Chapter 4
Greedy Algorithms

- The main idea of greedy algorithm is look some optimal solution locally and then try to extend globally. Usually the greedy algorithm is efficient.
- The greedy algorithm may not achieve optimal solution for the problem.
- We shall arrive at the greedy algorithm by first considering a dynamic programming approach and then showing that we can always make greedy choices to arrive at an optimal solution.

An activity-selection problem

Suppose we have a set $S = \{a_1, a_2, \ldots, a_n\}$ of n proposed activities that wish to use a resource (for example, a_i are presentations, which need to use one classroom). Each activity a_i has a start time s_i and a finish time f_i, where $0 \leq s_i < f_i < \infty$. If selected, activity a_i takes place during the time internal $[s_i, f_i)$. Activity a_i and a_j are compatible if $[s_i, f_i) \cap [s_j, f_j) = \emptyset$, that is, if $s_i \geq f_j$ or $s_j \geq f_i$. In the activity-selection problem, we wish to select a maximum-size subset of mutually compatible activities. We assume that the activities are sorted in monotonically increasing order of finish time:

$$f_1 \leq f_2 \leq \cdots \leq f_{n-1} \leq f_n.$$

Example: Suppose the activity set S is as follows.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Then the subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities. But it is not the largest subset. The subsets $\{a_1, a_4, a_8, a_{11}\}$ or $\{a_2, a_4, a_9, a_{11}\}$ are largest subsets.

We first try to find some recursive method for the optimal subproblems.

Let S_{ij} denote the subset of some mutually compatible activities that start after activity a_i finishes and end before a_j starts, and suppose such a maximum set is A_{ij}. Let $a_k \in A_{ij}$ be an activity, then we claim that $A_{ik} = S_{ik} \cap A_{ij}$ must be an optimal solution of S_{ik}. Otherwise we will be able to improve A_{ij} and A_{ij} would not be optimal. Similarly, $A_{kj} = S_{kj} \cap A_{ij}$ is also optimal. Therefore $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$. Let $c[i, j]$ denote the size of optimal solution for the set S_{ij}, then we have the following formula

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \emptyset \end{cases}$$

- From the above formula, we can develop a dynamic programming.
- We want to use a simpler method to solve the problem, greedy choice.
- Intuition suggests that we should choose an activity that leaves the resource available for as many other activities as possible.
- We first want to choose a_1 (recall that $f_i, i = 1, \ldots, n$, are sorted) because f_1 is the earliest finish time of any activities.
- Let $S_k = \{a_i \in S : s_i \geq f_k\}$ be the set of activities that start after activity a_k finishes. If we make the greedy choice of activity a_1, then S_1 remains as the only subproblem to solve.

Before we use the above idea to solve the problem, we want to make sure that the solution will be optimal. We have the following theorem.

Theorem Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

Proof. Let A_k be the maximum-size subset of mutually compatible activities in S_k. Let a_j be the activity in A_k with the earliest finish time. If $a_j = a_m$, we are done. Otherwise, a_m must compatible to all the activities in $A_k \backslash \{a_j\}$ since $f_m \leq f_j$. Let $A'_k = A_k \backslash \{a_j\} \cup \{a_m\}$, then $|A'_k| = |A_k|$. So A'_k is a maximum-size subset of mutually compatible activities of S_k. □

In the procedure solves the problem on S_k, where s, f are arrays already sorted according to the finish time.

```
1: procedure RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
2:     m = k + 1
3:     while m ≤ n and s[m] < f[k] do  ▷ find the first activity in S_k to finish
4:         m = m + 1
5:     end while
6:     if  m ≤ n then
7:         return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
8:     else
9:         return ∅
10:    end if
11: end procedure
```

We can call RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$ to obtain the optimal solution for the problem.

The running time is $\Theta(n)$: each activity is examined once in **while** loop. This is assume that the s, f are already sorted. If it is not sorted, then there are sorting algorithms with running time $O(n \log n)$.

```
1:  procedure GREEDY-ACTIVITY-SELECTOR(s, f)
2:      n = s.length
3:      A = {a_1}
4:      k = 1
5:      for m = 2 to n do
6:          if s[m] ≥ f[k] then
7:              A = A ∪ {a_m}
8:              k = m
9:          end if
10:     end for
11:     return A
12: end procedure
```

- The above is an iterative greedy algorithm.
- In the procedure, the variable k indexes the most recent addition to A, corresponding to the activity a_k.
- It is easy to see that the running time for this procedure is also $\Theta(n)$.

Summery of steps of solving activity selection problem.

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution.
3. Show that if we make the greedy choice, then only one subproblem remains.
4. Prove that it is always safe to make the greedy choice. (Steps 3 and 4 can occur in either order.)
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm.

In general, we design greedy algorithms according to the following sequence of steps:

Elements of the greedy strategy

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.
3. Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

Some properties of the problem can be used to see if a greedy algorithm is applicable.

First key ingredient is the greedy-choice property: we can assemble a globally optimal solution by making locally optimal (greedy) choices.

In dynamic programming, we also make choices, but the choices are depends on solved subproblems. In greedy algorithm, we make whatever choice seems best at the moment and then solve the subproblem that remains. So the greedy algorithm is top-down algorithm.

Another thing is the problem exhibits optimal substructure: an optimal solution to the problem contains within it optimal solutions to subproblems.

In greedy algorithm, usually we arrived at a subproblem by having made the greedy choice in the original problem.

Then we need to prove that an optimal solution to the subproblem combined with the greedy choice already made will yield an optimal solution to the original problem.

Since both dynamic programming and greedy programming consider the optimal substructures, sometimes we may be confused which method is suitable for the solution.

Example: The 0-1 knapsack problem is the following. A thief robbing a store finds n items. The ith item is worth v_i dollars and weight w_i pounds, where v_i and w_i are integers. The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack. The problem is which items should he take. (0-1 means for each item take or not take).

In the fractional knapsack problem, the setup is the same, but the thief can take fractions of items, rather than having to take the whole item.

Both knapsack problems have the optimal substructure property.

- For the 0-1 problem, consider the most valuable load that weighs at most W pounds. If we remove item j from this load, the remaining load must be the most valuable load weighing at most $W - w_j$ that the thief can take from the $n - 1$ original items excluding j .
- For the comparable fractional problem, consider that if we remove a weight w of one item j from the optimal load, the remaining load must be the most valuable load weighing at most $W - w$ that the thief can take from the $n - 1$ original items plus $w_j - w$ pounds of item j.

Although the problems are similar, we can solve the fractional knapsack problem by a greedy strategy, but we cannot solve the 0-1 problem by such a strategy.

To solve the fractional problem, we first compute the value per pound v_i/w_i for each item. Obeying a greedy strategy, the thief begins by taking as much as possible of the item with the greatest value per pound. If the supply of that item is exhausted and he can still carry more, he takes as much as possible of the item with the next greatest value per pound, and so forth, until he reaches his weight limit W . Thus, by sorting the items by value per pound, the greedy algorithm runs in $O(n \log n)$ time.

The same greedy strategy does not work for the 0 - 1 knapsack problem.

Consider a small example which has 3 items and a knapsack that can hold 50 pounds. Item 1 weighs 10 pounds and is worth \$60. Item 2 weighs 20 pounds and is worth \$100. Item 3 weighs 30 pounds and is worth \$120. Thus, the value per pound of item 1 is greater than the value per pound of other two items. However, if we take item 1 first, then we will not get the optimal solution.

In the 0 - 1 problem, when we consider whether to include an item in the knapsack, we must compare the solution to the subproblem that includes the item with the solution to the subproblem that excludes the item before we can make the choice. The problem formulated in this way gives rise to many overlapping subproblems a hallmark of dynamic programming.

Huffman codes

We consider how to encode the data of sequence characters into binary codes efficiently. Suppose we have a 100,000 character data file which contains 6 different characters. We know the frequency of these characters.

We may use fixed-length codeword to encode, or use variable-length codeword to encode.

The following table shows the details of the example.

	a	b	c	d	e	f
frequency	45	13	12	16	9	5
fixed-length codeword	000	001	010	011	100	101
variable-length codeword	0	101	100	111	1101	1100

When we use the fixed-length codewords, the encoded file requires 300,000 bits. But if we use the variable-length codewords, the file requires
$(45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4) \times 1000 = 22,400$ bits. The reason of the efficiency of the variable-length encoding is that we use shorter codewords for more frequent characters.

