11 Matrices and graphs: Transitive closure

Atomic versus structured objects. Directed versus undirected graphs. Transitive closure. Adjacency and connectivity matrix. Boolean matrix multiplication. Efficiency of an algorithm. Asymptotic notation. Warshall's algorithm. Weighted graph. Minimum spanning tree.

In any systematic presentation of data objects, it is useful to distinguish *primitive* or *atomic objects* from *composite* or *structured objects*. In each of the preceding chapters we have seen both types: A bit, a character, or an identifier is usually considered primitive; a word of bits, a string of characters, an array of identifiers is naturally treated as composite. Before proceeding to the most common primitive objects of computation, numbers, let us discuss one of the most important types of structured objects, matrices. Even when matrices are filled with the simplest of primitive objects, bits, they generate interesting problems and useful algorithms.

11.1 Paths in a graph

Syntax diagrams and state diagrams are examples of a type of object that abounds in computer science: A *graph* consists of *nodes* or *vertices*, and of *edges* or *arcs* that connect a pair of nodes. Nodes and edges often have additional information attached to them, such as labels or numbers. If we wish to treat graphs mathematically, we need a definition of these objects.

Directed graph. Let N be the set of n elements $\{1, 2, \ldots, n\}$ and E a binary relation: $E \subseteq N \times N$, also denoted by an arrow, $\rightarrow$. Consider N to be the set of nodes of a directed graph G, and E the set of arcs (directed edges). A directed graph G may be represented by its *adjacency matrix* A (Fig. 11.1), an $n \times n$ boolean matrix whose elements A[i, j] determine the existence of an arc from i to j:

$$A[i, j] = \text{true} \quad \text{iff} \quad i \rightarrow j.$$

An arc is a path of length 1. From A we can derive all paths of any length. This leads to a relation denoted by a double arrow, $\Rightarrow$, called the *transitive closure* of E:

$$i \Rightarrow j, \text{ iff there exists a path from i to j}$$

(i.e., a sequence of arcs $i \rightarrow i_1$, $i_1 \rightarrow i_2$, $i_2 \rightarrow i_3$, $\ldots$, $i_k \rightarrow j$). We accept paths of length 0 (i.e., $i \Rightarrow i$ for all i). This relation $\Rightarrow$ is represented by a matrix $C = A^*$ (Fig. 11.1):

$$C[i, j] = \text{true} \quad \text{iff} \quad i \Rightarrow j.$$

C stands for *connectivity* or *reachability matrix*; $C = A^*$ is also called *transitive hull* or *transitive closure*, since it is the smallest transitive relation that "encloses" E.

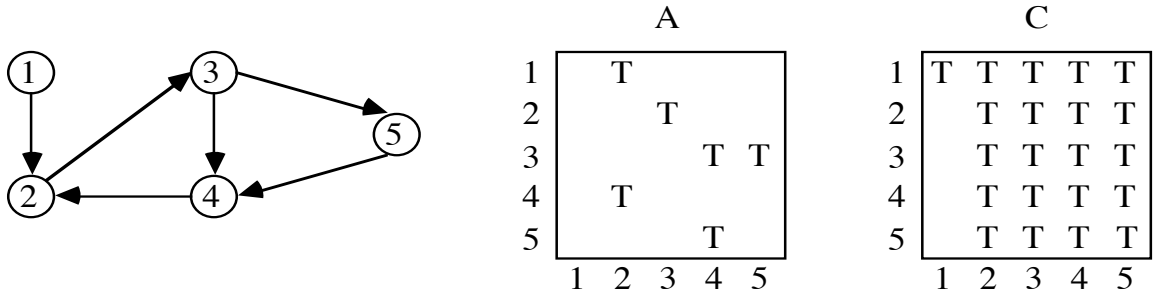

Figure 11.1: Example of a directed graph with its adjacency and connectivity matrix.

