9 Relations

Introduction

In this chapter we will learn how to use Maple to work with relations. We explain how to use Maple to represent binary relations using sets of ordered pairs, zero-one matrices, and directed graphs. We show how to use Maple to determine whether a relation has various properties using these different representations.

We also describe how to compute closures of relations. In particular, we show how to find the transitive closure of a relation using two different algorithms and we compare the time complexity of these algorithms. After explaining how to use Maple to work with equivalence relations, we show how to use Maple to work with partial orderings and how to use Maple to draw Hasse diagrams and how to implement topological sorting.

9.1 Relations and Their Properties

The first step in understanding and manipulating relations in Maple is to determine how to represent them. There is no specific package in Maple designed to handle relations. We will implement relations in Maple using the most convenient form for the question at hand. In this chapter, we will make use of sets of ordered pairs, zero-one matrices, and directed graphs in order to explore relations in Maple.

Relations as Ordered Pairs

First, we will represent relations as sets of ordered pairs. We define our own type for relations, which we'll call **rel**. Our reason for defining a type is that it gives us a way to ensure that when arguments are passed to procedures we write, the arguments are valid for that procedure. In the past, we've made use of Maple's existing types for this purpose. As an illustration of the utility of types, consider the procedure below.

```
> myFactorial := proc(n::posint)
    if n = 1 then
      return 1;
    else
      return n * myFactorial(n-1);
    end if;
  end proc:
```

In this simple example, when we declare **myFactorial** to be a procedure that takes **n** as an input, we also specify that **n** is supposed to be of type **posint**, *i.e.*, a positive integer. That way, if we try to compute the factorial of -3:

```
> myFactorial(-3);
Error, invalid input: myFactorial expects its 1st argument,
n, to be of type posint, but received -3
```

an error is generated. It is usually better for a program to generate an error when it receives invalid input than to attempt to operate on the bad input value. In the case of **myFactorial**, omitting the type check from the procedure definition would result in an infinite recursion.

We could also deal with the problem of potentially invalid arguments by checking "by hand." That is, putting the type checking in the body of the procedure rather than as part of its declaration. But Maple's automatic type checking results in more readable, and often faster, code.

As mentioned, we are going to represent relations as sets of ordered pairs. We will define two types. First, an "ordered pair" type that we'll call **pair**. And then the relation type, which will be called **rel**,will be defined to be a set of objects of type **pair**. We define the **pair** type as follows:

```
> `type/pair` := [anything,anything];
```

$$type/pair := [anything, anything] \quad \textbf{(9.1)}$$

On the left side of the assignment we have `**type/**name` enclosed in left single quotes. The right side of the assignment uses Maple's structured type syntax to indicate that the type is a list (indicated by the brackets) of two objects which may be of any type (indicated by the use of the **anything** keyword).

The following defines the relation type.

```
> `type/rel` := set(pair);
```

$$type/rel := set(pair) \quad \textbf{(9.2)}$$

In this case we indicate that an object of type **rel** is a set of objects of type **pair**. (Note: you may have thought braces would be used to indicate a set, but braces have a different meaning in a structured type. Also note: we cannot use the word **relation** for this type, as that type is already used by the Maple library.)

Creating Relations

Now that we've established the relation type, let's create an actual relation.

The divides relation

Example 4 in Section 9.1 describes the "divides relation", *i.e.*, $R = \{(a, b) \mid a \text{ divides } b\}$. We will write a procedure to construct this relation. The procedure will consider every possible ordered pair of elements and will include them in the relation if they satisfy the condition that the remainder obtained by dividing b by a is 0 (which is equivalent to saying that the first element of the pair divides the second).

```
> DividesRelation := proc(A::set(integer))
    local a, b, temp, R;
    R := {};
    for a in A do
      for b in A do
        if (irem(b,a) = 0) then
          R := R union {[a,b]};
        end if;
      end do;
    end do;
    return R;
  end proc:
```

Note that in defining this procedure, we use an unnamed structured type, **set(integer)**, to ensure that only sets of integers may be passed to the procedure.

We use the procedure to construct the relation on the integers 1 through 4.

```
> DividesRelation({1, 2, 3, 4});
```

$$\{[1, 1], [1, 2], [1, 3], [1, 4], [2, 2], [2, 4], [3, 3], [4, 4]\} \quad \textbf{(9.3)}$$

We can check that this procedure has really produced a relation (*i.e.*, an object of type **rel**) by using the **type** command.

```
> type(%9.3%,rel);
```

true **(9.4)**

The **type** command takes two arguments. The first is any expression and the second is the name of a type. The command returns true if the expression is of the specified type and false otherwise.

For convenience, we also define the following procedure for constructing the "divides relation" on the set $\{1, 2, \ldots, n\}$ given any positive integer n.

```
> DivRel := proc(n::posint)
    local i;
    DividesRelation({seq(i,i=1..n)});
  end proc:
```

For example:

```
> Div6 := DivRel(6);
```

Div6 := {[1, 1], [1, 2], [1, 3], [1, 4], [1, 5], [1, 6], [2, 2], [2, 4], [2, 6], [3, 3], [3, 6], [4, 4], [5, 5], [6, 6]} **(9.5)**

The inverse of a relation

Now that we have seen an example of a procedure that creates a relation, let's look at a simple example of a procedure that manipulates a relation.

For any relation R, its inverse relation, denoted R^{-1} is defined by $R^{-1} = \{(b, a) \mid (a, b) \in R\}$. The following procedure computes the inverse of a relation.

```
> InverseRelation := proc(R::rel)
    local u;
    map(u -> [u[2],u[1]],R);
  end proc:
```

The **map** command, which we use above, applies the function specified in the first argument to each operand of the second argument. In this case, the second argument is a **rel** object, that is, a set of ordered pairs. So the operands of **R** are the ordered pairs. The function, **u -> [u[2],u[1]]**, takes a pair **u**, and reverses its elements.

Since we've defined the "divides" relation, we can use the **InverseRelation** procedure to create the "multiple of" relation.

```
> Mul6 := InverseRelation(Div6);
```

Mul6 := {[1, 1], [2, 1], [2, 2], [3, 1], [3, 3], [4, 1], [4, 2], [4, 4], [5, 1], [5, 5], [6, 1], [6, 2], [6, 3], [6, 6]} **(9.6)**

Properties of Relations

Maple can be used to determine if a relation has a particular property, such as reflexivity, symmetry, antisymmetry or transitivity. This can be accomplished by creating Maple procedures that take as input the given relation, examine the elements of the relation, and return true or false based on if the relation has the property or not.

Before writing procedures to test for properties of relations, it will be convenient to have a routine that extracts the domain of a given relation. This procedure works by simply collecting all of the elements that appear as either entry in a pair of the relation. (Note that, strictly speaking, this need not equal the domain of the relation, since there may exist elements in the domain that are not related to any object in the domain. It might be better to call this the "effective domain" of the relation.)

```
> FindDomain := proc(R::rel)
    return (map(op,R));
```

```
end proc:
```

We again use the **map** command in this procedure. In this case, **map** is being used to apply the function **op** to each pair of **R**. This has the effect of removing the list structure from the pairs in the relation and leaving just the elements. Since **R** is a set, the result of the **map** is a set, and so duplicates are automatically removed.

Reflexivity

Now we are ready to begin testing relations for various properties. The first property we'll consider is reflexivity. Remember that a relation R is reflexive if $(a, a) \in R$ for every a in the domain.

To check to see if a relation is reflexive, we'll compute the domain of the relation and then check each element a of the domain to see if (a, a) is in the relation. If the procedure finds an element of the domain with $(a, a) \notin R$, then it returns false immediately. If it checks all of the members of the domain with no failures, then it returns true.

```
> IsReflexive := proc(R::rel)
    local a;
    for a in FindDomain(R) do
      if not ([a,a] in R) then
        return false;
      end if;
    end do;
    return true;
  end proc:
```

We can use this on the "divides" relation.

```
> IsReflexive(Div6);
```

true **(9.7)**

Symmetry

Next we will examine the symmetric and antisymmetric properties. To determine whether a relation is symmetric, we simply use the definition. That is, we check, for every member (a, b) of R, whether (b, a) is also a member of the relation. If we discover a pair in the relation for which the reverse pair is not in the relation, then we know that the relation is not symmetric. Otherwise, it must be symmetric. This is the logic employed by the following procedure.

```
> IsSymmetric := proc(R::rel)
    local u;
    for u in R do
      if not ([u[2],u[1]] in R) then
        return false;
      end if;
    end do;
    return true;
  end proc:
```

For example, we can see that the "divides" relation is not symmetric.

```
> IsSymmetric(Div6);
```

false **(9.8)**

The union of "divides" and "multiple of" is symmetric, however.

```
> DivOrMul6 := Div6 union Mul6;
```

DivOrMul6 := {[1, 1], [1, 2], [1, 3], [1, 4], [1, 5], [1, 6], [2, 1], [2, 2], [2, 4], [2, 6], [3, 1], [3, 3], [3, 6], [4, 1], [4, 2], [4, 4], [5, 1], [5, 5], [6, 1], [6, 2], [6, 3], [6, **(9.9)**

```
    6]}
> IsSymmetric(DivOrMul6);
```

true **(9.10)**

To determine whether a given relation R is antisymmetric, we again use the definition. Remember that a relation is antisymmetric when it has the property that whenever a pair (a, b) and its reverse (b, a) both belong to R, then it must be that $a = b$. To check this, we simply loop over all members u of R and see if the opposite pair belongs to R and whether the members of the pair are different.

```
> IsAntisymmetric := proc(R::rel)
    local u;
    for u in R do
      if (([u[2],u[1]] in R) and (u[1] <> u[2])) then
        return false;
      end if
    end do;
    return true;
  end proc:
```

We now use this procedure to check to see if the "divides" and "multiple of" relations defined earlier are antisymmetric.

```
> IsAntisymmetric(Div6);
```

true **(9.11)**

```
> IsAntisymmetric(Mul6);
```

true **(9.12)**

Transitivity

The transitive property is the most difficult to check. Recall the definition of transitive relations: a relation R is transitive if, whenever (a, b) and (b, c) are in R, then (a, c) must be as well.