- To use the variable-length codewords to encode, we need to define prefix codes, in which no codeword is also a prefix of some other codeword.
- When we use the prefix encoding, we can simply concatenate the codewords together without causing ambiguous.
- A binary tree can be used to help decode the variable-length codewords.

The tree in Figure 1 is corresponding to the above example of variable-length codewords. If we have a binary string 001011101, then we can start from the root and following the labeled edges. Edge 0 connects to the leave a, edges 101 connect to leave b, etc. So it is decodes as $aabe$.

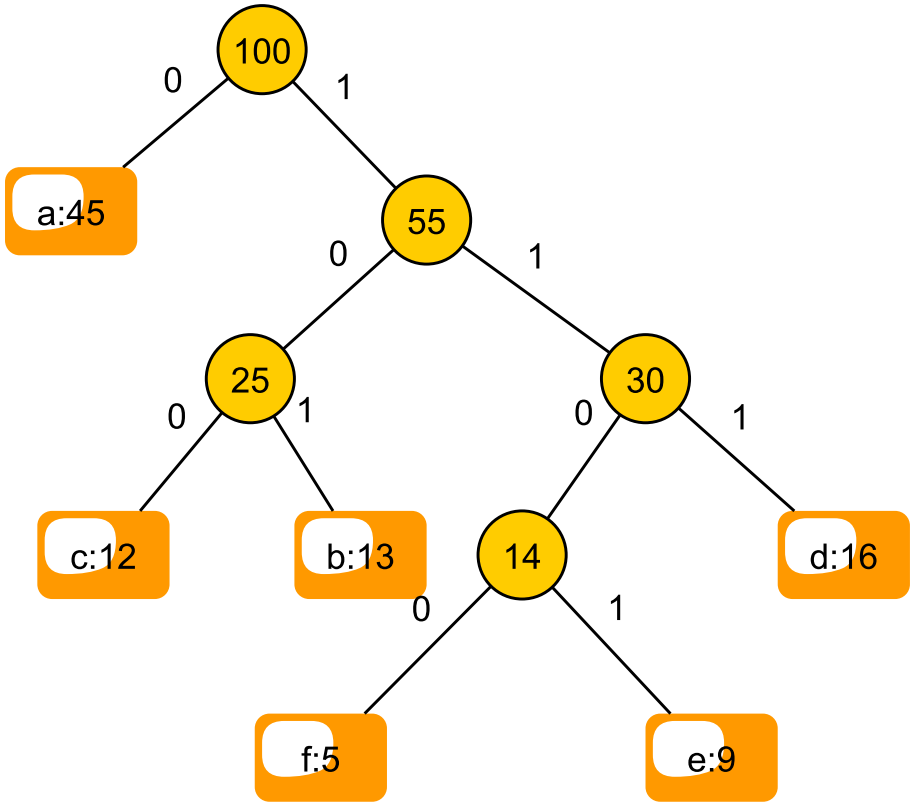

Figure 1: Binary tree for variable-length codewords

Given a tree T corresponding to a prefix code, we can easily compute the number of bits required to encode a file.

For each character c in an alphabet C, let the attribute $c.freq$ denote the frequency of c in the file and let $d_T(c)$ denote the depth of c's leaf in the tree. Note that $d_T(c)$ is also the length of the codeword for character c. The number of bits required to encode a file is thus

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c), \tag{1}$$

which we define as the cost of the tree T.

Huffman invented a greedy algorithm that constructs an optimal prefix code called Huffman code.

The procedure HUFFMAN gives the construction.

In the procedure, C is a set of n characters and each character $c \in C$ associated with an attribute $c.freq$. The procedure EXTRACT-MIN(Q) removes and returns the element with minimum frequency from Q. Q is a min-priority queue.

```
1:  procedure HUFFMAN(C)
2:      n = |C|
3:      Q = C
4:      for i = 1 to n − 1 do
5:          allocate a new node z
6:          z.left = x =EXTRACT-MIN(Q)
7:          z.right = y =EXTRACT-MIN(Q)
8:          z.freq = x.freq + y.freq
9:          INSERT(Q, z)
10:     end for
11:     return EXTRACT-MIN(Q)
12: end procedure
```

This procedure uses a bottom-up method. It begins with two least frequent characters as leaves and merge them to a node with the frequency the sum of these two leaves. The node is then put back to the pool. The **for** loop runs n times. We use a min-priority queue Q (minimum-heap: the first element is the minimum element), then the running time for the procedure will be $O(n \log n)$.

Next we need to prove that the procedure really created an optimal code.

Lemma 4.2.1 If an optimal code for a file is represented by a binary tree, then the tree is full binary, that is, every nonleaf node has two children.

Proof. Assume that there is an internal node A which has only one child B. Then we can remove the node A and the edge between A and B, and move B to the position of A. The resulting binary tree also represents the same file, but uses fewer bits. This is a contradiction. □

Lemma 4.2.2 Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

Proof. Let tree T be an optimal prefix code for the alphabet. Let a and b be two characters that are sibling leaves of maximum depth in T (Lemma 4.2.1 guarantees the existence of a and b). We may assume that $a.freq \leq b.freq$ and $x.freq \leq y.freq$. We have $x.freq \leq a.freq$ and $y.freq \leq b.freq$.

If $x.freq = b.freq$, then we have

$x.freq = b.freq = y.freq = a.freq$,

so the lemma is true. So we assume that $x.freq \neq b.freq$. Now we construct a tree T' from T by exchanging the positions of a and x. Then exchange the positions of b and y to obtain a tree T''.

Since $x \neq b$, x and y are sibling leaves in T''. By equation (1) the difference in cost between T and T' , $D = B(T) - B(T')$ is

$$\begin{aligned} D &= \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) \\ &= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_{T'}(x) - a.freq \cdot d_{T'}(a) \\ &= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) - a.freq \cdot d_T(x) \\ &= (a.freq - x.freq)(d_T(a) - d_T(x)) \\ &\geq 0. \end{aligned}$$

Similarly, we have $B(T') - B(T'') \geq 0$. Therefore $B(T) \geq B(T'')$. Since T is optimal, we must have $B(T) = B(T'')$. So T'' is also optimal. □

Next we consider the optimal substructure property for the optimal prefix codes.

Let C be an alphabet with frequency $c.freq$ for each $c \in C$. Let x and y be two characters in C with minimum frequency. Let z be a new character with $z.freq = x.freq + y.freq$ and
$C' = (C \backslash \{x, y\}) \cup \{z\}$.

Lemma 4.2.3 Let T' be any tree representing an optimal prefix code for alphabet C'. Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C.

Proof. For each character $c \in C \backslash \{x, y\}$, we have $d_T(c) = d_{T'}(c)$. Since $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we have

$$\begin{aligned} x.freq \cdot d_T(x) + y.freq \cdot d_T(y) &= (x.freq + y.freq)(d_{T'}(z) + 1) \\ &= z.freq \cdot d_{T'}(z) + (x.freq + y.freq), \end{aligned}$$

from which we have

$$B(T) = B(T') + x.freq + y.freq.$$

We now prove the lemma by contradiction. Suppose that T does not represent an optimal prefix code for C. Then there exists an optimal tree T'' such that $B(T'') < B(T)$. By Lemma 4.2.2, we may assume that T'' has x and y as siblings. Let T''' be the tree T'' with the common parent of x and y replaced by a leaf z with frequency $z.freq = x.freq + y.freq$. Then

$$\begin{aligned} B(T''') &= B(T'') - x.freq - y.freq \\ &< B(T) - x.freq - y.freq \\ &= B(T'), \end{aligned}$$

yielding a contradiction to the assumption that T' represents an optimal prefix code for C'. □

From the above two Lemmas, we obtain the following theorem.

Theorem 4.2.4

Procedure HUFFMAN produces an optimal prefix code.

Minimum spanning tree

Let $G = (V, E)$ be a undirected connected graph with a weight function: $E \to \mathbb{R}$. An acyclic set $T \subseteq E$ that connects all of the vertices of G is called a spamming tree of G. We want to find T whose total weight

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

is minimum. Such a problem is called minimum-spanning-tree problem.

Representations of a graph

There are two representations of a graph. For the adjacency-matrix representation of a graph $G = (V, E)$, we assume that vertices are labeled as $1, 2, \ldots, |V|$. The representation is a $|V| \times |V|$ matrix $A = (a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{otherwise} \end{cases}$$

For weighted graph, instead of using 1 in the matrix, we can use $w(i, j)$ as a_{ij} if $(i, j) \in E$.