(Undirected) graph. If the relation $E \subseteq N \times N$ is *symmetric* [i.e., for every ordered pair (i, j) of nodes it also contains the opposite pair (j, i)] we can identify the two arcs (i, j) and (j, i) with a single *edge*, the unordered pair (i, j). Books on graph theory typically start with the definition of *undirected* graphs (graphs, for short), but we treat them as a special case of directed graphs because the latter occur much more often in computer science. Whereas graphs are based on the concept of an edge *between* two nodes, *directed* graphs embody the concept of one-way *arcs* leading *from* a node *to* another one.

11.2 Boolean matrix multiplication

Let A, B, C be $n \times n$ boolean matrices defined by

```
type  nnboolean: array[1 .. n, 1 .. n] of boolean;
var  A, B, C: nnboolean;
```

The boolean matrix multiplication $C = A \cdot B$ is defined as

$$C[i, j] = \underset{1 \le k \le n}{\mathrm{OR}} \, (A[i, k] \text{ and } B[k, j])$$

and implemented by

```
procedure mmb(var a, b, c: nnboolean);
var  i, j, k: integer;
begin
  for  i := 1  to  n  do
    for  j := 1  to  n  do  begin
      c[i, j] := false;
      for  k := 1  to  n  do  c[i, j] := c[i, j] or (a[i, k] and b[k, j])      (*)
    end
end;
```

Remark: Remember (Section 4.3) that we usually assume the boolean operations 'or' and 'and' to be conditional (i.e., their arguments are evaluated only as far as necessary to determine the value of the expression). An extension of this simple idea leads to an alternative way of coding boolean matrix multiplication that speeds up the innermost loop above for large values of n. Explain why the following code is equivalent to (∗):

```
k := 1;
while  not c[i, j] and (k ≤ n)  do  { c[i, j] := a[i, k] and b[k, j];  k := k + 1 }
```

Multiplication also defines powers, and this gives us a first solution to the problem of computing the transitive closure. If A^L denotes the L-th power of A, the formula

$$A^{L+1}[i, j] = \underset{1 \le k \le n}{\text{OR}} \; (A^L[i, k] \text{ and } A[k, j])$$

has a clear interpretation: There exists a path of length L + 1 from i to j iff, for some node k, there exists a path of length L from i to k and a path of length 1 (a single arc) from k to j. Thus A^2 represents all paths of length 2; in general, A^L represents all paths of length L, for $L \ge 1$:

$A^L[i, j]$ = true iff there exists a path of length L from i to j.

Rather than dealing directly with the adjacency matrix A, it is more convenient to construct the matrix A' = A or I. The identity matrix I has the values 'true' along the diagonal, 'false' everywhere else. Thus in A' all diagonal elements A'[i, i] = true. Then A'^L describes all paths of length $\le L$ (instead of exactly equal to L), for $L \ge 0$. Therefore, the transitive closure is $A^* = A'^{(n-1)}$.

The efficiency of an algorithm is often measured by the number of "elementary" operations that are executed on a given data set. The execution time of an elementary operation [e.g. the binary boolean operators (and, or) used above] does not depend on the operands. To estimate the number of elementary operations performed in boolean matrix multiplication as a function of the matrix size n, we concentrate on the leading terms and neglect the lesser terms. Let us use asymptotic notation in an intuitive way; it is defined formally in Part IV.

The number of operations (and, or), executed by procedure 'mmb' when multiplying two boolean n × n matrices is $\Theta(n^3)$ since each of the nested loops is iterated n times. Hence the cost for computing $A'^{(n-1)}$ by repeatedly multiplying with A' is $\Theta(n^4)$. This algorithm can be improved to $\Theta(n^3 \cdot \log n)$ by repeatedly squaring: $A'^2, A'^4, A'^8, \ldots, A'^k$ where k is the smallest power of 2 with $k \ge n - 1$. It is not necessary to compute exactly $A'^{(n-1)}$. Instead of A'^{13}, for example, it suffices to compute A'^{16}, the next higher power of 2, which contains all paths of length at most 16. In a graph with 14 nodes, this set is equal to the set of all paths of length at most 13.