To check transitivity, we will consider all possible a, b, and c in the domain of R. Then if $(a, b) \in R$, $(b, c) \in R$, and $(a, c) \notin R$, we know that the relation is not transitive. If there is no such triple a, b, c to contradict transitivity, then we conclude that the relation is transitive.

Here is the procedure.

```
> IsTransitive := proc(R::rel)
    local DomR, a, b, c;
    DomR := FindDomain(R);
    for a in DomR do
      for b in DomR do
        for c in DomR do
          if (([a,b] in R) and ([b,c] in R)
                    and not([a,c] in R)) then
            return false;
          end if;
        end do;
      end do;
    end do;
    return true;
  end proc:
```

We see that the "divisible" relation is transitive. But we can cause it to fail to be transitive by removing the $(1, 6)$ pair, since $(1, 2)$ and $(2, 6)$ are in R.

```
> IsTransitive(Div6);
```

$$true \tag{9.13}$$

```
> R2 := Div6 minus {[1,6]};
```

$$R2 := \{[1,1],[1,2],[1,3],[1,4],[1,5],[2,2],[2,4],[2,6],[3,3],[3,6],[4,4],[5,5],[6,6]\} \tag{9.14}$$

```
> IsTransitive(R2);
```

$$false \tag{9.15}$$

9.2 *n*-ary Relations and Their Applications

Using Maple, we can construct an *n*-ary relation where *n* is a positive integer. As in the previous section, we will begin by defining types both for the elements of the relation (**tuple**) and for the *n*-ary relation (**nrel**). The only difference here, as compared to the types we defined in the previous section, is that we do not know the length of the list that makes up a **tuple**. So we make a generic list with the structured type syntax.

```
> `type/tuple` := list(anything);
```

$$type/tuple := list(anything) \tag{9.16}$$

```
> `type/nrel` := set(tuple);
```

$$type/nrel := set(tuple) \tag{9.17}$$

Consider the following 4-ary relation that represents student records.

```
> R3 := {["Adams", 9012345, "Politics", 2.98],
         ["Woo", 9100055, "Film Studies", 4.99],
         ["Warshall", 9354321, "Mathematics", 3.66]};
```

$$R3 := \{["Adams", 9012345, "Politics", 2.98], ["Woo", 9100055, "Film Studies", 4.99], ["Warshall", 9354321, "Mathematics", 3.66]\} \tag{9.18}$$

The first field represents the name of the student, the second field is the student ID number, the third field is the students' home department, and the last field stores the student's grade point average. Note that this relation is of type **nrel**.

```
> type(R3,nrel);
```

$$true \tag{9.19}$$

While we created a very generic *n*-ary relation type, you can also create more specific types for certain particular situations. For instance, the tuples in the relation above will always consist of a string, integer, string, and a floating point number. So we could make the following type specifically for that kind of relation.

```
> `type/studentrecord` := [string, integer, string, float];
```

$$type/studentrecord := [string, integer, string, float] \tag{9.20}$$

```
> `type/studentrel` := set(studentrecord);
```

$$type/studentrel := set(studentrecord) \tag{9.21}$$

```
> type(R3, studentrel);
```

$$true \tag{9.22}$$

Operations on *n*-ary Relations

Now we will create procedures that act on **nrel**s to compute projections and the join of relations.

Projection

We will construct a procedure for computing a projection of a relation. The procedure takes as input a **nrel** along with a list of integers representing the indices of the fields that are to remain. The output will be another **nrel**.

```
> ProjectRelation := proc(R::nrel, P::list(posint))
    local u, S;
    S := {};
    for u in R do
      S := S union {u[P]};
    end do;
    return S;
  end proc:
```

The expression **u[P]** returns the sublist of **u** consisting of the elements corresponding to the indices in the list **P**.

We can use this procedure with the relation we created earlier.

```
> ProjectRelation(R3, [2,4]);
```

$$\{[9012345, 2.98], [9100055, 4.99], [9354321, 3.66]\} \quad \textbf{(9.23)}$$

```
> ProjectRelation(R3, [3,4,1]);
```

$$\{["Film\ Studies", 4.99, "Woo"], ["Mathematics", 3.66, "Warshall"], ["Politics", 2.98, "Adams"]\} \quad \textbf{(9.24)}$$

Join

Now let's consider joins of relations. The join operation has applications to databases when tables of information need to be combined in a meaningful manner.

The join procedure that we will implement here follows the following outline.

1. Input two relations R and S and a positive integer p, representing the overlap between the relations.
2. Examine each element u of R and determine the last p fields of u.
3. Examine all elements v of S to determine if the first p fields of v match the last p fields of u.
4. Upon finding a match, we combine the elements and place the result in a relation T, which is returned as the output of the procedure.

```
> JoinRelation := proc(R::nrel, S::nrel, p::posint)
    local overlapR, overlapS, degreeR, i, u, isDone, x,
  joinElement, T;
    T := {};
    degreeR := nops(R[1]);
    # overlapR = indices for the last p entries in R elements
    overlapR := [seq(degreeR - p + i, i=1..p)];
    # overlapS = indices for the first p entries in S elements
    overlapS := [seq(i, i=1..p)];
    for u in R do
      # x = the part of u that is supposed to overlap
      x := u[overlapR];
      i := 1;
      # examine elements of S until either a match is found
      #  (indicated by isDone = true) or all elements are
      #  examined (tested by i > nops(S))
      isDone := false;
      while i <= nops(S) and isDone = false do
        if S[i][overlapS] = x then
```

```
            joinElement := [op(u), op(S[i][(p+1)..-1])];
            T := T union {joinElement};
            isDone := true;
          end if;
        i := i + 1;
        end do;
      end do;
      return T;
    end proc:
```

This procedure will be easier to understand with a simple example in hand.

```
> exampleR:={["a","A",1],["b","B",2],["c","C",3],["d","D",4]};
```

$$exampleR := \{["a", "A", 1], ["b", "B", 2], ["c", "C", 3], ["d", "D", 4]\} \quad \textbf{(9.25)}$$

```
> exampleS := {["A",1,"A1","a1"],["B",2,"B2","b2"],["D",4,"D4",
  "d4"]};
```

$$exampleS := \{["A", 1, "A1", "a1"], ["B", 2, "B2", "b2"], ["D", 4, "D4", "d4"]\} \quad \textbf{(9.26)}$$

The *R* relation elements consist of the lower case and upper case versions of letters and their position in the alphabet (up to "D"). The *S* relation consists of the capital letter, its position in the alphabet, and the two character strings combining letter and position for both upper and lower cases, but omitting the relation for "C".

The `JoinRelation` procedure beings by initializing the return relation, `T`, to the empty set. It then computes `degreeR`, the degree of the relation `R`, by accessing the first element of `R` and applying the `nops` command. In our example, this line would apply `nops` to the list `["a","A",1]` and thus set `degreeR` to `3`.

The next two lines set the lists `overlapS` and `overlapR`. These lists represent the indices of the overlapping elements in the `S` and `R` relations. The indices of the elements of `S` that overlap are merely `1` through `p`. For `R`, we need the last `p` indices, which are obtained by computing `degreeR - p + i` for `i` from `1` to `p`. For our example, the relations overlap in two fields, so `p` is `2`.

```
> overlapS := [1,2];
  overlapR := [2,3];
```

$$overlapS := [1, 2]$$

$$overlapR := [2, 3] \quad \textbf{(9.27)}$$

The `for` loop begins the heart of the procedure. For each element `u` of `R`, the variable `x` is set to the portion of `u` that may overlap with elements of `S`. The variable `i` is a counter that tracks the progress as it compares `x` to the elements of `S`. And the `isDone` variable is used to track whether a match has been found from `S`. It is initialized to false and is set to true if a match is found, thereby short-circuiting the `while` loop.

The `while` loop is used to check the elements of `S` one at a time until all of the elements of `S` have been checked or until a match has been found. The condition in the if statement takes the `i`th element of `S` (*i.e.*, `S[i]`) and looks at the part of that list that may overlap (namely, `S[i][overlapS]`). For example:

```
> exampleS[3];
```

$$["D", 4, "D4", "d4"] \quad \textbf{(9.28)}$$

```
> exampleS[3][overlapS];
```

$$["D", 4] \quad \textbf{(9.29)}$$

If this segment of `S` is equal to `x`, then a match has been found. The variable `joinElement` is set to the result of combining `u` (from `R`) with the element from `S` that matches. This is accomplished by taking all of `u`, using `op(u)`, together with the portion of the matching element from `S` beyond the overlap, with `op(S[i][(p+1)..-1])`.

The resulting `joinElement` is added to the `T` relation. Also, `isDone` is set to true which ends the while loop, so that the procedure goes on to the next element of `R`.

We conclude this section by applying the `JoinRelation` procedure to Example 11 of Section 9.2.

```
> TeachingAssignments := {["Cruz",    "Zoology",          335],
                          ["Cruz",    "Zoology",          412],
                          ["Farber",  "Psychology",       501],
                          ["Farber",  "Psychology",       617],
                          ["Grammer","Physics",           544],
                          ["Grammer","Physics",           551],
                          ["Rosen",   "Computer Science", 518],
                          ["Rosen",   "Mathematics",      575]};
```

TeachingAssignments := {["Cruz", "Zoology", 335], ["Cruz", "Zoology", 412], ["Farber", "Psychology", 501], ["Farber", "Psychology", 617], ["Grammer", "Physics", 544], ["Grammer", "Physics", 551], ["Rosen", "Mathematics", 575], ["Rosen", "Computer Science", 518]} **(9.30)**

```
> ClassSchedule :=
    {["Computer Science", 518, "N521", "2:00 P.M."],
     ["Mathematics",     575, "N502", "3:00 P.M."],
     ["Mathematics",     611, "N521", "4:00 P.M."],
     ["Physics",         544, "B505", "4:00 P.M."],
     ["Psychology",      501, "A100", "3:00 P.M."],
     ["Psychology",      617, "A110", "11:00 A.M."],
     ["Zoology",         335, "A100", "9:00 A.M."],
     ["Zoology",         412, "A100", "8:00 A.M."]};
```