The adjacency-list representation of a graph $G = (V, E)$ consists of an array adj of $|V|$ lists, one for each vertex in V. For each $u \in V$, the adjacency list $adj[u]$ contains all the vertices adjacent to u in G. For weighted graph, we simply store the weight $w(u, v)$ of the edge (u, v) with vertex v in u's list.

An adjacency-list representation requires $\Theta(V + E)$ memory space, while an adjacency-matrix representation needs $\Theta(V^2)$ space.

The Breath-first search

Given a graph $G(V, E)$ and a distinguished source vertex v, we consider search algorithms, which explore the edges of G to discover every vertex that is reachable from s.

The Breath-first search procedure assumes that the input graph is represented using adjacency list. The algorithm constructs a breadth-first tree, initially containing only its root, which is the source vertex s. Whenever the search discovered a vertex v in the course of scanning the adjacency list of an already discovered vertex u, the vertex v and the edge (u, v) are added to the tree.

For each vertex $u \in V$, we define several attributes on it. $u.\pi$ denote u's predecessor (in the breadth-first tree). If u has no predecessor, then $u.\pi = \text{NIL}$. The attribute $u.d$ holds the distance from the source vertex s to vertex u.

The algorithm uses a FIFO queue Q. The attribute $u.color$ gives a color to u to indicate if it is processed. The white color means it is not processed, the gray color means it is put into the queue, and the black color means it has been processed. The attribute $u.d$ holds the distance from the source s to vertex u computed by the algorithm.

```
1:  procedure BFS(G, s)
2:      for each vertex u ∈ G.V − {s} do
3:          u.color = WHITE
4:          u.d = ∞
5:          u.π = NIL
6:      end for
7:      s.color = GRAY
8:      s.d = 0
9:      s.π = NIL
10:     Q = ∅
11:     ENQUEUE(Q, s)
12:     while Q ≠ ∅ do
13:         u =DEQUEUE(Q)
14:         for each v ∈ G.adj[u] do
15:             if v.color == WHITE then
16:                 v.color = GRAY
```

```
17:                 v.d = u.d + 1
18:                 v.π = u
19:                 ENQUEUE(Q, v)
20:             end if
21:         end for
22:         u.color = BLACK
23:     end while
24: end procedure
```

In this procedure, initialization uses $O(V)$ time, the queue operation is also using $O(V)$ time because each vertex goes to the queue once. The total time spent in scanning adjacency lists is $O(E)$. The running time of BFS procedure is $O(V + E)$.

Define the shortest-path distance $\delta(s, v)$ from s to v as the minimum number of edges in any path from vertex s to vertex v; if there is no path from s to v, then $\delta(s, v) = \infty$. We call a path of length $\delta(s, v)$ from s to v a shortest path from s to v. Before showing that breadth-first search correctly computes shortest path distances, we investigate an important property of shortest-path distances.

Lemma 4.3.1 Let $G = (V, E)$ be a directed or undirected graph, and let $s \in V$ be an arbitrary vertex. Then for any edge $(u, v) \in E$, $\delta(s, v) \leq \delta(s, u) + 1$.

The proof of the lemma is simple.

Lemma 4.3.2 Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source $s \in V$. Then upon termination, for each vertex $v \in V$, the value $v.d$ composed by BFS satisfies $v.d \geq \delta(s, v)$.

Proof. We use induction on the number of ENQUEUE operations. Our inductive hypothesis is that $v.d \geq \delta(s, v)$ for all $v \in V$.

The basis of the induction is the situation immediately after enqueuing s in BFS. The inductive hypothesis holds here, because $s.d = 0 = \delta(s, s)$ and $v.d = \infty \geq \delta(s, v)$ for all $v \in V - \{s\}$.

search from a vertex u. The inductive hypothesis implies that $u.d \geq \delta$. From the assignment performed by line 17 and from Lemma 4.3.1, we obtain

$$v.d = u.d + 1 \geq \delta(s, u) + 1 \geq \delta(s, v).$$

Vertex v is then enqueued, and it is never enqueued again because it is also grayed and the then clause of lines 15 - 19 is executed only for white vertices. Thus, the value of $v.d$ never changes again, and the inductive hypothesis is maintained. □

Lemma 4.3.3 Suppose that during the execution of BFS on a graph $G = (V, E)$, the queue Q contains the vertices $\langle v_1, v_2, \ldots, v_r \rangle$, where v_1 is the head of Q and v_r is the tail. Then $v_r.d \leq v_1.d + 1$ and $v_i.d \leq v_{i+1}.d$ for $i = 1, 2, .r - 1$.

Proof. The proof is by induction on the number of queue operations. Initially, when the queue contains only s, the lemma certainly holds.

For the inductive step, we must prove that the lemma holds after both dequeuing and enqueuing a vertex. If the head v_1 of the queue is dequeued, v_2 becomes the new head. (If the queue becomes empty, then the lemma holds vacuously.) By the inductive hypothesis, $v_1.d \leq v_2.d$. But then we have $v_r.d \leq v_1.d + 1 \leq v_2.d + 1$, and the remaining inequalities are unaffected. Thus, the lemma follows with v_2 as the head.

When we enqueue a vertex v in line 19 of BFS, it becomes v_{r+1}. At that time, we have already removed vertex u, whose adjacency list is currently being scanned, from the queue Q, and by the inductive hypothesis, the new head $v_1.d \geq u.d$. Thus, $v_{r+1}.d = v.d = u.d + 1 \leq v_1.d + 1$. From the inductive hypothesis, we also have $v_r.d \leq u.d + 1$, and so $v_r \leq u.d + 1 = v.d = v_{r+1}.d$, and the remaining inequalities are unaffected. Thus, the lemma follows when v is enqueued. □

Corollary 4.3.4 Suppose that vertices v_i and v_j are enqueued during the execution of BFS, and that v_i is enqueued before v_j. Then $v_i.d \leq v_j.d$ at the time that v_j is enqueued.

Theorem 4.3.5 Let $G = (V, E)$ be a directed or undirected graph, and suppose the BFS is run on G from a given source vertex $s \in V$. Then during its execution, BFS discovers every vertex $v \in V$ that is reachable from the source s, and upon termination, $v.d = \delta(s.v)$ for all $v \in V$. Moreover, for any $v \neq s$ that is reachable from s, one of the shortest paths from s to v is a shortest path from s to $v.\pi$ followed by the edge $(v.\pi, v)$.

Proof. Assume, for the purpose of contradiction, that some vertex receives a d value not equal to its shortest-path distance. Let v be the vertex with minimum $\delta(s, v)$ that receives such an incorrect d value; clearly $v \neq s$. By Lemma 4.3.2, $v.d \geq \delta(s, v)$, and thus we have that $v.d > \delta(s, v)$. Vertex v must be reachable from s, for if it is not, then $\delta(s, v) = \infty \geq v.d$. Let u be the vertex immediately preceding v on a shortest path from s to v, so that $\delta(s, v) = \delta(s, u) + 1$. Because $\delta(s, u) < \delta(s, v)$, and because of how we chose v, we have $u.d = \delta(s, u)$. Putting these properties together, we have

$$v.d \quad > \quad \delta(s, v) = \delta(s, u) + 1 = u.d + 1. \tag{2}$$

Now consider the time when BFS chooses to dequeue vertex u from Q. At this time, vertex v is either white, gray, or black. We shall show that in each of these cases, we derive a contradiction to inequality (2). If v is white, then line 17 sets $v.d = u.d + 1$, contradicting inequality (2). If v is black, then it was already removed from the queue and, by Corollary, we have $v.d \leq u.d$, again contradicting inequality (2). If v is gray, then it was painted gray upon dequeuing some vertex w, which was removed from Q earlier than u and for which $v.d = w.d + 1$. By Corollary, however, $w.d \leq u.d$, and so we have $v.d = w.d + 1 \leq u.d + 1$, once again contradicting inequality (2).

Thus we conclude that $v.d = \delta(s, v)$ for all $v \in V$.