11.3 Warshall's algorithm

In search of a faster algorithm we consider other ways of iterating over the set of all paths. Instead of iterating over paths of growing length, we iterate over an increasing number of nodes that may be used along a path from node i to node j. This idea leads to an elegant algorithm due to Warshall [War 62]:

Compute a sequence of matrices $B_0, B_1, B_2, \ldots, B_n$:

$B_0[i, j] = A'[i, j] = \text{true}$ iff $i = j$ or $i \rightarrow j$.
$B_1[i, j] = \text{true}$ iff $i \Rightarrow j$ using at most node 1 along the way.
$B_2[i, j] = \text{true}$ iff $i \Rightarrow j$ using at most nodes 1 and 2 along the way.
…
$B_k[i, j] = \text{true}$ iff $i \Rightarrow j$ using at most nodes 1, 2, … , k along the way.

The matrices $B_0, B_1, \ldots$ express the existence of paths that may touch an increasing number of nodes along the way from node i to node j; thus B_n talks about unrestricted paths and is the connectivity matrix $C = B_n$.

An iteration step $B_{k-1} \rightarrow B_k$ is computed by the formula

$B_k[i, j] = B_{k-1}[i, j]$ or $(B_{k-1}[i, k]$ and $B_{k-1}[k, j])$.

The cost for performing one step is $\Theta(n^2)$, the cost for computing the connectivity matrix is therefore $\Theta(n^3)$. A comparison of the formula for Warshall's algorithm with the formula for matrix multiplication shows that the n-ary 'OR' has been replaced by a binary 'or'.

At first sight, the following procedure appears to execute the algorithm specified above, but a closer look reveals that it executes something else: The assignment in the innermost loop computes new values that are used immediately, instead of the old ones.

```
procedure warshall(var a: nnboolean);
var i, j, k: integer;
begin
  for k := 1 to n do
    for i := 1 to n do
      for j := 1 to n do
        a[i, j] := a[i, j] or (a[i, k] and a[k, j])
        { this assignment mixes values of the old and new matrix }
end;
```

A more thorough examination, however, shows that this "naively" programmed procedure computes the correct result *in-place* more efficiently than would direct application of the formulas for the matrices B_k. We encourage you to verify that the replacement of old values by new ones leaves intact all values needed for later steps; that is, show that the following equalities hold:

$$B_k[i, k] = B_{k-1}[i, k] \text{ and } B_k[k, j] = B_{k-1}[k, j].$$

Exercise: Distances in a directed graph: Floyd's algorithm

Modify Warshall's algorithm so that it computes the shortest distance between any pair of nodes in a directed graph where each arc is assigned a length ≥ 0. We assume that the data is given in an $n \times n$ array of reals, where d[i, j] is the length of the arc between node i and node j. If no arc exists, then d[i, j] is set to ∞, a constant that is the largest real number that can be represented on the given computer. Write a procedure 'dist' that works on an array d of type

```
type  nnreal = array[1 .. n, 1 .. n] of real;
```

Think of the meaning of the boolean operations 'and' and 'or' in Warshall's algorithm, and find arithmetic operations that play an analogous role for the problem of computing distances. Explain your reasoning in words and pictures.

Solution

The following procedure 'dist' implements Floyd's algorithm [Flo 62]. We assume that the length of a nonexistent arc is ∞, that $x + \infty = \infty$, and that $\min(x, \infty) = x$ for all x.

```
procedure dist(var d: nnreal);
var  i, j, k: integer;
begin
  for  k := 1  to  n  do
    for  i := 1  to  n  do
      for  j := 1  to  n  do
        d[i, j] := min(d[i, j], d[i, k] + d[k, j])
end;
```

Exercise: Shortest paths

In addition to the distance d[i, j] of the preceding exercise, we wish to compute a shortest path from i to j (i.e., one that realizes this distance). Extend the solution above and write a procedure 'shortestpath' that returns its result in an array 'next' of type:

```
type  nnn = array[1 .. n, 1 .. n] of 0 .. n;
```

next[i, j] contains the next node after i on a shortest path from i to j, or 0 if no such path exists.