ClassSchedule := {["Physics", 544, "B505", "4:00 P.M."], ["Zoology", 335, "A100", "9:00 A.M."], ["Zoology", 412, "A100", "8:00 A.M."], ["Mathematics", 575, "N502", "3:00 P.M."], ["Mathematics", 611, "N521", "4:00 P.M."], ["Psychology", 501, "A100", "3:00 P.M."], ["Psychology", 617, "A110", "11:00 A.M."], ["Computer Science", 518, "N521", "2:00 P.M."]} **(9.31)**

```
> TeachingSchedule := JoinRelation(TeachingAssignments,
  ClassSchedule, 2);
```

TeachingSchedule := {["Cruz", "Zoology", 335, "A100", "9:00 A.M."], ["Cruz", "Zoology", 412, "A100", "8:00 A.M."], ["Farber", "Psychology", 501, "A100", "3:00 P.M."], ["Farber", "Psychology", 617, "A110", "11:00 A.M."], ["Grammer", "Physics", 544, "B505", "4:00 P.M."], ["Rosen", "Mathematics", 575, "N502", "3:00 P.M."], ["Rosen", "Computer Science", 518, "N521", "2:00 P.M."]} **(9.32)**

We can make the result a little more readable with a simple loop.

```
> for u in TeachingSchedule do print(u) end do;
                ["Cruz", "Zoology", 335, "A100", "9:00 A.M."]
                ["Cruz", "Zoology", 412, "A100", "8:00 A.M."]
```

```
["Farber", "Psychology", 501, "A100", "3:00 P.M."]
["Farber", "Psychology", 617, "A110", "11:00 A.M."]
["Grammer", "Physics", 544, "B505", "4:00 P.M."]
["Rosen", "Mathematics", 575, "N502", "3:00 P.M."]
["Rosen", "Computer Science", 518, "N521", "2:00 P.M."]
```
(9.33)

9.3 Representing Relations

As was stated earlier in this chapter, Maple allows us to represent and manipulate relations in a variety of ways. From this point forward, we will consider exclusively binary relations. This gives us additional options for how we represent relations. In this section, we will see how to represent binary relations with zero-one matrices and digraphs.

Representing Relations Using Matrices

We begin with representations of relations with zero-one matrices.

A first example

We create a matrix with the **Matrix** command. To produce a square matrix, this is as simple as giving the Matrix command with a single argument — the dimension of the matrix.

```
> exampleMatrix := Matrix(4);
```

$$exampleMatrix := \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \textbf{(9.34)}$$

(For a non-square matrix, you would need two arguments for the number of rows and the number of columns.) Observe that Maple creates the matrix and puts zeros in all the entries by default.

Right now, this matrix doesn't represent a very interesting relation. We need to change entries to 1 to represent elements of the domain that are related to each other. For instance, if $(1, 2) \in R$ then we need to change the $(1, 2)$ entry to a 1. To do this, we enter the following.

```
> exampleMatrix[1,2] := 1;
```

$$exampleMatrix_{1,\,2} := 1 \quad \textbf{(9.35)}$$

We can see that it modified the matrix:

```
> exampleMatrix;
```

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \textbf{(9.36)}$$

Let's make this matrix represent the relation "is one less than" on $\{1, 2, 3, 4\}$, as in, "1 is one less than 2."

```
> exampleMatrix[2,3] := 1:
> exampleMatrix[3,4] := 1:
> exampleMatrix;
```

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{9.37}$$

Transforming a set of pairs representation into a matrix

Now we'll create a procedure to turn a relation of type **rel** (defined in the first section) into a matrix representation. Doing so is fairly straightforward. Given a relation *R*, whose domain consists of integers, we can use **FindDomain** from above to extract the domain. We then create a square matrix whose size is equal to the largest integer in the domain, which we can obtain with the **max** function. Then we simply have to loop through the elements of the relation and set the value of the corresponding entry in the matrix to 1.

```
> RelToMatrix := proc(R::rel)
    local u, M, domain;
    domain := FindDomain(R);
    M := Matrix(max(domain));
    for u in R do
      M[op(u)] := 1;
    end do;
    return M;
  end proc:
```

We use this procedure to convert the relations we defined earlier, specifically **Div6** and **DivOrMul6** into matrices.

```
> Div6M := RelToMatrix(Div6);
```

$$Div6M := \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{9.38}$$

```
> DivOrMul6M := RelToMatrix(DivOrMul6);
```

$$DivOrMul6M := \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \tag{9.39}$$

Now that we have zero-one matrix representations of relations to work with, we can use these matrices to determine which properties apply to them. In this form, it is sometimes easier to determine whether a relation is reflexive, symmetric, or antisymmetric.

Checking properties

For example, to determine whether or not a relation is reflexive from its zero-one matrix representation, we only need to check the diagonal entries. If any diagonal entry is 0, then the relation is not reflexive.

We begin the procedure by checking to make sure the matrix is square. For if not, there is no chance that the relation is reflexive, and, moreover, code later in the procedure will result in an error for non-square matrices. We use the **RowDimension** and **ColumnDimension** commands from the **LinearAlgebra** package to determine number of rows and columns of the matrix.

```
> IsReflexiveM := proc(M::Matrix)
    local i, numrows, numcols;
    numrows := LinearAlgebra[RowDimension](M);
    numcols := LinearAlgebra[ColumnDimension](M);
    if numrows <> numcols then
      return false;
    end if;
    for i from 1 to numrows do
      if M[i,i] = 0 then
        return false;
      end if;
    end do;
    return true;
  end proc:
```

We can now use this to test a few of the relations above.

```
> IsReflexiveM(exampleMatrix);
```

false **(9.40)**

```
> IsReflexiveM(Div6M);
```

true **(9.41)**

Symmetry is particularly easy to test, because of the fact that a relation is symmetric if and only if its matrix representation is symmetric. Recall from Chapter 2 of the textbook that a matrix is symmetric when it is equal to its transpose. So all we need to do is compute the transpose of the matrix, using the **Transpose** command in the **LinearAlgebra** package, and check to see if the matrices are equal. Checking equality of matrices requires using the **Equal** command.

```
> IsSymmetricM := proc(M::Matrix)
    return LinearAlgebra[Equal](M,LinearAlgebra[Transpose](M));
  end proc:
> IsSymmetricM(Div6M);
```

false **(9.42)**

```
> IsSymmetricM(DivOrMul6M);
```

true **(9.43)**

Representing Relations Using Digraphs

Now we turn to representing relations with directed graphs, commonly called digraphs. Maple provides a package called **GraphTheory** that contains the functions we will need to create and view graphs.

```
> with(GraphTheory):
```

You create a digraph in Maple using the **Digraph** command. This command has a variety of possible calling sequences, including the ability to give the command only one argument: a set of 2-element lists representing the edges of the graph. For example, consider Bob and his sister Barb,

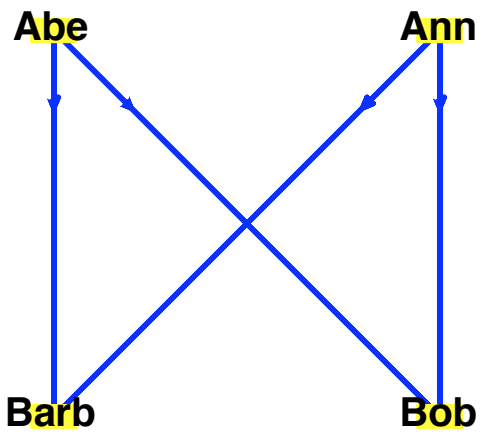
whose parents are Ann and Abe. We can make a directed graph representing the relation "parent of" as follows.

```
> familyTree := Digraph({["Ann","Bob"],["Ann","Barb"],["Abe",
  "Bob"],["Abe","Barb"]});
```

familyTree := Graph 1: a directed unweighted graph with 4 vertices and 4 arc(s) **(9.44)**

The command **DrawNetwork** provides a visual representation of the graph.

```
> DrawNetwork(familyTree);
```



Note that we use the **DrawNetwork** command rather than Maple's **DrawGraph** command. In this context, a network is a connected directed graph with at least one "source" and at least one "sink." A "source" is an element with no edges into it and a "sink" is an element with no edges out of it. In the above, Abe and Ann are sources and Barb and Bob are sinks. In the language of posets, described in Section 9.6, sources and sinks are equivalent to minimal and maximal elements. If the digraph associated to a relation is a network, then **DrawNetwork** has a distinct advantage over **DrawGraph**, namely, **DrawNetwork** uses the source and sink to arrange the elements in a natural way producing a much more illustrative graph. If the digraph is not a network, then we can still use **DrawGraph**.

Many, but not all, of the relations we may wish to visualize are networks. So we'll create a procedure, **DrawRelation**, that will check to see if the digraph associated to the given relation is a network and then use the appropriate draw command. In this procedure, we apply the **IsNetwork** test to the digraph associated to the relation. With this syntax, **IsNetwork** does not return a boolean value. Instead, it returns two sets, the first being the set of all sources and the second the set of all sinks. If either of these is empty, then the digraph is not a network.

Before creating the **DrawRelation** procedure, there are two more issues we must address. First, Maple does not allow for self-loops in a graph object. For example, if we tried to make Abe his own parent in the example above, Maple would raise an error.

```
> Digraph({["Ann","Bob"],["Ann","Barb"],["Abe","Bob"],["Abe",
  "Barb"],["Abe","Abe"]});
Error, (in GraphTheory:-Graph) invalid edge/arc: ["Abe",
"Abe"]
```

In other words, in order for a relation to be passed to the **Digraph** command, the relation must be *irreflexive* (*i.e.*, no element can be related to itself). We can still use Maple to visualize relations as graphs, we just have to keep this limitation in mind. And we have to write a procedure that removes

any pair of the form `[a,a]` from the relation before constructing the graph.