All vertices v reachable from s must be discovered, for otherwise they would have $\infty = v.d > \delta(s, v)$. To conclude the proof of the theorem, observe that if $v.\pi = u$, then $v.d = u.d + 1$. Thus, we can obtain a shortest path from s to v by taking a shortest path from s to $v.\pi$ and then traversing the edge $(v.\pi, v)$. □

The Depth-first Search

DFS may be composed of several trees that is different from the BFS. Instead define a predecessor tree, we define predecessor subgraph (may a forest) as $G_\pi = (V, E_\pi)$, where

$$E_\pi = \{(v.\pi, v) : v \in V \text{ and } v.\pi \neq \text{NIL}\}.$$

For the DFS, we visit the vertex in depth recursively and then search backtracks (actually used stack since we use recursive procedure calling). We use two attributes to record time-stamps. $v.d$ records when v is first discovered (grayed v), and $v.f$ records the the search finishes v's adjacency list (blackens v).

```
1:  procedure DFS(G)
2:      for each vertex u ∈ G.V do
3:          u.color = WHITE
4:          u.π = NIL
5:      end for
6:      time = 0
7:      for each u ∈ G.V do
8:          if v.color == WHITE then
9:              DFS-Visit(G, u)
10:         end if
11:     end for
12: end procedure
```

```
1:  procedure DFS-Visit(G, u)
2:      time = time + 1
3:      u.d = time
4:      u.color = GRAY
5:      for each v ∈ G.adj[u] do
6:          if v.color == WHITE then
7:              v.π = u
8:              DFS-Visit(G, v)
9:          end if
10:     end for
11:     u.color = BLACK
12:     time = time + 1
13:     u.f = time
14: end procedure
```

Since $\sum_{v \in V} |adj[v]| = \Theta(E)$ in DFS-Visit, and the initialization and the for loop in line 7 of DFS execute $\Theta(V)$ time, the running time for the DFS is $\Theta(V + E)$.

As an application of DFS procedure, we consider a topological sort of a directed acyclic graph, or a dag. A topological sort of a dag $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v) then u appears before v in the ordering. Note that if the graph contains a cycle, then no liner ordering is possible.

```
1: procedure TOPOLOGICAL-SORT(G)
2:     call DFS(G) to compute finishing times v.f for each vertex
   v
3:     as each vertex is finished, insert it onto the front of a linked
   list
4:     return the linked list of vertices
5: end procedure
```

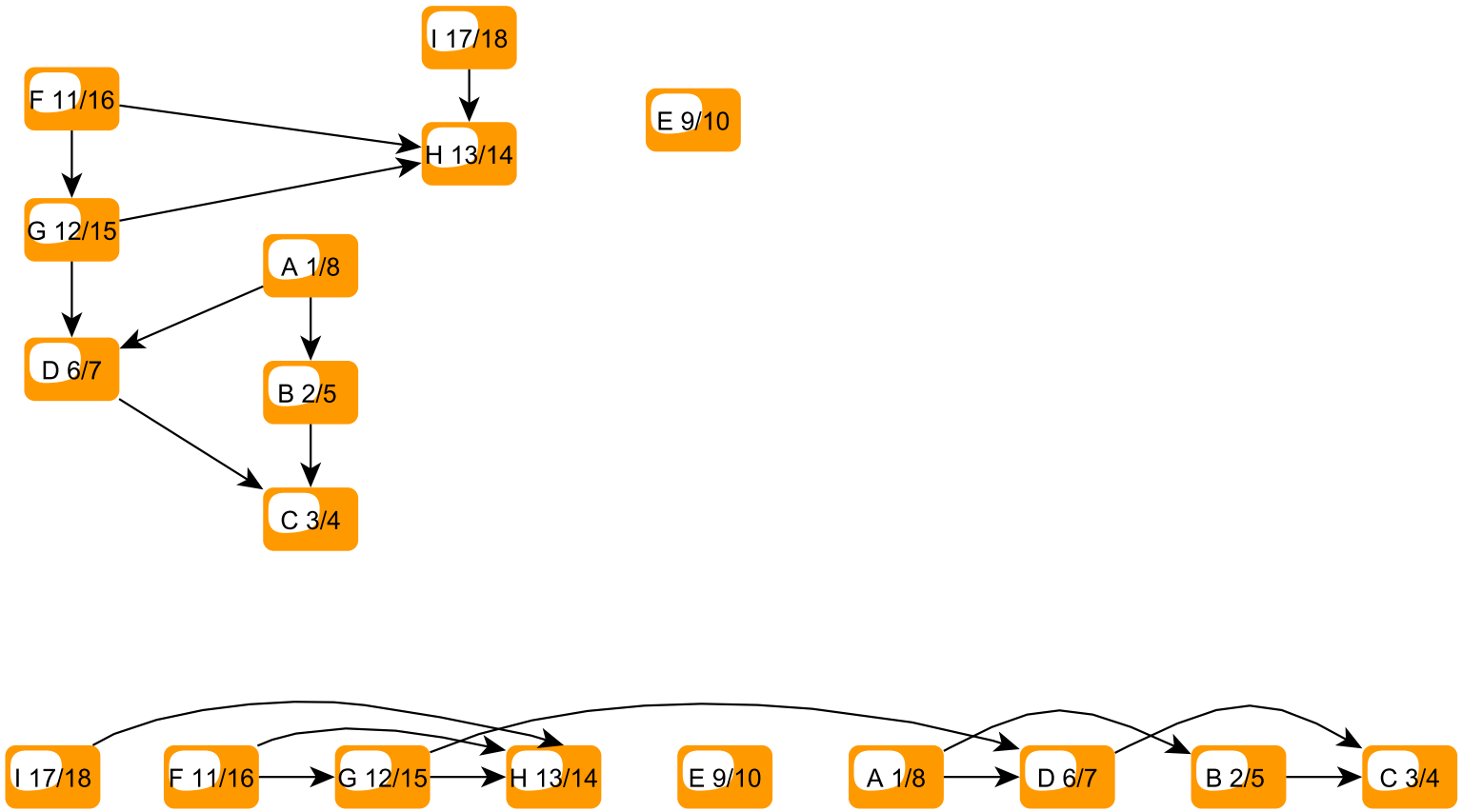

Figure 2: Example of topological sort

The Figure 2 shows a small example of topological sort of a dag. The top part is the original graph with labels indicating the discovery and finishing times under the DFS. The lower part shows the results of the topological sort. In the sorting processing, the vertex v with smallest $v.f$ is first put into the linked list, then second smallest, and so on.

We can perform a topological sort in time $\Theta(V + E)$, since DFS takes $\Theta(V + E)$ time and it takes $O(1)$ time to insert each of the $|V|$ vertices onto the front of the linked list.

Greedy algorithm for MST

We can use a greedy approach to the problem. The main idea is that we can grow the minimum tree one edge at a time such that the subset chosen is a subset of some minimum spanning tree.

Suppose a subset A is chosen, we can determine an edge (u, v) that we can add to A such that $A \cup \{(u, v)\}$ is also a subset of a minimum spanning tree. We call such an edge a **safe edge** for A.

```
1: procedure GENERIC-MST(G, w)
2:     A = ∅
3:     while A does not form a spanning tree do
4:         find an edge (u, v) that is safe for A
5:         A = A ∪ {(u, v)}
6:     end while
7:     return A
8: end procedure
```

The initialization $A = \emptyset$ in the procedure satisfies the loop invariant. The maintenance is done by adding safe edge.

We need to prove that safe edge exists and we have some method to find out it.

To prove that, we need some definitions.

- A **cut** (S, V_S) of an undirected graph $G = (V, E)$ is a partition of V.
- We say that an edge $(u, v) \in E$ **crosses** the cut $(S, V - S)$ if one of its endpoints is in S and the other is in $V - S$. We say that a cut **respects** a set A of edges, if no edge in A crosses the cut.
- An edge is a **light edge** crossing a cut if its weight is the minimum of any edge crossing the cut.

In general we will say that an edge is a light edge satisfying a given property if its weight is the minimum of any edges satisfying the property.

Theorem 4.3.6 Let $G = (V, E)$ be a connected, undirected graph with a real-values weight function$w$ defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V - S)$. Then (u, v) is safe for A.

Proof. Let T be a minimum spanning tree that includes A, and assume that T does not contain the light edge (u, v), since otherwise we are done. We will construct another minimum spanning tree T' that includes $A \cup \{(u, v)\}$.

If the edge (u, v) is added to T, then it forms a cycle with the edges on the simple path p from u to v in T. Since u and v are on opposite sides of the cut $(S, v - S)$, at least one edge in T lies on the simple path p and also crosses the cut. Let (x, y) be any such edge. The edge (x, y) is not in A, because the cut respects A. Since (x, y) is on the unique simple path from u to v in T , removing (x, y) breaks T into two components. Adding (u, v) reconnects them to form a new spanning tree $T' = (T - \{(x, y)\}) \cup \{(u, v)\}$.