Solution

```
procedure shortestpath(var d: nnreal; var next: nnn);
var  i, j, k: integer;
begin
  for  i := 1  to  n  do
    for  j := 1  to  n  do
      if  d[i, j] ≠ ∞  then  next[i, j] := j  else  next[i, j] := 0;
  for  k := 1  to  n  do
    for  i := 1  to  n  do
      for  j := 1  to  n  do
        if  d[i, k] + d[k, j] < d[i, j]  then
          { d[i, j] := d[i, k] + d[k, j];  next[i, j] := next[i, k] }
end;
```

It is easy to prove that next[i, j] = 0 at the end of the algorithm iff d[i, j] = ∞ (i.e., there is no path from i to j).

11.4 Minimum spanning tree in a graph

Consider a *weighted graph* $G = (V, E, w)$, where $V = \{v_1, \dots, v_n\}$ is the set of vertices, $E = \{e_1, \dots, e_m\}$ is the set of edges, each edge e_i is an unordered pair (v_j, v_k) of vertices, and $w: E \to R$ assigns a real number to each edge, which we call its weight. We consider only *connected* graphs G, in the sense that any pair (v_j, v_k) of vertices is connected by a sequence of edges. In the following example, the edges are labeled with their weight (Fig. 11.2).

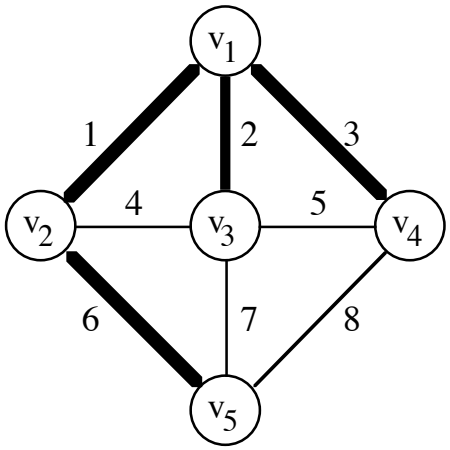

Figure 11.2: Example of a minimum spanning tree.

A *tree* T is a connected graph that contains no circuits: any pair (v_j, v_k) of vertices in T is connected by a unique sequence of edges. A *spanning tree* of a graph G is a subgraph T of G, given by its set of edges $E_T \subseteq E$, that is a tree and satisfies the additional condition of being maximal, in the sense that no edge in $E \setminus E_T$ can be added to T without destroying the tree property. *Observation:* A connected graph G has at least one spanning tree. The *weight* of a spanning tree is the sum of the weights of all its edges. A *minimum spanning tree* is a spanning tree of minimal weight. In Figure 11.2, the bold edges form the minimal spanning tree.

Consider the following two algorithms:

Grow:

$E_T := \varnothing$; *{ initialize to empty set }*
while T is not a spanning tree do
$E_T := E_T \cup$ {a min cost edge that does not form a circuit when added to E_T}

Shrink:

$E_T := E$; *{ initialize to set of all edges }*
while T is not a spanning tree do
$E_T := E_T \setminus$ {a max cost edge that leaves T connected after its removal}

Claim: The "growing algorithm" and "shrinking algorithm" determine a minimum spanning tree.

If T is a spanning tree of G and $e = (v_j, v_k) \notin E_T$, we define Ckt(e, T), "the circuit formed by adding e to T" as the set of edges in E_T that form a path from v_j to v_k. In the example of Fig. 11.2 with the spanning tree shown in bold edges we obtain $Ckt((v_4, v_5), T) = \{(v_4, v_1), (v_1, v_2), (v_2, v_5)\}$.