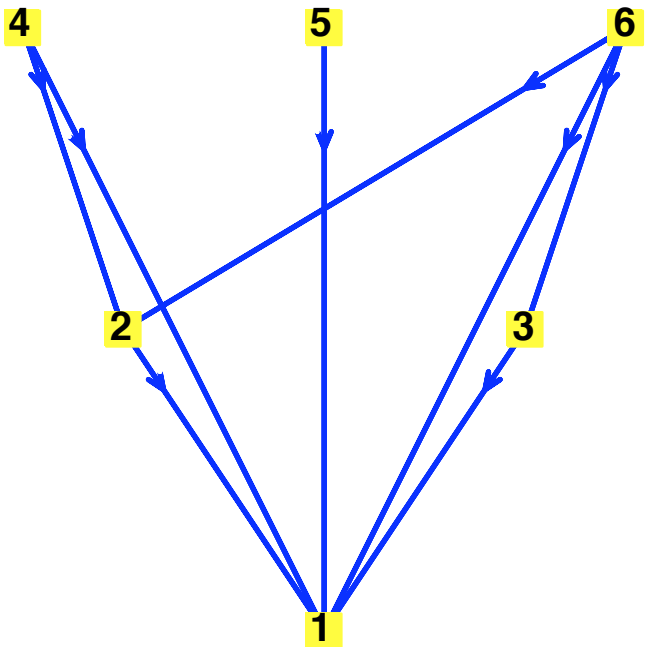
```
> MakeIrreflexive := proc(R::rel)
    local E, u;
    E := {};
    for u in R do
      if u[1] <> u[2] then
        E := E union {u};
      end if;
    end do;
    return E;
  end proc:
```

Finally, Maple draws networks with the sources at the top, which, in this context, yields graphs that are upside down in comparison to the usual way we draw them. So we'll reverse the relation using our `InverseRelation` procedure. Now we are ready to define the `DrawRelation` procedure.

```
> DrawRelation := proc(R::rel)
    local IrrR, RevR, G, S;
    IrrR := MakeIrreflexive(R);
    RevR := InverseRelation(IrrR);
    G := GraphTheory[Digraph](RevR);
    S := GraphTheory[IsNetwork](G);
    if S[1] = {} or S[2] = {} then
      GraphTheory[DrawGraph](G);
    else
      GraphTheory[DrawNetwork](G);
    end if;
  end proc:
```

Now we can convert our relations to graphs and visualize them.

```
> DrawRelation(Div6);
```



We can use this representation to determine whether or not the relation is transitive. To do this, we use Maple's implementation of the Floyd-Warshall all-pairs shortest path algorithm called `AllPairsDistance`. This procedure returns a matrix whose (i,j) entry represents the shortest

path from vertex i to vertex j. For example, the all-pairs distance matrix for the **Div6** relation is:

```
> AllPairsDistance(Digraph(MakeIrreflexive(Div6)));
```

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 \\ \infty & 0 & \infty & 1 & \infty & 1 \\ \infty & \infty & 0 & \infty & \infty & 1 \\ \infty & \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & 0 \end{bmatrix} \tag{9.45}$$

In a graph of a transitive relation, the distance between any two distinct elements must be either 1 or infinite (meaning there is no path between them). To see this, assume that you have a transitive relation and suppose there are elements A and Z that the all-pairs algorithm has determined have distance 3. That means there must be two elements, say M and N, such that A is connected to M is connected to N is connected to Z. From the point of view of the relation, then, (A, M) and (M, N) and (N, Z) are all members of the relation. But if the relation is transitive, the fact that (A, M) and (M, N) are in the relation means that (A, N) is in the relation. So, A to N to Z is a shorter path (of length 2). Applying transitivity again shows that A and Z are adjacent. While this does not amount to a proof, it should be convincing that we can check for transitivity by making sure that no two vertices in the graph of a relation have distance which is finite and greater than 1.

```
> IsTransitiveG := proc(R::rel)
    local G, D, i, j;
    G := Digraph(MakeIrreflexive(R));
    D := AllPairsDistance(G);
    for i from 1 to LinearAlgebra[RowDimension](D) do
      for j from 1 to LinearAlgebra[ColumnDimension](D) do
        if D[i,j] > 1 and D[i,j] < infinity then
          return false;
        end if;
      end do;
    end do;
    return true;
  end proc:
```

```
> IsTransitiveG(Div6);
```

true **(9.46)**

```
> IsTransitiveG(R2);
```

false **(9.47)**

9.4 Closures of Relations

In this section, we will develop algorithms to compute the reflexive, symmetric, and transitive closures of binary relations. We begin with the reflexive closure.

Reflexive Closure

The algorithm for computing the reflexive closure of a relation, with the matrix representation, is very simple. We simply set each diagonal entry equal to 1. The resulting matrix represents the reflexive closure of the relation.

There is one technical consideration to bear in mind: in order to avoid changing the matrix that we

pass to these procedures, we begin by assigning a temporary variable, **ans**, to **LinearAlgebra[Copy](M)**. It is common in programming languages, and Maple is no exception, that variables assigned to high-level objects like matrices are stored as references. This means that if you have a matrix named **M**, and you enter **N := M;**, you don't get two different matrices, you get two names for the same matrix. And changing one changes both. The **Copy** command forces Maple to make a distinct matrix so that the original is preserved.

Here is the procedure for computing the reflexive closure.

```
> ReflexiveClosure := proc(M::Matrix)
    local i, ans;
    ans := LinearAlgebra[Copy](M);
    for i from 1 to LinearAlgebra[ColumnDimension](M) do
      ans[i,i] := 1;
    end do;
    return ans;
  end proc:
```

(Note that all the closure operations only apply to a relation on a set and are generally not valid for a relation from one set to a different set. This means we may assume that the matrix representation of the relation is square. If the matrix passed to this or the following procedures is not square, an error will likely occur.)

We use this procedure to find the reflexive closure of the example relation we introduced earlier in the chapter.

```
> ReflexiveClosure(exampleMatrix);
```

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \textbf{(9.48)}$$

Recall that **exampleMatrix** represented the "is one less than" relation. Looking at the matrix above, you can see that the reflexive closure includes equality.

Symmetric Closure

Next we write a procedure for constructing the symmetric closure of a relation R. We use the observation that if (a, b) is a member of R then (b, a) must be included in the symmetric closure, so we can simply add it to the relation.

```
> SymmetricClosure := proc(M::Matrix)
    local i, j, ans;
    ans := LinearAlgebra[Copy](M);
    for i from 1 to LinearAlgebra[RowDimension](M) do
      for j from 1 to LinearAlgebra[ColumnDimension](M) do
        if ans[i,j] = 1 then
          ans[j,i] := 1;
        end if;
      end do;
    end do;
    return ans;
  end proc:
```

Applying this to our **exampleMatrix** yields the "different by 1" relation. And applying it to the "is a divisor of" relation yields the "is a divisor or multiple of" relation.

```
> SymmetricClosure(exampleMatrix);
```

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{9.49}$$

```
> SymmetricClosure(Div6M);
```

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \tag{9.50}$$

Transitive Closure

Having created the reflexive and symmetric closures, we turn to implementing the transitive closure in Maple. This is a more difficult problem than the earlier cases, both in terms of computational complexity and implementation. In the text, there are two algorithms outlined (a generic transitive closure and Warshall's algorithm) and both will be covered in this section.

One transitive closure procedure

First we will implement the transitive closure algorithm presented as Algorithm 1 in Chapter 9 of the text. This will require the Boolean join and Boolean product operations on zero-one matrices that were introduced in Chapter 2. Recall from Section 2.6 of this manual that the **Bits** package includes Maple's functionality for bit-wise operations. In particular, remember that the **And** and **Or** functions correspond to the Boolean operations $\wedge$ and $\vee$. Here are some examples.

```
> with(Bits):
> And(1,1);
```

$$1 \tag{9.51}$$

```
> And(0,1);
```

$$0 \tag{9.52}$$

```
> Or(0,1);
```

$$1 \tag{9.53}$$

```
> Or(1,1);
```

$$1 \tag{9.54}$$

Now we can recreate the Boolean join matrix operation. Recall that for zero-one matrices A and B of the same size, the join of A and B is the matrix $A \vee B$ whose (i,j) entry is $A_{ij} \vee B_{ij}$. Refer to Section 2.6 of this manual for a detailed discussion of this procedure.

```
> BoolJoin := proc(A::'Matrix'({0,1}),B::'Matrix'({0,1}))
    local numrows, numcols, R, r, c;
    uses LinearAlgebra, Bits;
    numrows := RowDimension(A);
    numcols := ColumnDimension(A);
    if numrows <> RowDimension(B) or
```

```
        numcols <> ColumnDimension(B) then
      error "Input matrices must be of the same size.";
    end if;
    R := Matrix(numrows,numcols);
    for r from 1 to numrows do
      for c from 1 to numcols do
        R[r,c] := Or(A[r,c],B[r,c]);
      end do;
    end do;
    return R;
  end proc:
```

For example,

```
> joinA := Matrix([[1,0],[0,1]]);
```

$$joinA := \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{9.55}$$

```
> joinB := Matrix([[1,1],[0,0]]);
```

$$joinB := \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \tag{9.56}$$

```
> BoolJoin(joinA,joinB);
```

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \tag{9.57}$$

Next, recall that for appropriately sized zero-one matrices, the Boolean product $A \odot B$ is the matrix whose (i, j) entry is obtained by the formula

$$\bigvee_{k=1}^{n} (a_{ik} \wedge b_{kj})$$

where n is the number of columns of A, which is also the number of rows of B. This is implemented in the `BoolProduct` procedure. Again, refer to Section 2.6.

```
> BoolProduct := proc(A::'Matrix'({0,1}),B::'Matrix'({0,1}))
    local m, k, n, C, i, j, c, p;
    uses LinearAlgebra, Bits;
    m := RowDimension(A);
    k := ColumnDimension(A);
    if k <> RowDimension(B) then
      error "Dimension mismatch.";
    end if;
    n := ColumnDimension(B);
    C := Matrix(m,n);
    for i from 1 to m do
      for j from 1 to n do
        c := And(A[i,1],B[1,j]);
        for p from 2 to k do
          c := Or(c,And(A[i,p],B[p,j]));
        end do;
        C[i,j] := c;
      end do;
    end do;
    return C;
  end proc:
```

As an example,

```
> productA := Matrix([[1,0],[0,1],[1,0]]);
```

$$productA := \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{9.58}$$

```
> productB := Matrix([[1,1,0],[0,1,1]]);
```

$$productB := \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \tag{9.59}$$

```
> BoolProduct(productA,productB);
```