We next show that T' is a minimum spanning tree. Since (u, v) is a light edge crossing $(S, V - S)$ and (x, y) also crosses this cut, $w(u, v) \leq w(x, y)$. Therefore,

$$\begin{aligned} w(T') &= w(T) - w(x, y) + w(u, v) \\ &\leq w(T). \end{aligned}$$

But T is a minimum spanning tree , so $w(T) \leq w(T')$. Therefore $w(T) = w(T')$ and T' must be a minimum spanning tree. Since $A \subseteq T'$ and $A \cup \{(u, v)\} \subseteq T'$, (u, v) is safe for A. □

In the procedure GENERIC-MST and in Theorem 4.3.6, the set A is a subset of edges. A must be acyclic, but not necessary connected. So A is a forest, and each of the connected components is a tree.

The **while** loop in GENERIC-MST executes $|V| - 1$ times because the spanning tree has $|V| - 1$ edges and each loop adds one edge to A.

Corollary Let $G = (V, E)$ be a connected, undirected graph with a weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, and let $C = (V_C, E_C)$ be a connected component (tree) in the forest $G_A = (V, A)$. If (u, v) is a light edge connecting C to some other component in G_A, then (u, v) is safe for A.

Proof. The cut $(V_C, V - V_C)$ respects A, and (u, v) is a light edge for this cut. Therefore, (u, v) is safe for A. □

The algorithms of Krukal and Prim

To use the GENERIC-MST, we need some method to find safe edge in the statement line 4 of the procedure. Two algorithms described here elaborate on that method.

For the implementation of graphs, we use the adjacency lists.

In Kruskal's algorithm, the set A is a forest whose vertices are all those of the given graph. The safe edge added to A is always a least-weight edge in the graph that connects two disjoint components.

To implement Kruskal algorithm, we need some simple procedures to maintain the "forest". For a vertex x, we assign a parent $x.p$ (some vertex which represent the subset that contains x) and a rank $p.rank$ (an integer which can be viewed as the level in a tree that x sits) to it. To initialize the setting, the following procedure is called.

1: **procedure** MAKE-SET(x)
2: $\quad x.p = x$
3: $\quad x.rank = 0$
4: **end procedure**

Then we need to merge some subsets of the vertices into one subset. Suppose x and y are two vertices in two disjoint subsets. We want to merge them to one subset. Then basically we just need to change the parent for one of the vertices. The following procedure decides how to change one parent.

```
1: procedure LINK(x, y)
2:     if x.rank > y.rank then
3:         y.p = x
4:     else
5:         x.p = y
6:         if x.rank == y.rank then
7:             y.rank = y.rank + 1
8:         end if
9:     end if
10: end procedure
```

The procedure uses the vertex with larger rank as the parent. In this way, we can keep the height of the tree lower. Now if you have changed the parent of x, then all the vertices in the same subset need to be changed. The procedure FIND-SET is used to find the parent of a vertex in general.

```
1: procedure FIND-SET(x)
2:     if x ≠ x.p then
3:         x.p = FIND-SET (x.p)
4:     end if
5:     return x.p
6: end procedure
```

Now the UNION is simple.

procedure UNION(x, y)
    LINK(FIND-SET(x), FIND-SET(y))
end procedure

```
1: procedure MST-Kruskal(G, w)
2:     A = ∅
3:     for each vertex v ∈ G.V do
4:         Make-Set(v)
5:     end for
6:     sort the edges of G.E into nondecreasing order by weight w
7:     for each (u, v) ∈ G.E, taken in nondecreasing order by
   weight do
8:         if Find-Set(u) ≠ Find-Set(v) then
9:             A = A ∪ {(u, v)}
10:            Union(u, v)
11:        end if
12:    end for
13:    return A
14: end procedure
```

The **for** loop in line 7 examines edges in order of weight, from lowest to highest. The loop checks, for each edge (u, v), whether u and v belong to the same subtree. If they do, then they cannot be added to the forest, and so edge is discarded. Otherwise, the edge (u, v) is added to A and two subtrees are merged to one subtree.

Now we consider the running time of MST-Kruskal. The sort in line 6 is $O(E \log E)$. When we use the Link to merge the subtrees, the height of the tree is $\log V$. The **for** loop in line 7 takes $O((E)$ Find-Set and Union operations on the disjoint forest. Along with the $|V|$ Make-Set operations, these take a total of $O((V + E) \log V)$ time. Since G is connected, we have $|V| - 1 \leq |E| \leq |V|^2$. So we have that the running time of Kruskal's algorithm is $O(E \log V)$.

The Prim's algorithm is also based on the generic greedy algorithm. In Prim's algorithm, the set A forms a single tree. The safe edge added to A is a least-weight edge connecting the tree to a vertex not in tree.

In the Prim's algorithm, each vertex v is assigned an attribute key which is the minimum weight of any edge connecting v to a vertex in the tree. If no such an edge exist, $v.key = \infty$.

Another attribute $v.\pi$ names the parent of v in the tree.

We use a min-priority queue Q based on the key attributes to house all the vertices not in the tree yet. The EXTRACT-MIN (Q) will return the minimum element and then delete it from Q. The algorithm implicitly maintains the set A as

$$A \quad = \quad \{(v, v.\pi) : v \in V - \{r\} - Q\}.$$

The procedure can choose any vertex r to start finding the MST.

```
1: procedure MST-PRIM(G, w, r)
2:     for each u ∈ G.V do                                  ▷ initial for Q
3:         u.key = ∞
4:         u.π = NIL
5:     end for
6:     r.key = 0                                            ▷ initial r
7:     Q = G.V
8:     while Q ≠ ∅ do
9:         u = EXTRACT-MIN(Q) ▷ move lightest vertex from Q to
   A
10:        for each v ∈ G.adj[u] do ▷ update the key of vertices in
   Q
11:            if v ∈ Q and w(u, v) < v.key then
12:                v.π = u
13:                v.key = w(u.v)
```

14: **end if**

15: **end for**

16: **end while**

17: **end procedure**

The minimum spanning tree now is $A = \{(v, v.\pi) : v \in V - \{r\}\}$ with the root r.

The initial Q uses $O(V)$ time and we can arrange Q as min-heap.

The **while** in line 8 executes $|V|$ times, and since each EXTRACT-MIN operation takes $O(\log V)$ time, the total time for all calls to EXTRACT-MIN is $O(V \log V)$.

The **for** loop in line 10 executes $O(E)$ times altogether, since the sum of the lengths of all adjacency lists is $2|E|$. Since the Q is a min-heap, the operations are in $O(\log V)$ time. The total time for Prime's algorithm is $O(V \log V + E \log V) = O(E \log V)$.

If we use a Fibonacci heap (which will be discussed later), the running time of Prim's algorithm improves to $O(E + V \lg V)$.

Shortest paths

In the shortest-paths problem, we are given a weighted, directed graph$G = (V, E)$, with weight function $w : E \to \mathbb{R}$. The **weight** $w(p)$ of $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

The shortest-path weight $\delta(u, v)$ from u to v is defined as:

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there is a path from } u \text{ to } v, \\ \infty & \text{otherwise} \end{cases}$$

A shortest path from vertex u to vertex v is defined as any path p with weight $w(p) = \delta(u, v)$.

For the shortest-path problem, we may consider single-destination (or single source) shortest path which finds a shortest path to a given destination from each vertex (or from the source to each vertex).

We also can consider single-pair shortest path which finds a shortest path from a source vertex v to a vertex u. However, all the known algorithms for single-pair shortest path have the same worst-case asymptotic running time as the best single-source algorithms. So we mainly consider the single-destination short path problem.

To use greedy algorithm, we need some optimal substructure of the shortest path problem. We have the following lemma.

Lemma 4.4.1 Suppose a directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$ is given. Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from vertex v_0 to vertex v_k. For any i and j, $0 \leq i \leq j \leq k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of p from v_i to v_j. Then p_{ij} is a shortest path from v_i to v_j.

