This book is licensed under a Creative Commons Attribution 3.0 License

11. Matrices and graphs: transitive closure

Learning objectives:

- 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.

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 *node*s or *vertices*, and of *edge*s or *arc*s 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, … , n} and E a binary relation: E N $\subseteq \xi$ 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 (Exhibit 11.1), an n ξ n boolean matrix whose elements A[i, j] determine the existence of an arc from i to j:

A[i, j] = true iff $i \rightarrow j$.

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:

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

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

C[i, j] = true iff i ⇒ 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.

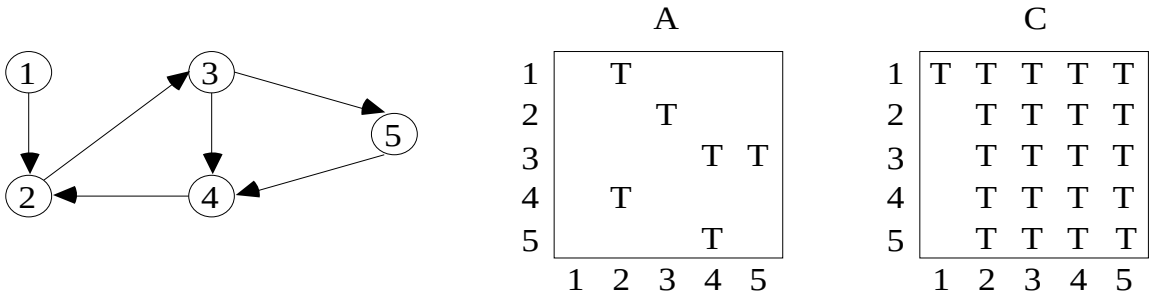

Exhibit 11.1: Example of a directed graph with its adjacency and connectivity matrix.

(Undirected) graph. If the relation E ⊆ N ξ 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.

Boolean matrix multiplication

Let A, B, C be n ξ 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 · B is defined as and implemented by

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

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 (in the section, “Pascal and its dialects: Lingua franca of computer science”) 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;
```

This book is licensed under a Creative Commons Attribution 3.0 License

```
  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+1} denotes the L-th power of A, the formula

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

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 ≥ 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 ≤ L (instead of exactly equal to L), for L ≥ 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 ξ 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 , ... , A'^k where k is the smallest power of 2 with k ≥ 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 1.

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] = true iff i = j or $i \rightarrow j$.

B_1[i, j] = true iff $i \Rightarrow j$ using at most node 1 along the way.

B_2[i, j] = true iff $i \Rightarrow j$ using at most nodes 1 and 2 along the way

...

B_k[i, j] = 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:

$B_k[i, k] = B_{k-1}[i, k]$ 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 ξ 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 + ∞ = ∞, and that min(x, ∞) = 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;
```

This book is licensed under a Creative Commons Attribution 3.0 License

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).

Minimum spanning tree in a graph

Consider a *weighted* graph G = (V, E, w), where V = {v_1, ..., v_n} is the set of vertices, E = {e_1, ... , e_m} is the set of edges, each edge e_i is an unordered pair (v_j, v_k) of vertices, and w: E → 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 (Exhibit 11.2).

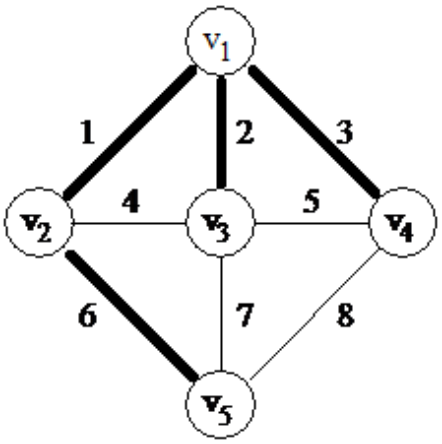

Exhibit 11.2: Example of a minimum spanning tree.

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 Exhibit 11.2, the bold edges form the minimal spanning tree.

Consider the following two algorithms:

Grow:

$E_T := \emptyset$; *{ 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 $e = (v_j, v_k) \notin ET$, we define Ckt(e, T), "the circuit formed by adding e to T" as the set of edges in ET that form a path from v_j to v_k. In the example of Exhibit 11.2 with the spanning tree shown in bold edges we obtain $Ckt((v_4, v_5), T) = \{(v4, v_1), (v_1, v_2), (v_2, v_5)\}$.

Exercise

Show that for each edge $e \notin ET$ there exists exactly one such circuit. Show that for any $e \notin ET$ and any $t \notin Ckt(e, T)$ the graph formed by $(ET \setminus \{t\}) \cup \{e\}$ 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 $e \notin ET$, $t \notin Ckt(e, T)$ 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 $Ckt(e", T') \subseteq_{T"}$. 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 $e" \in Ckt(e', T")$. 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 $e" \in Ckt(e', T")$.

This book is licensed under a Creative Commons Attribution 3.0 License

Proof of the Theorem: Assume that T' is a local minimum spanning tree. Let T" be a minimum spanning tree. If T' ≠ T" the lemma implies the existence of e' ∈ Ckt(e", T') \ E_{T}" and e" ∈ Ckt(e', T") \ E_{T}'.

If w(e') < w(e"), the graph defined by the edges (E_{T}" \ {e"}) ∪ {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') ≥w (e"). (∗)

If w(e') > w(e"), the graph defined by the edges (E_{T}' \ {e'}) ∪ {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') ≤ w(e"). (∗∗)

From (∗) and (∗∗) it follows that w(e') = w(e") must hold. The graph defined by the edges (E_{T}" \ {e"}) ∪ {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 the section entitled “Minimum spanning tree in a graph”.