$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix} \tag{9.60}$$

We are now ready to implement Algorithm 1 from Section 9.4 for calculating the transitive closure. Recall that the idea of this algorithm is that we compute Boolean powers of the matrix of the relation, up to the size of the domain. At each step, we use the Boolean join on $A = M^{[i]}$ and the result matrix B.

```
> TransitiveClosure := proc(M::Matrix)
    local i, A, B;
    uses LinearAlgebra, Bits;
    A := Copy(M);
    B := Copy(M);
    for i from 2 to RowDimension(M) do
      A := BoolProduct(A,M);
      B := BoolJoin(B,A);
    end do;
    return B;
  end proc:
```

We test our transitive closure procedure on Example 7 from Section 9.4, where it was found that the relation with matrix representation

$$M_R = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix}$$

has transitive closure

$$M_{R^*} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}.$$

```
> example7 := Matrix([[1,0,1],[0,1,0],[1,1,0]]);
```

$$example7 := \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \tag{9.61}$$

```
> TransitiveClosure(example7);
```

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \tag{9.62}$$

Warshall's algorithm

Next we consider Warshall's algorithm, as presented as Algorithm 2 in Section 9.4. This algorithm is straightforward to implement.

```
> Warshall := proc(M::Matrix)
    local i, j, k, W, n;
    uses LinearAlgebra, Bits;
    n := ColumnDimension(M);
    W := Copy(M);
    for k from 1 to n do
      for i from 1 to n do
        for j from 1 to n do
          W[i,j] := Or(W[i,j], And(W[i,k],W[k,j]));
        end do;
      end do;
    end do;
    return W;
  end proc:
```

Applying this to the same example as before, we see that the result is correct.

```
> Warshall(example7);
```

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \tag{9.63}$$

We can compare these two procedures in terms of execution time using Maple's **time** command. But, we must point out that this comparison for a single example does not prove anything about the complexity or relative performance of the two algorithms. Rather, it serves as a demonstration that, even for relations on small domains, the difference in the computational complexity of the algorithms is noticeable. We shall consider the following zero-one matrix that represents a relation on the set {1, 2, 3, 4, 5, 6}.

```
> transitiveCompare := Matrix([[0,0,0,0,0,1],[1,0,1,0,0,0],[1,
  0,0,1,0,0],[1,0,0,0,1,0],[1,0,0,0,0,1],[0,1,0,0,0,0]]);
```

$$transitiveCompare := \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{9.64}$$

```
> st:=time(): Warshall(transitiveCompare): time()-st;
```

$$0. \tag{9.65}$$

```
> st:=time(): TransitiveClosure(transitiveCompare):  time()-st;
```

```
0.005                                                    (9.66)
```

From this example, we can see that Warshall's algorithm can be a substantial improvement over the alternative, at least on this specific example. The reader is encouraged to explore this further.

9.5 Equivalence Relations

In this section we will examine how we can use Maple to compute with equivalence relations. There are three specific problems that we will address here: given an equivalence relation on a set, how to compute the equivalence class of an element; how to determine the number of equivalence relations on a finite set; and how to compute the smallest equivalence relation that contains a given relation on some finite set. Note that in this section, relations are assumed to be represented as in the start of this chapter, as objects of type **rel**.

First, we provide a test that determines whether or not a relation is an equivalence relation. Using the work that we've already done and recalling that an equivalence relation is simply a relation that is reflexive, symmetric, and transitive, this task is a simple one.

```
> IsEquivalenceRelation := proc(R::rel);
    if IsReflexive(R) and IsSymmetric(R)
                      and IsTransitive(R) then
      return true;
    else
      return false;
    end if;
  end proc:
```

As an example, let's define the equivalence relation "congruent mod 4" on the integers from 0 to n.

```
> makeMod4Rel := proc(n::posint)
    local R, i, j;
    R := {};
    for i from 0 to n do
      for j from 0 to n do
        if modp(i-j,4) = 0 then
          R := R union {[i,j]};
        end if;
      end do;
    end do;
    return R;
  end proc:
> makeMod4Rel(8);
```

$$\{[0,0],[0,4],[0,8],[1,1],[1,5],[2,2],[2,6],[3,3],[3,7],[4,0],[4,4],[4,8],[5,1],[5,5],[6,2],[6,6],[7,3],[7,7],[8,0],[8,4],[8,8]\} \quad (9.67)$$

```
> IsEquivalenceRelation((9.67));
```

$$true \quad (9.68)$$

Equivalence Classes

Recall that, given an equivalence relation R and a member a of the domain of R, the equivalence class of a is the set of all members b of the domain for which the pair (a, b) belongs to R. In other words, it is the set of all elements in the domain that are R-equivalent to a. So to determine the equivalence class of a particular element of the domain, the algorithm is fairly simple. We just search through R looking for all pairs of the form (a, b), adding each such second element b to the class. We do not have to search for pairs of the form (b, a) because equivalence relations are

symmetric. The following procedure returns the equivalence class for a given equivalence relation and a point in the domain.

```
> EquivalenceClass := proc(R::rel, a::anything)
    local u, S;
    S := {};
    for u in R do
      if u[1] = a then
        S := S union {u[2]};
      end if;
    end do;
    return S;
  end proc:
```

As an example, we compute the equivalence class of 3 in the modulo 4 relation on the domain $\{1, 2, \ldots 30\}$.

```
> EquivalenceClass(makeMod4Rel(30),3);
```

$$\{3, 7, 11, 15, 19, 23, 27\} \qquad \textbf{(9.69)}$$

Number of Equivalence Relations on a Set

Next, we consider how to construct all of the equivalence relations on a given (finite) set. The straightforward way to do this is to construct all relations on the given domain and then check them to see if they are equivalence relations. Since a relation on a set A is merely a subset of $A \times A$, generating all relations is the same as generating all subsets of $A \times A$.

To implement this, we begin by creating the set $A \times A$ using Maple's **cartprod** function. The **cartprod** function takes a list of sets and returns a table with two entries: **finished** and **nextvalue**. The **finished** entry is a boolean indicating whether the last element of the product has been reached. The **nextvalue** entry is a function which returns the next element of the Cartesian product. This may seem a strange approach, but it allows for the conceptual creation of the Cartesian product of two sets without the need to actually store the entire product. In this case, we do want to create and store the entire product, so we will use the following structure.

```
> Cprod := combinat[cartprod]([{1,2,3},{8,9}]):
> CprodSet := {}:
  while not Cprod[finished] do
    CprodSet := CprodSet union {Cprod[nextvalue]()}:
  end do:
  CprodSet;
```

$$\{[1, 8], [1, 9], [2, 8], [2, 9], [3, 8], [3, 9]\} \qquad \textbf{(9.70)}$$

To complete our equivalence relation procedure, we use the **powerset** command from the **combinat** package. Recall that the powerset of a set is the set of all subsets. So the powerset of $A \times A$ is the set of all relations on A. Then we just check them one by one to see which are equivalence relations.

```
> AllEquivalenceRelations := proc(A::set)
    local C, AA, P, R, E, a, b;
    C := combinat[cartprod]([A,A]);
    AA := {};
    while not C[finished] do
      AA := AA union {C[nextvalue]()};
    end do;
    P := combinat[powerset](AA);
    E := {};
```

```
    for R in P do
      if IsEquivalenceRelation(R) then
        E := E union {R};
      end if;
    end do;
    return E;
  end proc:
```

For example, there are 15 equivalence relations on $\{1, 2, 3\}$.

```
> AllEquivalenceRelations({1,2,3});
```

$$\{\{\}, \{[1,1]\}, \{[2,2]\}, \{[3,3]\}, \{[1,1],[2,2]\}, \{[1,1],[3,3]\}, \{[2,2],[3,3]\}, \{[1,1],[2,2],[3,3]\}, \{[1,1],[1,2],[2,1],[2,2]\}, \{[1,1],[1,3],[3,1],[3,3]\}, \{[2,2],[2,3],[3,2],[3,3]\}, \{[1,1],[1,2],[2,1],[2,2],[3,3]\}, \{[1,1],[1,3],[2,2],[3,1],[3,3]\}, \{[1,1],[2,2],[2,3],[3,2],[3,3]\}, \{[1,1],[1,2],[1,3],[2,1],[2,2],[2,3],[3,1],[3,2],[3,3]\}\} \quad \textbf{(9.71)}$$

```
> nops((9.71));
```

$$15 \quad \textbf{(9.72)}$$

Closure

The last question to be considered in this section is the problem of finding the smallest equivalence relation containing a relation R.

The key idea is that we need to find the smallest relation containing R that is reflexive, symmetric, and transitive. Recalling the previous section on closures, it is natural to think that we may compute the reflexive closure, the symmetric closure, and then the transitive closure, one after the other. The only concern would be that one closure would no longer have one of the previous properties. The following outlines why this is not the case.

1. First create the reflexive closure of R, call it P.
2. Compute the symmetric closure of P and call this Q. Note that Q is still reflexive since no pairs were removed from the relation and no elements were added to the domain. So Q is both symmetric and reflexive.
3. Compute the transitive closure of Q and name this S. Note that S is still reflexive for the same reason as above. And S is still symmetric since, if (a, b) and (b, c) are in Q to force the addition of (a, c), then since Q is symmetric, (c, b) and (b, a) must also being in Q forcing (c, a) to also be included in S. Hence, S is an equivalence relation.