Proof. If p_{ij} is not a shortest path, then there is a shortest path $p'_{ij} = \langle v_i, v'_{i+1}, \ldots, v'_{j-1}, v_j \rangle$ such that $w(p'_{ij}) < w(p_{ij})$. But $p' = \langle v_0, v_1, \ldots, v_i, v'_{i+1}, \ldots, v'_{j-1}, v_j, \ldots v_k \rangle$ is a path from v_0 to v_k with $w(p') < w(p)$ which is impossible. □

The Bellman-Ford algorithm

In some applications of the shortest paths problem, the graph may include some edges with negative weights. Consider the single-source shortest path problem. If the graph contains a negative-weight cycle reachable from the source vertex s, then the shortest path weight are not well defined. Because the path can repeat the cycle any number of times, that makes the weight smaller than any given number. So when we treat a graph with negative weight edges, we only consider those graphs that do not contain any negative-weight cycle.

A shortest path in a graph contains no cycle. If there is a cycle among the path with no-negative weight, then we can remove the cycle. In fact, in this case the cycle cannot has positive weight, otherwise the path would not be the shortest.

For the single-source shortest path problem of a weighted graph $G = (V, E)$, we are finding a shortest-paths tree $G' = (V', E')$ rooted at the source vertex s, where $V' \subseteq V, E' \subseteq E$, satisfying

1. V' is the set of vertices reachable from s in G,
2. G' forms a rooted tree with root s, and
3. for all $v \in V'$, the unique simple path from s to v in G' is a shortest path from s to v in G.

To compute shortest path, we maintain two attributes for a vertex v in the graph.

For each vertex $v \in G.V$, we define a predecessor $v.\pi$ that is either another vertex or NIL.

In the shortest path algorithm we set the π attributes so that the chain of predecessors originating at a vertex v runs backwards along a shortest path from s to v.

We also define the predecessor subgraph $G_\pi = (V_\pi, E_\pi)$ induced by the π values. In this subgraph, V_π is the set of vertices of G with non-NIL predecessors, plus the source s:

$$V_\pi = \{v \in V : v.\pi \neq NIL\} \cup \{s\}.$$

The directed edge set E_π is the set of edges induced by the π values for vertices in V_π:

$$E_\pi = \{(v.\pi, v) \in E : v \in V_\pi - \{s\}\}.$$

Another attribute for a vertex v is $v.d$ which is an upper bound on the weight of a shortest path from source s to v. We call $v.d$ a shortest-path estimate.

We can use the following $\Theta(V)$-time procedure to initialize these attributes.

```
1: procedure INITIALIZE-SINGLE-SOURCE(G, s)
2:     for each v ∈ G.V do
3:         v.d = ∞
4:         v.π = NIL
5:     end for
6:     s.d = 0
7: end procedure
```

The next procedure of **relaxing** an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u, and updating $v.d$ and $v.\pi$.

```
1: procedure RELAX(u, v, w)
2:     if v.d > u.d + w(u, v) then
3:         v.d = u.d + w(u, v)
4:         v.π = u
5:     end if
6: end procedure
```

The Bellman-Ford algorithm solves the single-source shortest path problem in general case in which edge weights may be negative.

The algorithm returns a boolean value indicating if there is a negative cycle that is reachable from the source (that is, if the shortest path tree exists or not).

```
1:  procedure BELLMAN-FORD(G, w, s)
2:      INITIALIZE-SINGLE-SOURCE(G, s)
3:      for i = 1 to |G.V| − 1 do
4:          for each edge (u, v) ∈ G.E do
5:              RELAX(u, v, w)
6:          end for
7:      end for
8:      for each edge (u, v) ∈ G.E do
9:          if v.d > u.d + w(u, v) then
10:             return FALSE
11:         end if
12:     end for
13:     return TRUE
14: end procedure
```

The running time for this algorithm is $O(VE)$. The initialization takes $\Theta(V)$ time, the nested **for** loops in line 3 execute RELAX $(|V| - 1)|E|$ times. The loop in line 8 takes $O(E)$ time.

Next we prove the correctness of the algorithm.

Lemma 4.4.2[Triangle inequality]

Let G be a weighted directed graph with source s. Then for all edges $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

Proof. The proof is simple and omitted. □

Lemma 4.4.3[Upper-bound property]

Let G be a weighted directed graph with source s. Suppose that G is initialized by INITIALIZE-SINGLE-SOURCE(G, s). Then $v.d \geq \delta(s, v)$ for all $v \in V$. Moreover, once $v.d$ achieves its lower bound $\delta(s, v)$, it never changes.

Proof. We prove the invariant $v.d \geq \delta(s, v)$ by induction. For the basis, $v.d = \infty$ after initialization for all $v \in V - \{s\}$, so $v.d \geq \delta(s, v)$, and $s.d = 0 \geq \delta(s, s)$ (note that $\delta(s, s) = -\infty$ if s is on a negative cycle).

For the inductive step, consider the relaxation of an edge (u, v). By induction hypothesis, $x.d \geq \delta(s, x)$ for all $x \in V$ prior to the relaxation. The only d value that may change is $v.d$. If it changes, we have

$$\begin{aligned} v.d &= u.d + w(u, v) \\ &\geq \delta(s, u) + w(u, v) \quad \text{(by the inductive hypothesis)} \\ &\geq \delta(s, v) \quad \text{(by the triangle inequality).} \end{aligned}$$

We have just shown that $v.d \geq \delta(s, v)$, and it cannot increase because relaxation steps do not increase d values. □

Lemma 4.4.4[Convergence property]

Let G be a weighted directed graph with source s. Let $s \rightsquigarrow u \rightarrow v$ be a shortest path in G for some vertices $u, v \in V$. Suppose that G is initialized by INITIALIZE-SINGLE-SOURCE(G, s) and then a sequence of relaxation steps that includes the call RELAX(u, v, w) is executed on the edges of G. If $u.d = \delta(s, u)$ at any time prior to the call, then $v.d = \delta(s, v)$ at all times after the call.

Proof. If, just prior to relaxing edge (u, v), we have $v.d > u.d + w(u, v)$, then $v.d = u.d + w(u, v)$ afterward, we have $v.d \leq u.d + w(u, v)$. Otherwise, $v.d \leq u.d + w(u, v)$ and $v.d$ and $u.d$ will be unchanged. By the upper-bound property, if $u.d = \delta(s, u)$ at some point prior to relaxing edge (u, v), then this equality holds thereafter. In particular, after relaxing edge (u, v), we have

$$\begin{aligned} v.d &\leq u.d + w(u, v) \\ &= \delta(s, u) + w(u, v) \\ &= \delta(s, v) \quad \text{(by Lemma 4.4.1).} \end{aligned}$$

However, by the upper-bound proper, $v.d \geq \delta(s, v)$. Therefore $v.d = \delta(s, v)$. □

Lemma 4.4.5[Path-relaxation property]

Let G be a weighted directed graph with source s. Consider any shortest path $p = \langle v_0, v_1, \ldots, v_k \rangle$ from $s = v_0$ to v_k. If G is initialized by INITIALIZE-SINGLE-SOURCE(G, s) and then a sequence of relaxation steps occurs that includes, in order, relaxing the edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$ after these relaxations and at all times afterward.

Proof. We show by induction that after the ith edge of path p is relaxed, we have $v_i.d = \delta(s, v_i)$. For the basis, $i = 0$, and before any edges of p have been relaxed, we have $v_0.d = 0 = \delta(s, s)$. By the upper-bound property, the value of $s.d$ never changes after initialization.

For inductive step, we assume that $v_{i-1}.d = \delta(s, v_{i-1})$. By the convergence property, after relaxing this edge, we have $v_i.d = \delta(s, v_i)$, and this equality in maintained at all times thereafter. □

Lemma 4.4.6 Let G be a weighted directed graph with source s, and assume that G contains no negative-weight cycle that are reachable from s. Then after execute BELLMAN-FORD algorithm, $v.d = \delta(s, v)$ for all vertices v that are reachable from s.

Proof. Consider any vertex v that is reachable from s, and let $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = s$ and $v_k = v$, be any shortest path from s to v. Because shortest path is simple, p has at most $|V| - 1$ edges, so $k \leq |V| - 1$. Each of the $|V| - 1$ iterations relaxed all $|E|$ edges. Among the edges relaxed in ith iteration, for $i = 1, 2, \ldots, k$, is (v_{i-1}, v_i). By the path-relaxation property,
$v.d = v_k.d = \delta(s, v_k) = \delta(s, v)$ □

Theorem 4.4.7 [Correctness of the Bellman-Ford algorithm]

Let BELLMAN-FORD be run on a weighted, directed graph $G = (V, E)$ with source s and weight function $w : E \to \mathbb{R}$. If G contains no negative-weight cycles that are reachable from s, then the algorithm returns TRUE, we have $v.d = \delta(s, v)$ for all vertices $v \in V$, and the predecessor subgraph G_π is a shortest-paths tree rooted at s. If G does contain a negative-weight cycle reachable from s, then the algorithm returns FALSE.