Exercise

Show that for each edge $e \notin E_T$ there exists exactly one such circuit. Show that for any $e \notin E_T$ and any $t \in Ckt(e, T)$ the graph formed by $(E_T \setminus \{t\}) \cup \{e\}$ is still a spanning tree.

A *local minimum spanning tree* of G is a spanning tree T with the property that there exist no two edges $e \notin E_T$, $t \in Ckt(e, T)$ with $w(e) < w(t)$.

Consider the following 'exchange algorithm', which computes a local minimum spanning tree:

Exchange:

T := any spanning tree;
while there exists $e \notin E_T$, $t \in Ckt(e, T)$ with $w(e) < w(t)$ do
$E_T := (E_T \setminus \{t\}) \cup \{e\}$; *{ exchange }*

Theorem: A local minimum spanning tree for a graph G is a minimum spanning tree.

For the proof of this theorem we need:

Lemma: If T' and T" are arbitrary spanning trees for G, $T' \neq T"$, then there exist $e" \notin E_{T'}$, $e' \notin E_{T"}$, such that $e" \in Ckt(e', T")$ and $e' \in Ckt(e", T')$.

Proof: Since T' and T" are spanning trees for G and $T' \neq T"$, there exists $e" \in E_{T"} \setminus E_{T'}$. Assume that $Ckt(e", T') \subseteq E_{T"}$. Then e" and the edges in Ckt(e", T') form a circuit in T" that contradicts the assumption that T" is a tree. Hence there must be at least one $e' \in Ckt(e", T') \setminus E_{T"}$.
Assume that for all $e' \in Ckt(e", T') \setminus E_{T"}$ we have $e" \notin Ckt(e', T")$. Then

$$\{e"\} \cup (Ckt(e", T') \cap E_{T"}) \cup \bigcup_{e' \in Ckt(e", T') \setminus E_{T"}} Ckt(e', T")$$

forms a circuit in T" that contradicts the proposition that T" is a tree. Hence there must be at least one $e' \in Ckt(e", T') \setminus E_{T"}$ with $e" \in Ckt(e', T")$.

Proof of the Theorem: Assume that T' is a local minimum spanning tree. Let T" be a minimum spanning tree. If $T' \neq T"$ the lemma implies the existence of $e' \in Ckt(e", T') \setminus E_{T"}$ and $e" \in Ckt(e', T") \setminus E_{T'}$.
If $w(e') < w(e")$, the graph defined by the edges $(E_{T"} \setminus \{e"\}) \cup \{e'\}$ is a spanning tree with lower weight than T". Since T" is a minimum spanning tree, this is impossible and it follows that

$$w(e') \geq w(e"). \qquad (*)$$

If $w(e') > w(e")$, the graph defined by the edges $(E_{T'} \setminus \{e'\}) \cup \{e"\}$ is a spanning tree with lower weight than T'. Since T" is a local minimum spanning tree, this is impossible and it follows that

$$w(e') \leq w(e"). \qquad (**)$$

From $(*)$ and $(**)$ it follows that $w(e') = w(e")$ must hold. The graph defined by the edges $(E_{T"} \setminus \{e"\}) \cup \{e'\}$ is still a spanning tree that has the same weight as T". We replace T" by this new minimum spanning tree and continue the replacement process. Since T' and T" have only finitely many edges the process will terminate and T" will become equal to T'. This proves that T" is a minimum spanning tree.

The theorem implies that the tree computed by 'Exchange' is a minimum spanning tree.

Exercises

1. Consider how to extend the transitive closure algorithm based on boolean matrix multiplication so that it computes **(a)** distances and **(b)** a shortest path.

2. Prove that the algorithms 'Grow' and 'Shrink' compute local minimum spanning trees. Thus they are minimum spanning trees by the theorem of Section 11.4.