We implement this method as the composition of the four methods **RelToMatrix**, **ReflexiveClosure**, **SymmetricClosure**, and then **TransitiveClosure**. We use Maple's composition operator, @.

```
> EquivalenceClosure := TransitiveClosure @ SymmetricClosure @
  ReflexiveClosure @ RelToMatrix;
```

$$EquivalenceClosure := TransitiveClosure@SymmetricClosure@ReflexiveClosure@RelToMatrix \quad \textbf{(9.73)}$$

As an example, recall the **Div6** relation representing the "is a divisor of" on $\{1, 2, 3, 4, 5, 6\}$. We can see that the smallest equivalence relation that contains **Div6** is the relation in which every number is related to every other number.

```
> EquivalenceClosure(Div6);
```

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{9.74}$$

This is unsurprising, since 1 is a divisor of every number meaning that, in any equivalence relation containing the "divides" relation, 1 is related to every number. We can make this example slightly more interesting by removing 1.

```
> Div17minus1 := DividesRelation({$2..17});
```

Div17minus1 := {[2, 2], [2, 4], [2, 6], [2, 8], [2, 10], [2, 12], [2, 14], [2, 16], [3, 3], [3, 6], [3, 9], [3, 12], [3, 15], [4, 4], [4, 8], [4, 12], [4, 16], [5, 5], [5, 10], [5, 15], [6, 6], [6, 12], [7, 7], [7, 14], [8, 8], [8, 16], [9, 9], [10, 10], [11, 11], [12, 12], [13, 13], [14, 14], [15, 15], [16, 16], [17, 17]} **(9.75)**

```
> interface(rtablesize=17):
  EquivalenceClosure(Div17minus1);
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{9.76}$$

(Note the first row and column still correspond to 1 because of the way the matrix is constructed in **RelToMatrix**.) In this example, you see that 11, 13, and 17 become isolated, being the three primes in the set which do not have multiples of them also included.

9.6 Partial Orderings

In this section, we consider partial orderings (or partial orders) and related topics, including maximal and minimal elements, Hasse diagrams, and lattices. We will explore these topics in Maple, and leave the exploration of other topics related to partial orderings to the reader.

Partial Orders and Examples

First, we will define a new Maple type for partial orders and create some examples of them.

Recall that a partial order is a binary relation on a set that satisfies the three conditions of being reflexive, antisymmetric, and transitive. Previously in this section we defined various types (e.g., `rel`, our relation type) via the structured type syntax. In this case, though, it is not enough to know the structure of the type's data, we also want to insist on certain properties. We define the type to be a procedure that tests an object against the definition of a partial order. It is very similar to the `IsEquivalenceRelation` procedure we created in the previous section.

Here is the definition of the partial order type.

```
> `type/po` := proc(obj)
    type(obj, rel) and IsReflexive(obj)
                   and IsAntisymmetric(obj)
                   and IsTransitive(obj);
    end proc:
```

Unlike the structured type syntax, we explicitly define a procedure. The main rules for defining types in this way are that the procedure must take one argument and it must return true or false.

Now we can use the name `po` as the second argument to the `type` command for checking to see if an object is of the type. For example, we can check that the `Div6` relation we defined earlier (recall that this is the "divides" relation on the set 1 through 6) is a partial order.

```
> type(Div6,po);
```

true **(9.77)**

We create some additional examples of partial orderings that we can use in the remainder of the section. The `Div17minus1` relation (this was the "divides" relation on the set 2 through 17) is a partial order.

```
> type(Div17minus1,po);
```

true **(9.78)**

Indeed, all the relations created via the `DividesRelation` or `DivRel` procedures will be partial orders.

Next, we create a procedure to produce examples of a class of lattices (we will discuss lattices more below, for now it is enough that these examples are partial orders). The `DivisorLattice` procedure will create the partial order whose domain is the set of positive divisors of a given number and whose order is defined by the "divides" relation. We only need to apply the `DividesRelation` procedure to the divisors of the given number.

```
> DivisorLattice := proc(n::posint)
    DividesRelation(numtheory[divisors](n));
  end proc:
```

The `divisors` command in the `numtheory` package produces all of the positive divisors of the given integer.

```
> DivisorLattice(10);
```

$\{[1, 1], [1, 2], [1, 5], [1, 10], [2, 2], [2, 10], [5, 5], [5, 10], [10, 10]\}$ **(9.79)**

Finally, for a bit of variety, we create the posets whose Hasse diagrams are shown in Figure 8(a) and Figure 10 in Section 9.6.

```
> Fig8A := {["a","a"],["a","b"],["a","c"],["a","d"],["a","e"],
  ["a","f"],["b","b"],["b","c"],["b","d"],["b","e"],["b","f"],
  ["c","c"],["c","e"],["c","f"],["d","d"],["d","e"],["d","f"],
  ["e","e"],["e","f"],["f","f"]};
```

$$Fig8A := \{["a","a"], ["a","b"], ["a","c"], ["a","d"], ["a","e"], ["a","f"], ["b","b"], ["b","c"], ["b","d"], ["b","e"], ["b","f"], ["c","c"], ["c","e"], ["c","f"], ["d","d"], ["d","e"], ["d","f"], ["e","e"], ["e","f"], ["f","f"]\} \quad (9.80)$$

```
> type(Fig8A,po);
```

$$true \quad (9.81)$$

```
> Fig10 := {["A","A"],["A","B"],["A","D"],["A","F"],["A","G"],
  ["B","B"],["B","D"],["B","F"],["B","G"],["C","C"],["C","B"],
  ["C","D"],["C","F"],["C","G"],["D","D"],["D","G"],["E","E"],
  ["E","F"],["E","G"],["F","F"],["F","G"],["G","G"]};
```

$$Fig10 := \{["A","A"], ["A","B"], ["A","D"], ["A","F"], ["A","G"], ["B","B"], ["B","D"], ["B","F"], ["B","G"], ["C","B"], ["C","C"], ["C","D"], ["C","F"], ["C","G"], ["D","D"], ["D","G"], ["E","E"], ["E","F"], ["E","G"], ["F","F"], ["F","G"], ["G","G"]\} \quad (9.82)$$

```
> type(Fig10,po);
```

$$true \quad (9.83)$$

Hasse Diagrams

Now that we have defined a type and have examples at our disposal, we turn to the problem of having Maple draw Hasse diagrams of partial orders. As demonstrated in the textbook, a Hasse diagram is a very useful tool for visualizing and understanding posets. Drawing the Hasse diagram for a poset is not as simple as drawing all of the elements of the set and then connecting all related pairs with an edge. Doing so would create an extremely messy, and not very useful, diagram. Instead, a Hasse diagram contains only those edges that are absolutely necessary to reveal the structure of the poset.

Covering relations

The covering relation for a partial order is a minimal representation of the partial order, from which the partial order can be reconstructed via transitive and reflexive closure.

Let $\preccurlyeq$ be a partial order on a set S. Recall that an element y in S *covers* an element x in S if $x \prec y$, $x \neq y$, and there is no element z of S, different from x and y, such that $x \prec z \prec y$. In other words, y covers x if y is greater than x and there is no intermediary element. The set of pairs (x, y) for which y covers x is the covering relation of $\preccurlyeq$.

As a simple example, consider the set $\{1, 2, 3, 4\}$ ordered by magnitude, *i.e.*, the usual "less than or equal to." This relation consists of 10 ordered pairs:

$$\{(1,1), (1,2), (1,3), (1,4), (2,2), (2,3), (2,4), (3,3), (3,4), (4,4)\}.$$

Its covering relation is the set

$$\{(1,2), (2,3), (3,4)\},$$

which consists of only 3 pairs. All the other pairs of the partial order can be inferred from the covering relation using transitivity and reflexivity. For instance, $(1, 3)$ can be recovered from $(1, 2)$ and $(2, 3)$ via transitivity. Note that the covering relation involves many fewer pairs and

thus is a much more efficient way to represent the partial order, at least in terms of storage.

Our goal in this subsection is to write a procedure that will have Maple draw the Hasse diagram of a given partial order. Since a Hasse diagram is, in fact, the graph of the associated covering relation, we will create a procedure to find the covering relation of the partial order.

First, we need a test to check whether a given element covers another.

```
> Covers := proc(R::po, x, y)
    local z;
    if x = y then
      return false;
    end if;
    if not [x,y] in R then
      return false;
    end if;
    for z in FindDomain(R) minus {x,y} do
      if ([x,z] in R) and ([z,y] in R) then
        return false;
      end if;
    end do;
    return true;
  end proc:
```

This procedure works by first checking to see if the two elements x and y are equal to each other or if the pair (x, y) fails to be in the partial order. In either of these situations, y does not cover x. Assuming the pair of elements passes these basic hurdles, the procedure then checks every other element of the domain. If it can find an element that sits between x and y, then we know they don't cover. If no element sits between them, then in fact y does cover x.

Now we can construct the covering relation of a partial order using the following Maple procedure. This procedure simply checks every element of the given relation to see if one covers the other and adds those that do to the output relation C.

```
> CoveringRelation := proc(R::po)
    local C, u;
    C := {};
    for u in R do
      if Covers(R, u[1], u[2]) then
        C := C union {[u[1],u[2]]};
      end if;
    end do;
    return C;
  end proc:
```

Let's look at a couple of examples. First, the example described above, of the set $\{1, 2, 3, 4\}$ ordered by magnitude.

```
> CoveringRelation({[1,1],[1,2],[1,3],[1,4],[2,2],[2,3],[2,4],
  [3,3],[3,4],[4,4]});
```

$$\{[1, 2], [2, 3], [3, 4]\} \tag{9.84}$$

As a second example, let's consider a lattice.

```
> CoveringRelation(DivisorLattice(30));
```

$$\{[1, 2], [1, 3], [1, 5], [2, 6], [2, 10], [3, 6], [3, 15], [5, 10], [5, 15], [6, 30], [10, 30], [15, 30]\} \tag{9.85}$$

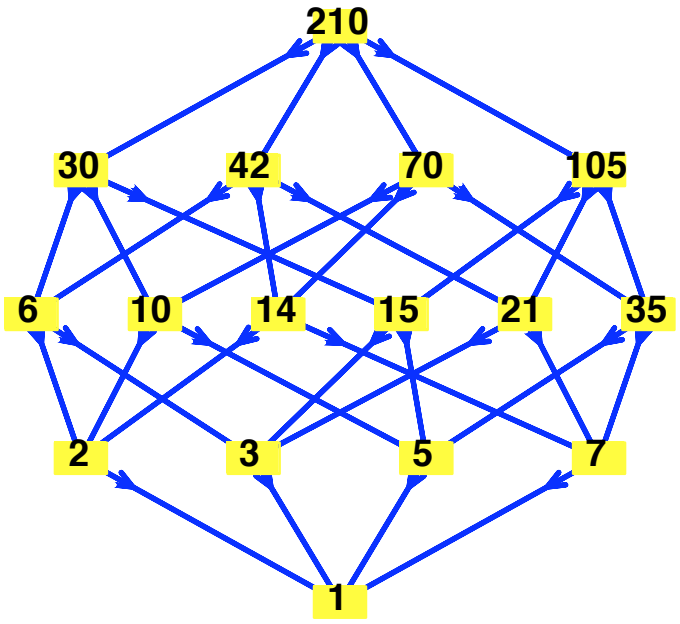
Drawing Hasse diagrams

Now we will use the covering relation in order to write a program to draw the Hasse diagram for partial orders. By using the **CoveringRelation** procedure that we just completed and the **Digraph** and **DrawNetwork** procedures from the **GraphTheory** package, we can create the graph associated to a partial order. As we did previously in this chapter when graphing relations, we reverse the relation in order to have the smallest elements at the bottom, as is typical.