Proof. Suppose that graph G contains no negative-weight cycles that are reachable from the source s. We first prove the claim that at termination, $v.d = \delta(s, v)$ for all vertices $v \in V$. If vertex v is reachable from s, then Lemma 4.4.6 proves this claim. If v is not reachable from s, then $v.d = \infty = \delta(s, v)$ by upper-bound property. Thus, the claim is proven. Lemma 4.4.1, along with the claim, implies that G_π is a shortest-paths tree. Now we use the claim to show that Bellman-Ford returns TRUE. At termination, we have for all edges $(u, v) \in E$,

$$\begin{aligned} v.d &= \delta(s, v) \\ &\leq \delta(s, u) + w(u, v) \text{ (by the triangle inequality)} \\ &= u.d + w(u, v), \end{aligned}$$

and so none of the tests in line 7 causes Bellman-Ford to return FALSE. Therefore, it returns TRUE.

Now, suppose that graph G contains a negative-weight cycle that is reachable from the source s; let this cycle be $c = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = v_k$. Then,

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) \quad < \quad 0. \tag{3}$$

Assume for the purpose of contradiction that the Bellman-Ford algorithm returns TRUE. Thus, $v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$ for $i = 1, 2, \ldots, k$. Summing the inequalities around cycle c gives us

$$\begin{aligned} \sum_{i=1}^{k} v_j.d \quad &\leq \quad \sum_{i=1}^{k} (v_{i-1}.d + w(v_{i-1}, v_i)) \\ &= \quad \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i). \end{aligned}$$

Since $v_0 = v_k$, each vertex in c appears exactly once in each of the summations $\sum_{i=1}^{k} v_i.d$ and $\sum_{i=1}^{k} v_{i-1}.d$, so

$$\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d.$$

Moreover, $v_i.d$ is finite for $i = 1, 2, \ldots, k$. Thus,

$$0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i),$$

which contradicts inequality (3). We conclude that the Bellman-Ford algorithm returns TRUE if graph G contains no negative-weight cycles reachable from the source, and FALSE otherwise. □

Dijkstra's algorithm

When all the weights are nonnegative, we can use Dijkstra's algorithm, the running time of which is lower than the Bellman-Ford algorithm.

The algorithm maintains a set S of vertices whose final shortest path weights from the source s have already been determined. It uses a min-priority queue Q keyed by their d value.

```
1:  procedure DIJKSTRA(G, w, s)
2:      INITIALIZE-SINGLE-SOURCE(G, s)
3:      S = ∅
4:      Q = G.V
5:      while Q ≠ ∅ do
6:          u = EXTRACT-MIN(Q)
7:          S = S ∪ {u}
8:          for each vertex v ∈ G.adj[u] do
9:              RELAX(u, v, w)
10:         end for
11:     end while
12: end procedure
```

Theorem 4.4.8[Correctness of Dijkstra's algorithm]

Dijkstra's algorithm, run on a non-negative weighted directed graph G with a source s, terminates with $u.d = \delta(s, u)$ for all vertices $u \in G.V$.

Proof. We claim that at the start of each iteration of the **while** loop, $v.d = \delta(s, v)$ for each $v \in S$.

Initially, $S = \emptyset$, so the claim is true. Assume that the claim is not always true and let u be the first vertex for which $u.d \neq \delta(s, u)$ when it is added to S.

We must have $u \neq s$ because s is the first vertex added to S and $s.d = \delta(s, s) = 0$.

Because $u \neq s$, $S \neq \emptyset$ when u is added to S. There must be some path from s to u otherwise $u.d = \delta(s, u) = \infty$. So there is a shortest path p from s to u.

Prior to adding u to S, p connects $s \in S$ and $v \in V - S$. We consider the first vertex y along p such that $y \in V - S$.

Let $x \in S$ be y's predecessor along p. We can decompose path p into $s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$. Because the path $s \overset{p_1}{\rightsquigarrow} x \rightarrow y$ is the shortest path from s to y and $x.d = \delta(s, x)$, edge (x, y) was relaxed when x was added to S and $y.d = \delta(s, y)$ by convergence property. So we have

$$\begin{aligned} y.d &= \delta(s, y) \\ &\leq \delta(s, u) \\ &\leq u.d \end{aligned}$$

But both u and y were in $V - S$ when u was chosen in line 5, we have $u.d \leq y.d$. Therefore we have

$$y.d = \delta(s, y) = \delta(s, u) = u.d.$$

Therefore, our claim is always true. □

All pairs shortest paths

Now we consider the problem of finding shortest paths between all pairs of vertices in a graph. Suppose we are given a weighted, directed graph $G = (V, E)$ with a weight function $w : E \mapsto \mathbb{R}$ that maps edges to real-valued weights. We wish to find, for every pair of vertices $u, v \in V$, a shortest (least-weight) path from u to v, where the weight of a path is the sum of the weights of its constituent edges.

We typically want the output in tabular form: the entry in us row and vs column should be the weight of a shortest path from u to v.

We can run Dijkstra's algorithm or Bellman-ford algorithm for each of the vertices, but we want to find more efficient algorithms.

We will use an adjacency-matrix representation of a graph instead of adjacency-list representation. For convenience, we assume that the vertices are numbered $1, 2, \ldots, |V|$, and the matrix representation of the directed graph is $W = (w_{ij})$, where

$$w_{ij} = \begin{cases} 0 & \text{if } i = j, \\ \text{the weight of edge}(i,j) & \text{if } i \neq j \text{ and } (i,j) \in E, \\ \infty & \text{if } i \neq j \text{ and } (i,j) \notin E. \end{cases}$$

We allow negative-weight edges, but the input graph contains no negative-weight cycle.

A dynamic-programming method

Since when we compute all pairs shortest paths there will be a lot of repeated computations, we consider to use dynamic programming.

To do that, we first need to characterize the structure of an optimal solution.

Suppose that we represent the graph by an adjacency matrix $W = (w_{ij})$. Consider a shortest path p from vertex i to vertex j , and suppose that p contains at most m edges. Assuming that there are no negative-weight cycles, m is finite. If $i = j$, then p has weight 0 and no edges. If vertices i and j are distinct, then we decompose path p into $i \overset{p'}{\rightsquigarrow} k \to j$, where path p' now contains at most $m - 1$ edges. By Lemma 4.4.1, p' is a shortest path from i to k, and so $\delta(i, j) = \delta(i, k) + w_{kj}$.

Next we consider recursive solution to the problem. So we define $l_{ij}^{(m)}$ be the minimum weight of any path from vertex i to vertex j that contains at most m edges. When $m = 0$, we have

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j, \\ \infty & \text{if } i \neq j. \end{cases}$$

For $m \geq 1$, we compute $l_{ij}^{(m)}$ as the minimum of $l_{ij}^{(m-1)}$ (the weight of a shortest path from i to j consisting of at most $m-1$ edges) and the minimum weight of any path from i to j consisting of at most m edges, obtained by looking at all possible predecessors k of j. Thus, we recursively define

$$\begin{aligned} l_{ij}^{(m)} &= \min\left(l_{ij}^{(m-1)}, \min_{1\leq k\leq n}\{l_{ik}^{(m-1)} + w_{kj}\}\right) \\ &= \min_{1\leq k\leq n}\{l_{ik}^{(m-1)} + w_{kj}\}. \end{aligned} \tag{4}$$

The latter equality follows since when $k = j, w_{kj} = 0$.

For any pair of vertices i and j for which $\delta(i,j) < \infty$, there is a shortest path from vertex i to vertex j that is simple and contains $m \leq n-1$ edges. Therefore $\delta(i,j) = l_{ij}^{(m)} = l_{ij}^{(m+1)} = \cdots = l_{ij}^{n-1}$. In general, we have

$$\delta(i,j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \cdots .$$

Taking as our input the matrix $W = (w_{ij})$, we now compute a series of matrices $L^{(1)}, L^{(2)}, \ldots L^{(n-1)}$, where for $m = 1, 2, \ldots, n-1$, $L^{(m)} = (l_{ij}^{(m)})$. The final matrix $L^{(n-1)}$ contains the actual shortest-path weights. Observe that $l_{ij}^{(1)} = w_{ij}$ for all vertices $i, j \in V$, so $L^{(1)} = W$.

