be 'partially' transitive in that only certain values of c, not too big, will work in the above formula:

$$\forall a, b, c : Node.\ (Path[a, c] \wedge Path[c, b] \wedge c \leq N \rightarrow Path[a, b])$$

Now suppose we have achieved this partial transitivity, and we have a path

$$a \rightarrow c_1 \rightarrow c_2 \rightarrow \ldots \rightarrow c_n \rightarrow b$$

The partial transitivity tells us that provided the nodes $c_1, \ldots, c_n$ (let us call these the *transit nodes* of the path, as distinct from the endpoints a and b) are all $\leq N$, then we have $Path[a, b]$.

This leads to a new idea of how complicated a path is:

A *simple* path is one whose transit nodes (no matter how many) are all small — they have numerically small codes.
A *complicated* path is one whose transit nodes (no matter how few) include big ones.

The simplest paths from a to b have no transit nodes at all: they are just edges $a \to b$.

The next simplest are the paths that use node 1 as a transit node. These are of the form $a \to 1 \to b$.

Next, with node 2 also as a transit node, we have the possible forms

$$a \to 2 \to b,\ a \to 1 \to 2 \to b,\ a \to 2 \to 1 \to b$$

We quantify this numerically by defining the *transit maximum* of a path to be the maximum numerical code of its transit nodes (or 0 if there are none). Let us write $a \to_N b$ if there is a path from a to b with transit maximum $\leq N$.

Suppose we have already determined where there are paths of transit maximum $\leq N$, in other words we have computed the relation $\to_N$. Any path from a to b of transit maximum $N+1$ must use node $N+1$ in transit, and by much the same argument as before we do not need to consider such paths that use node $N+1$ more than once in transit (find the first and last transit occurrences of $N+1$ and cut out all the path in between them). Then we have

$$a \to \ldots \to (N+1) \to \ldots \to b$$

where the two sections of this have transit maximum at most N and so have already been found. To reiterate, once we know about all the paths of transit maximum $\leq N$, then all the paths of transit maximum $N+1$ from a to b can be recognized by the pattern $a \to_N (N+1) \to_N b$, the two sections of this being paths that we already know about.

Detailed reasoning

initialization: This follows because $a \to_0 b$ iff $a \to b$.

finalization: Because $a \to_{Size} b$ iff $a \to^+ b$.

reestablishing the invariant: Let N_1 be the value of N before the increment, and let J be the following, which follows from the invariant I_2:

$$\forall a, b : Node.\ (Path[a,b] \to (a \to^+ b)) \wedge ((a \to_{N_1} b \to Path[a,b]))$$

No iteration of the `FOR` loops ever spoils the truth of J so it is still true after the `FOR` loops. However, the invariant will say something stronger than J because of the increment of N, and we must check this.

Suppose $a \rightarrow_N b$, so there is a path from a to b with transit maximum $\leq N$ (which is now $N_1 + 1$). If all its transit nodes are actually $\leq N_1$, then $a \rightarrow_{N_1} b$ and so by J we know $Path[a, b]$. The only remaining case is when some transit node is equal to N. Then by splitting up the path we see that $a \rightarrow_{N_1} N \rightarrow_{N_1} b$, so by J we know that $Path[a, N]$ and $Path[N, b]$. The `FOR` loop iteration when $i = a$ and $j = b$ makes $Path[a, b]$ equal to TRUE and it remains so for ever.

Implementation

```
PROCEDURE TransClos(Edge: AdjMatrix; VAR Path: AdjMatrix);
(*pre: none
 *post:  Path represents transitive closure of Edge
 *notation: a-> n   b  means there is some path
 *          a-> c1-> c2-> ... -> cr -> b(r>=0)
 *       where c1, ... , cr are all <=n,i.e. its transit maximum is <=n.
 *          Hence a-> +b  iff a-> Size b.
 *)
VAR N: CARDINAL;
    i,j: Node;
BEGIN
  CopyAdjMatrix(Path,Edge);
  N:=0;
(*loop invariant I2:
 * N<= Size
 * & (A)a,b:Node.
 *   ((Path[a,b]-> (a-> +b))
 * & ((a-> N b)-> Path[a,b]))
 *variant = Size-N
 *)
   WHILE N < Size DO
     N:=N+1;
     FOR i:=1 TO Size DO
       FOR j:=1 TO Size DO
         Path[i,j] := Path[i,j] OR (Path[i,N] AND Path[N,j])
       END
     END
   END
END TransClos;
```

Efficiency

There are now three nested loops (for N, i and j), each one being executed *Size* times, so the total number of iterations is of the order of $Size^3$. This is the best of our three algorithms.

We could optimize this further. For instance, we could replace the `FOR` loops by

```
FOR i:=1 TO Size DO
  IF Path[i,N]
  THEN
    FOR j:=1 TO N  DO
      Path[i,j] := Path[i,j] OR Path[N,j]
    END
   END
END
```

(QUESTION: can you prove that this has the same result as the preceding version?) However, this is local fine tuning. The step from the original version to Warshall's was a fundamental change of algorithm, with a new Invariant.

13.4 Summary

- We have given three algorithms to compute transitive closures, each one fundamentally more efficient than the previous one.
- The most efficient is Warshall's algorithm. It would be difficult to see clearly why it works without the use of loop invariants.
- The reasoning about `FOR` loops was essentially different from the loop invariant technique used for `WHILE` loops.

13.5 Exercises

1. Given a graph with *Size* nodes, show that for any nodes a and b, if $a \rightarrow^+ b$ then $a \rightarrow^r b$ for some $r \leq Size$. HINT: use course of values induction on n to show $\forall n : nat.\ P(n)$, where

$$P(n) \equiv (a \rightarrow^n b) \rightarrow \exists r : nat.\ (r \leq Size \wedge (a \rightarrow^r b)).$$

2. Use Warshall's algorithm 'in place' to implement the following procedure (without using any array other than *Graph*):

```
PROCEDURE TransClos(VAR Graph: AdjMatrix);
(*pre: none
 *post: Graph represents transitive closure of Graph_0
 *)
```

3. Modify the detailed reasoning of the first algorithm to justify the second.
4. Warshall's algorithm can be modified to compute shortest paths between nodes in a graph. Here is the specification:

```
TYPE Matrix = ARRAY Node, Node OF CARDINAL;

PROCEDURE ShortPaths(Edge: AdjMatrix; VAR SP: Matrix);
 (*pre: none
  *post: (A)i,j:Node. (A)r:nat.
  *         (1 <= SP[i,j] <= Size+1
  *         & (SP[i,j] = r & r <= Size -> (i -> ^r j))
  *     & ((i -> ^r j) & r >= 1 -> SP[i,j] <= r))
  *)
```

The idea is that if there is any path at all from i to j then there is one of length *Size* or less, and $SP[i,j]$ is to be the shortest such length. If there is no path, then $SP[i,j]$ is to be $Size + 1$.
Show how to modify the invariant and code of Warshall's algorithm to solve this new problem. You will probably need to use the relation $\rightarrow_N^r$, defined by $(i \rightarrow_N^r j)$ iff there is a path of length r from i to j, with transit nodes all $\leq N$.