```
> HasseDiagram := proc(R::po)
    local C, D;
    C := InverseRelation(CoveringRelation(R));
    D := GraphTheory[Digraph](C);
    GraphTheory[DrawNetwork](D);
  end proc:
```

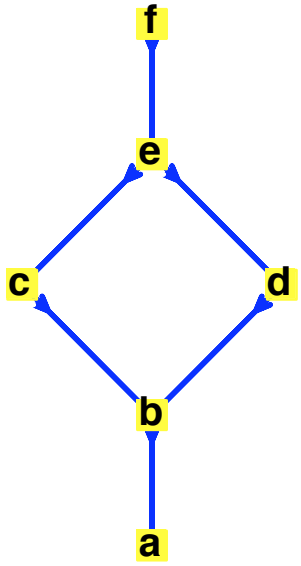
As an example, here is a diagram representing the divisor lattice of 210.

```
> HasseDiagram(DivisorLattice(2*3*5*7),1,210);
```

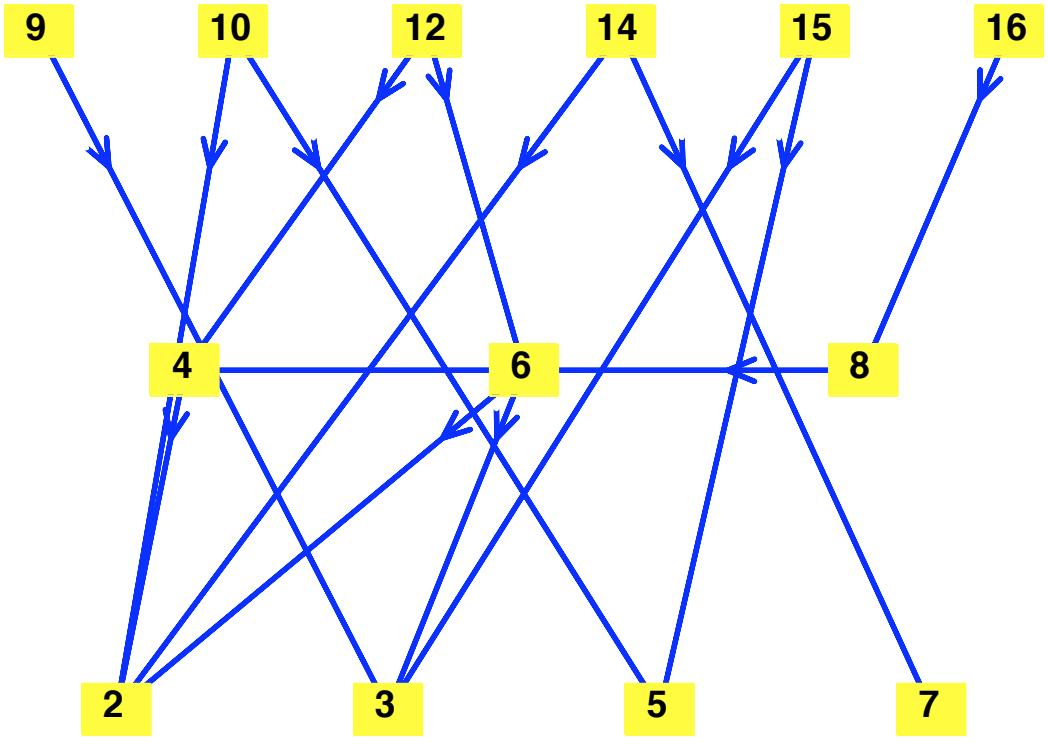


And here are the Hasse diagrams for some of the other examples we discussed in this section.

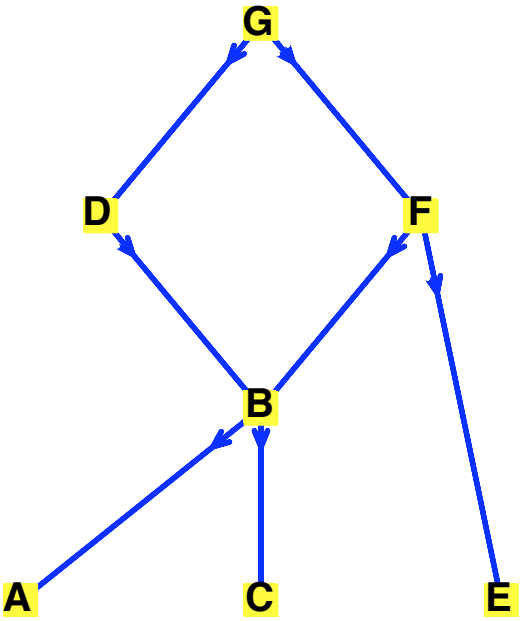
```
> HasseDiagram(Fig8A);
```




```
> HasseDiagram(Div17minus1);
```




```
> HasseDiagram(Fig10);
```



Comparing this last example to the diagram given in the textbook illustrates that, while using Maple's **DrawNetwork** command doesn't result in quite as appealing graphs as those that are created by hand, it still provides a fairly useful graph.

Maximal and Minimal Elements

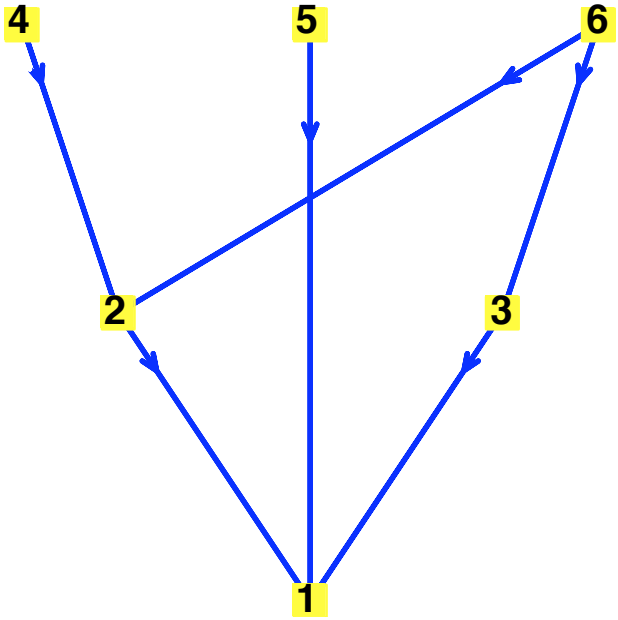
We will construct a procedure that determines the set of minimal elements of a partially ordered set.

The procedure takes two arguments: a partial order *R* and a subset *S* of the domain of *R*. It returns the set of minimal elements of *S* with respect to *R*. It first initializes the set of minimal elements to all of *S* and then removing those that are not minimal.

```
> MinimalElements := proc(R::po, S::set)
    local M, s, t;
    if S minus FindDomain(R) <> {} then
      error "Set must be in the domain of the relation";
    end if;
    M := S;
    for s in S do
      for t in S minus {s} do
        if [t,s] in R then
          M := M minus {s};
        end if;
      end do;
    end do;
    return M;
  end proc:
```

We can see this work on our **Div6** partial order. Since we'll be using the **Div6** partial order for many examples in this section, it's Hasse diagram may also be useful.

```
> HasseDiagram(Div6);
```




```
> MinimalElements(Div6,{1,2,3,4,5,6});
```

$$\{1\} \tag{9.86}$$

```
> MinimalElements(Div6,{2,3,4,5,6});
```

$$\{2, 3, 5\} \tag{9.87}$$

Note that, by reversing the relation and thus the order, we can compute maximal elements very easily.

```
> MaximalElements := proc(R::po, S::set)
    MinimalElements(InverseRelation(R),S);
  end proc:
> MaximalElements(Div6,{1,2,3,4,5,6});
```

$$\{4, 5, 6\} \tag{9.88}$$

Least Upper Bound

Next we will write a procedure for computing the least upper bound of a set with respect to a partial order, if it exists. Our procedure will return the value **NULL** in the case that the set has no least upper bound.

First we create a procedure **IsUpperBound** that determines whether a given element is an upper bound of a set with respect to a relation. It accomplishes this by checking to make sure that the given element is greater than every element of the set.

```
> IsUpperBound := proc(R::po, S::set, u::anything)
    local s;
    if not u in FindDomain(R) then
      error "Element is not in the domain of the relation."
    end if;
    for s in S do
      if not [s,u] in R then
        return false;
      end if;
    end do;
    return true;
  end proc:
```

For example, under the **Div6** relation, 6 is an upper bound of {1, 2, 3}, but not of {1, 2, 3, 4}.

```
> IsUpperBound(Div6,{1,2,3},6);
```

true **(9.89)**

```
> IsUpperBound(Div6,{1,2,3,4},6);
```

false **(9.90)**

Next we write a procedure to find all of the upper bounds for a given set. We do this by considering every element of the domain of the relation and checking to see which are upper bounds, using the **IsUpperBound** procedure.

```
> UpperBounds := proc(R::po, S::set)
    local U, DomR, d;
    DomR := FindDomain(R);
    if S minus DomR <> {} then
      error "set must be contained in the domain of the
  relation."
    end if;
    U := {};
    for d in DomR do
      if IsUpperBound(R, S, d) then
        U := U union {d};
      end if;
    end do;
    return U;
  end proc:
```

For instance, the upper bounds of the set {1, 2} under **Div6** are:

```
> UpperBounds(Div6, {1,2});
```

{2, 4, 6} **(9.91)**

To complete the task of finding the least upper bound of a set, we merely use **UpperBounds** to compute all of the upper bounds for the set, use **MinimalElements** to see which of the upper bounds are minimal, and then check to see how many minimal upper bounds are found. If there is exactly one minimal upper bound, then this is the least upper bound. Otherwise, the set has no least upper bound.