15.5 Multiplication of complex numbers

Let us turn our attention from noncomputable functions and undecidable problems to very simple functions that are obviously computable, and ask about their complexity: How many primitive operations must be executed in evaluating a specific function? As an example, consider arithmetic operations on real numbers to be primitive, and consider the product z of two complex numbers x and y:

$$x = x_1 + i \cdot x_2 \text{ and } y = y_1 + i \cdot y_2,$$
$$x \cdot y = z = z_1 + i \cdot z_2.$$

The complex product is defined in terms of operations on real numbers as follows:

$$z_1 = x_1 \cdot y_1 - x_2 \cdot y_2,$$
$$z_2 = x_1 \cdot y_2 + x_2 \cdot y_1.$$

It appears that one complex multiplication requires four real multiplications and two real additions/subtractions. Surprisingly, it turns out that multiplications can be traded for additions. We first compute three intermediate variables using one multiplication for each, and then obtain z by additions and subtractions:

$$p_1 = (x_1 + x_2) \cdot (y_1 + y_2),$$
$$p_2 = x_1 \cdot y_1,$$
$$p_3 = x_2 \cdot y_2,$$
$$z_1 = p_2 - p_3,$$
$$z_2 = p_1 - p_2 - p_3.$$

This evaluation of the complex product requires only 3 real multiplications, but 5 real additions / subtractions. This trade of one multiplication for three additions may not look like a good deal in practice, because many computers have arithmetic chips with fast multiplication circuitry. In theory, however, the trade is clearly favorable. The cost of an addition grows linearly in the number of digits, whereas the cost of a multiplication using the standard method grows quadratically. The key idea behind this algorithm is that "linear combinations of k products of sums can generate more than k products of simple terms". Let us exploit this idea in a context where it makes a real difference.

15.6 Complexity of matrix multiplication

The *complexity of an algorithm* is given by its time and space requirements. Time is usually measured by the number of operations executed, space by the number of variables needed at any one time (for input, intermediate results, and output). For a given algorithm it is often easy to count the number of operations performed in the worst and in the best case; it is usually difficult to determine the average number of operations performed (i.e., averaged over all possible input data). Practical algorithms often have time complexities of the order $O(\log n)$, $O(n)$, $O(n \cdot \log n)$, $O(n^2)$, and space complexity of the order $O(n)$, where n measures the size of the input data.

The *complexity of a problem* is defined as the minimal complexity of all algorithms that solve this problem. It is almost always difficult to determine the complexity of a problem, since all possible algorithms must be considered, including those yet unknown. This may lead to surprising results that disprove obvious assumptions.

The complexity of an algorithm is an upper bound for the complexity of the problem solved by this algorithm. An algorithm is a witness for the assertion: You need at most this many operations to solve this problem. But a specific algorithm never provides a *lower bound* on the complexity of a problem - it cannot extinguish the hope for a more efficient algorithm. Occasionally, algorithm designers engage in races lasting decades that result in (theoretically) faster and faster algorithms for solving a given problem. Volker Strassen started such a race with his 1969 paper "Gaussian Elimination Is Not Optimal" [Str 69], where he showed that matrix multiplication requires fewer operations than had commonly been assumed necessary. The race has not yet ended.

The obvious way to multiply two $n \times n$ matrices uses three nested loops, each of which is iterated n times, as we saw in a transitive hull algorithm in Chapter 11. The fact that the obvious algorithm for matrix multiplication is of time complexity $\Theta(n^3)$, however, does not imply that the matrix multiplication problem is of the same complexity.