```
> LeastUpperBound := proc(R::po, S::set)
    local U, M;
    U := UpperBounds(R,S);
    M := MinimalElements(R,U);
    if nops(M) <> 1 then
      return NULL;
    else
      return op(M);
    end if;
  end proc:
```

For example, the least upper bounds of {1, 2} and {1, 2, 3} are found below, while {4, 5} has no least upper bound in the domain of **Div6** and so does not return a value.

```
> LeastUpperBound(Div6, {1,2});
```

2 **(9.92)**

```
> LeastUpperBound(Div6, {1,2,3});
```

6 **(9.93)**

```
> LeastUpperBound(Div6, {4,5});
```

Lattices

As the last topic in this section, we will consider the problem of determining whether a partial order is a lattice. The approach we will take is a good example of "top down programming." The test we design here will confirm that the procedure **DivisorLattice** written at the beginning of this section does indeed produce lattices.

Recall that a partial order is a lattice if every pair of elements has both a least upper bound and a greatest lower bound (in lattices, these are also referred to as the supremum and infimum of the pair or as their meet and join). With this in mind, we can write the following procedure (with the understanding that the helper procedures still need to be written).

```
> IsLattice := proc(R::po)
    HasLUBs(R) and HasGLBs(R);
  end proc:
```

We need to write the two helper relations: **HasLUBs** to determine if the partial order satisfies the property that every pair of elements has a least upper bound, and **HasGLBs** to determine if every pair has a greatest lower bound. Just as we did above with the **MaximalElements** procedure, we really only need to write one procedure if we recognize that a partial order satisfies the greatest lower bound property if the inverse relation satisfies the least upper bound property. So we compose **HasLUBs** with the **InverseRelation** procedure to create **HasGLBs**.

```
> HasGLBs := HasLUBs @ InverseRelation:
```

Now we complete the work by coding the **HasLUBs** procedure. We must test whether, for a given relation *R*, each pair *a* and *b* in the domain of *R* has a least upper bound with respect to *R*.

```
> HasLUBs := proc(R::po)
    local DomR, a, b;
    DomR := FindDomain(R);
    for a in DomR do
      for b in DomR do
        if LeastUpperBound(R, {a,b}) = NULL then
          return false;
        end if;
      end do;
    end do;
    return true;
  end proc:
```

Finally, all of the subroutines that go into making up the **IsLattice** program are complete, and we can test it on some examples. Contrast the relations constructed by the **DivRel** procedure versus those made by **DivisorLattice**.

```
> IsLattice(DivRel(20));
```

false **(9.94)**

```
> IsLattice(DivisorLattice(20));
```

true **(9.95)**

Solutions to Computer Projects and Computations and Explorations

Computer Projects 15

Given a partial ordering on a finite set, find a total ordering compatible with it using topological sorting.

Solution: The textbook contains a detailed explanation of topological sorting and summarizes it as Algorithm 1 of Section 9.6.

The set S is initialized to the domain of the given relation. At each step, find a minimal element (using the **MinimalElements** procedure we created above) of S. This minimal element is removed from S and added as the next largest element of the total ordering. This repeats until S is empty and consequently all elements are in the total order.

```
> TopSort := proc(R::po)
    local S, a, T;
    T := [];
    S := FindDomain(R);
    while S <> {} do
      a := MinimalElements(R,S)[1];
      S := S minus {a};
      T := [op(T),a];
    end do;
    return T;
  end proc:
```

We apply this procedure to Fig10.

```
> TopSort(Fig10);
```

["A", "C", "B", "D", "E", "F", "G"] **(9.96)**

Computations and Explorations 1

Display all the different relations on a set with four elements.

Solution: As usual, Maple is much too powerful to solve only the single instance of the general problem suggested by this question. We provide a very simple procedure that will compute all relations on any finite set. This procedure merely constructs the Cartesian product $C = S \times S$ and then makes use of the **powerset** command to obtain all of the relations on the set.

```
> AllRelations := proc(S::set)
    local s, t, C;
    C := {};
    for s in S do
      for t in S do
        C := C union {[s,t]};
      end do;
    end do;
    return combinat[powerset](C);
  end proc:
```

We now test our procedure on a set with 2 elements. (This keeps the output to a reasonable length.)

```
> AllRelations({1,2});
```

{{ }, {[1, 1]}, {[1, 2]}, {[2, 1]}, {[2, 2]}, {[1, 1], [1, 2]}, {[1, 1], [2, 1]}, {[1, 1], [2, 2]}, {[1, 2], [2, 1]}, {[1, 2], [2, 2]}, {[2, 1], [2, 2]}, {[1, 1], [1, 2], [2, 1]}, {[1, 1], [1, 2], [2, 2]}, {[1, 1], [2, 1], [2, 2]}, {[1, 2], [2, 1], [2, 2]}, {[1, 1], [1, 2], [2, 1], [2, 2]}} **(9.97)**

The reader is encouraged to determine the running time and output length for the procedure

when the input set has cardinality 4 or 5. Keep in mind that there are 2^{n^2} relations on a set with n members.

Computations and Explorations 4

Determine how many transitive relations there are on a set with n elements for all positive integers n with $n \leq 7$.

Solution: We will construct each possible $n \times n$ zero-one matrix using an algorithm similar to binary counting. The approach is as follows:

1. For each number from 0 to $2^{n^2} - 1$, we create a list of 0s and 1s that is the base 2 representation of that integer. We can do this with the **convert** command. The syntax **convert(i,base,2)** returns a list whose entries are the base 2 representation of the integer, with the first entry being the 1s place, the second entry the 2s place, the third entry the 4s place, etc.
2. Then create a matrix M whose elements are that list of values. These are all possible 2^{n^2} zero-one matrices (the reader is encouraged to prove this statement). The **Matrix** command with the syntax **Matrix(r,c,L)** produces a matrix of dimension $r \times c$ whose entries are specified by the values in the list L.
3. Finally, evaluate the transitive closure of each of those matrices, using the **Warshall** procedure from Section 9.4 above. We test to see if the matrix is transitive by checking to see if it is equal to its transitive closure. If so, it is counted as a transitive relation.

The implementation is as follows:

```
> CountTransitive := proc(n::posint)
    local i, j, T, M, count;
    count := 0;
    for i from 0 to (2^(n^2) - 1) do
      T := convert(i,base,2);
      M := Matrix(n,n,T);
      if LinearAlgebra[Equal](M,Warshall(M)) then
        count := count + 1;
      end if;
    end do;
    return count;
  end proc:
```

We use our procedure on a relatively small value and leave further computations to the reader.

```
> CountTransitive(3);
```


(9.98)

Computations and Explorations 5

Find the transitive closure of a relation of your choice on a set with at least 20 elements. Either use a relation that corresponds to direct links in a particular transportation or communications network or use a randomly generated relation.

Solution: We will generate a random zero-one matrix with dimension 10 × 10, and then apply Warshall's algorithm to compute the transitive closure.

To generate a random zero-one matrix, we use the **RandomMatrix** command from the

LinearAlgebra package. We will use the form of the command with four arguments. The first two arguments are the row and column dimensions of the matrix. The third argument will be the equation `generator = 0..1`. This indicates that the generated entries should be random integers from the range `0..1` (*i.e.*, they will be 0s and 1s). The final argument is the equation `density = .25`. This tells Maple to only fill about 25 % of the entries in the matrix. The other entries are left 0. (Note that those entries that are randomly selected to be filled are filled with 1 or 0 with equal likelihood.) This produces a fairly sparse matrix and increases the chance that the transitive closure will have entries that are not 1.

```
> LinearAlgebra[RandomMatrix](10, 10, generator = 0..1,
  density = .25);
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \quad \textbf{(9.99)}$$

```
> Warshall((9.99));
```

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \end{bmatrix} \quad \textbf{(9.100)}$$

Exercises

Exercise 1. The `RelToMatrix` procedure converts a relation of type `rel` to a zero-one matrix representation while the `DrawRelation` procedure displays a digraph representation of a `rel`. Write procedures to (a) convert a zero-one matrix representation of a relation to a `rel` and to (b) display a digraph representation of a matrix representation.

Exercise 2. Write a Maple procedure with the signature

`mkRelation(S::set(integer), e::expression)`

that creates the relation $\{(a, b) \in S \times S : e(a, b)\ \textit{is true}\}$. That is, `mkRelation` should return the set of all ordered pairs (a, b) of elements of S for which the expression evaluates to true when a and b are substituted for the variables in the expression e. The input expression e should be a boolean valued Maple expression involving two integer variables x and y as well as operators that take integer operands. For example, your procedure should accept an expression such as

`x + y < x * y`.

Exercise 3. Write a Maple procedure to generate a random relation on a given finite set of integers.

Exercise 4. Use the procedure you wrote in the preceding exercise to investigate the probability that an arbitrary relation has each of the following properties: (a) reflexivity; (b) symmetry; (c) anti-symmetry; and (d) transitivity.

Exercise 5. Write Maple procedures to determine whether a given relation is irreflexive or asymmetric. (See the text for definitions of these properties.)

Exercise 6. Investigate the ratio of the size of an arbitrary relation to the size of its transitive closure. How much does the transitive closure make a relation "grow" on average?

Exercise 7. Examine the function φ defined as follows. For a positive integer n, we define $\varphi(n)$ to be the number of relations on a set of n elements whose transitive closure is the "all" relation. (If A is a set, then the "all" relation on A is the relation $A \times A$ with respect to which every member of A is related to every other member of A, including itself.)

Exercise 8. Write a Maple procedure that finds the antichain with the greatest number of elements in a partial ordering. (See the text for the definition of antichain.)

Exercise 9. The transitive reduction of a relation G is the smallest relation H such that the transitive closure of H is equal to the transitive closure of G. Use Maple to generate some random relations on a set with ten elements and find the transitive reduction of each of these random relations.

Exercise 10. Write a Maple procedure that computes a partial order, given its covering relation.

Exercise 11. Write a Maple procedure to determine whether a given lattice is a Boolean algebra, by checking whether it is distributive and complemented. (See the text for definitions).