Strassen's matrix multiplication

The standard algorithm for multiplying two $n \times n$ matrices needs n^3 scalar multiplications and $n^2 \cdot (n-1)$ additions; for the case of 2×2 matrices, eight multiplications and four additions. But seven scalar multiplications suffice if we accept 18 additions/subtractions.

$$\begin{bmatrix} r_{11} & r_{12} \\ r_{21} & r_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}.$$

Evaluate seven expressions, each of which is a product of sums:

$$p_1 = (a_{11} + a_{22}) \cdot (b_{11} + b_{22}),$$
$$p_2 = (a_{21} + a_{22}) \cdot b_{11},$$
$$p_3 = a_{11} \cdot (b_{12} - b_{22}),$$
$$p_4 = a_{22} \cdot (-b_{11} + b_{21}),$$
$$p_5 = (a_{11} + a_{12}) \cdot b_{22},$$
$$p_6 = (-a_{11} + a_{21}) \cdot (b_{11} + b_{12}),$$
$$p_7 = (a_{12} - a_{22}) \cdot (b_{21} + b_{22}).$$

The elements of the product matrix are computed as follows:

$$r_{11} = p_1 + p_4 - p_5 + p_7,$$
$$r_{12} = p_3 + p_5,$$
$$r_{21} = p_2 + p_4,$$
$$r_{22} = p_1 - p_2 + p_3 + p_6.$$

This algorithm does not rely on the commutativity of scalar multiplication. Hence it can be generalized to $n \times n$ matrices using the divide-and-conquer principle. For reasons of simplicity consider n to be a power of 2 (i.e., $n = 2^k$); for other values of n, imagine padding the matrices with rows and columns of zeros up to the next power of 2. An $n \times n$ matrix is partitioned into four $n/2 \times n/2$ matrices:

$$\begin{bmatrix} R_{11} & R_{12} \\ R_{21} & R_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} * \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}.$$

The product of two $n \times n$ matrices by Strassen's method requires seven (not eight) multiplications and 18 additions/subtractions of $n/2 \times n/2$ matrices. For large n, the work required for the 18 additions is negligible compared to the work required for even a single multiplication (why?); thus we have saved one multiplication out of eight, asymptotically at no cost.

Each $n/2 \times n/2$ matrix is again partitioned recursively into four $n/4 \times n/4$ matrices; after $\log_2 n$ partitioning steps we arrive at 1×1 matrices for which matrix multiplication is the primitive scalar multiplication. Let T(n) denote the number of scalar arithmetic operations used by Strassen's method for multiplying two $n \times n$ matrices. For $n > 1$, T(n) obeys the recursive equation

$$T(n) = 7 \cdot T\left(\frac{n}{2}\right) + 18 \cdot \left(\frac{n}{2}\right)^2.$$

If we are only interested in the leading term of the solution, the constants 7 and 2 justify omitting the quadratic term, thus obtaining

$$\begin{aligned} T(n) &= 7 \cdot T\left(\frac{n}{2}\right) = 7 \cdot 7 \cdot T\left(\frac{n}{4}\right) = 7 \cdot 7 \cdot 7 \cdot T\left(\frac{n}{8}\right) = \ldots \\ &= 7^{\log_2 n} \cdot T(1) = n^{\log_2 7} \cdot T(1) \approx n^{2.81} \cdot T(1). \end{aligned}$$

Thus the number of primitive operations required to multiply two $n \times n$ matrices using Strassen's method is proportional to $n^{2.81}$, a statement that we abbreviate as "Strassen's matrix multiplication takes time $\Theta(n^{2.81})$".

Does this asymptotic improvement lead to a more efficient program in practice? Probably not, as the ratio

$$\frac{n^3}{n^{2.81}} \approx n^{0.2} = \sqrt[5]{n}$$

grows too slowly to be of practical importance: For $n \approx 1000$, for example, we have $\sqrt[5]{1024} = 4$ (remember: $2^{10} = 1024$). A factor of 4 is not to be disdained, but there are many ways to win or lose a factor of 4. Trading an algorithm with simple code, such as straightforward matrix multiplication, for another that requires more elaborate bookkeeping, such as Strassen's, can easily result in a fourfold increase of the constant factor that measures the time it takes to execute the body of the innermost loop.