The heart of the algorithm is the following procedure, which, given matrices $L^{(m-1)}$ and W, returns the matrix $L^{(m)}$.

```
1: procedure EXTEND-SHORTEST-PATHS(L, W)
2:     n = L.rows
3:     let L' = (l_ij) be a new n × n matrix
4:     for i = 1 to n do
5:         for j = 1 to n do
6:             l'_ij = ∞
7:             for k = 1 to n do
8:                 l'_ij = min(l'_ij, l_ik + w_ki)
9:             end for
10:        end for
11:    end for
12:    return L'
13: end procedure
```

It is easy to see that the running time for this procedure is $\Theta(n^3)$.
We can use the following procedure to compute $L^{(n-1)}$

```
1: procedure SLOW-ALL-PAIRS-SHORTEST-PATHS(W)
2:     n = W.rows
3:     L^(1) = W
4:     for m = 2 to n − 1 do
5:         let L^(m) be a new n × n matrix
6:         L^(m) = EXTEND-SHORTEST-PATHS(L^(m−1), W)
7:     end for
8:     return L^(n−1)
9: end procedure
```

Since the running time for EXTEND-SHORTEST-PATHS is $\Theta(n^3)$, the above procedure is $\Theta(n^4)$.

To improve the algorithm, we can reconsider the recursive formula (4). Recall that $l_{ij}^{(k)} = l_{ij}^{(m)}$ for $k > m$ if there is a shortest path with m edges from vertex i to vertex j. We have

$$\begin{aligned}
l_{ij}^{(1)} &= w_{ij} \\
l_{ij}^{(2)} &= \min_{1 \le k \le n} \{l_{ik}^{(1)} + l_{kj}^{(1)}\} \\
l_{ij}^{(4)} &= \min_{1 \le k \le n} \{l_{ik}^{(2)} + l_{kj}^{(2)}\} \\
\cdots &\quad \cdots \\
l_{ij}^{(2m)} &= \min_{1 \le k \le n} \{l_{ik}^{(m)} + l_{kj}^{(m)}\}
\end{aligned}$$

```
1:  procedure FASTER-ALL-PAIRS-SHORTEST-PATHS(W)
2:      n = W.rows
3:      L^(1) = W
4:      m = 1
5:      while m < n − 1 do
6:          let L^(2m) be a new n × n matrix
7:          L^(2m) = EXTEND-SHORTEST-PATHS(L^(m), L^(m))
8:          m = 2m
9:      end while
10:     return L^(m)
11: end procedure
```

In the above procedure, the **while** loop runs $\lceil \lg(n-1) \rceil$ times. Since the running time for EXTEND-SHORTEST-PATHS is $\Theta(n^3)$, the running time for FASTER-ALL-PAIRS-SHORTEST-PATHS is $\Theta(n^3 \lg n)$.

The Floyd-Warshall algorithm

In the Floyd-Warshall algorithm, we characterize the structure of a shortest path differently from how we characterized it in previous section. The Floyd-Warshall algorithm considers the intermediate vertices of a shortest path, where an intermediate vertex of a simple path $p = \langle v_1, v_2, \ldots, v_l \rangle$ is any vertex of p other than v_1 or v_l, that is, any vertex in the set $\{v_2, \ldots, v_{l-1}\}$.

As before, we assume that the vertices of G are $V = \{1, 2, \ldots, n\}$. Let us consider a subset $\{1, 2, \ldots, k\}$ of vertices for some k. For any pair of vertices $i, j \in V$, consider all paths from i to j whose intermediate vertices are all drawn from $\{1, 2, \ldots, k\}$, and let p be a minimum-weight path from among them. (Path p is simple.) The Floyd-Warshall algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. The relationship depends on whether or not k is an intermediate vertex of path p.

- If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set $\{1, 2, \ldots, k-1\}$. Thus, a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$ is also a shortest path from i to j with all intermediate vertices in the set $\{1, 2, \ldots, k\}$.
- If k is an intermediate vertex of path p, then we decompose p into $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$, By Lemma 4.4.1, p_1 is a shortest path from i to k with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Similarly, p_2 is a shortest path from vertex k to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$.

Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set $\{1, 2, \ldots, k\}$. When $k = 0$, a path from vertex i to vertex j with no intermediate vertex numbered higher than 0 has no intermediate vertices at all. Such a path has at most one edge, and hence $d_{ij}^{(0)} = w_{ij}$. Following the above discussion, we define $d_{ij}^{(k)}$ recursively by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \end{cases} \tag{5}$$

Because for any path, all intermediate vertices are in the set $\{1, 2, \ldots, n\}$, the matrix $D^{(n)} = \left(d_{ij}^{(n)}\right)$ gives the final answer: $d_{ij}^{(n)} = \delta(i, j)$ for all $i, j \in V$.

```
1:  procedure FLOYD-WARSHALL(W)
2:      n = W.rows
3:      D^(0) = W
4:      for k = 1 to n do
5:          let D^(k) = (d_ij^(n)) be a n × n matrix
6:          for i = 1 to n do
7:              for j = 1 to n do
8:                  d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
9:              end for
10:         end for
11:     end for
12:     return D^(n)
13: end procedure
```

The running time of the Floyd-Warshall algorithm is determined by the triply nested for loops. Because each execution of line 8 takes $O(1)$ time, the algorithm runs in time $\Theta(n^3)$. As the previous dynamic program, the code is tight, with no elaborate data structures, and so the constant hidden in the Θ-notation is small. Thus, the Floyd-Warshall algorithm is quite practical for even moderate-sized input graphs.

Now we consider how to construct a shortest path. We need to define a predecessor matrix $\Pi = (\pi_{ij})$, where π_{ij} is NIL if either $i = j$ or there is no path from i to j, and otherwise π_{ij} is the predecessor of j on some shortest path from i.

To obtain Π, we compute a sequence of matrices $\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(n)}$, where $\Pi = \Pi^{(n)}$ and we define $\pi_{ij}^{(k)}$ as the predecessor of vertex j on a shortest path from j on a shortest path from vertex i with all intermediate vertices in the set $\{1, 2, \ldots, k\}$. Then we have

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty, \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty \end{cases}$$

For $k \geq 1$, if we take the path $i \rightsquigarrow k \rightsquigarrow j$, where $k \neq j$, , then the predecessor of j we choose is the same as the predecessor of j we chose on a shortest path from k with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Otherwise, we choose the same predecessor of j that we chose on a shortest path from i with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Formally, for $k \geq 1$,

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

For each vertex $i \in V$, define the predecessor subgraph of G for i as $G_{\pi,i} = (V_{\pi,i}, E_{\pi,i})$, where

$V_{\pi,i} = \{j \in V : \pi_{ij} \neq \text{NIL}\} \cup \{i\}$ and $E_{\pi,i} = \{(\pi_{ij}, j) : j \in V_{\pi,i} - \{i\}\}$.

If $G_{\pi,i}$ is a shortest-paths tree, then we can use the following procedure to print a shortest path from vertex i to vertex j.

```
1: procedure PRINT-ALL-PAIRS-PATH(Π, i, j)
2:     if i == j then
3:         print i
4:     else if π_ij == NIL then
5:         print "no path from i to j exists"
6:     else
7:         PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, π_ij)
8:         print j
9:     end if
10: end procedure
```

For the Π from the Floyd-Warshall algorithm, it can be proved that $G_{\pi,i}$ is a shortest path tree with root i.

Now we introduce the **transitive closure** of a directed graph $G = (E, V)$, which is a graph $G^* = (V, E^*)$, where

$E^* = \{(i, j) : \text{there is a path from vertex } i \text{ to vertex } j \text{ in } G\}$.

One way to compute the transitive closure of a graph in $\Theta(n^3)$ time is to assign a weight of 1 to each edge of E and run the Floyd-Warshall algorithm. If there is a path from vertex i to vertex j, we get $d_{ij} < n$. Otherwise, we get $d_{ij} = \infty$.

There is another, similar way to compute the transitive closure of G in $\Theta(n^3)$ time that can save time and space in practice. This method substitutes the logical operations $\vee$ (logical OR) and $\wedge$ (logical AND) for the arithmetic operations min and $+$ in the Floyd-Warshall algorithm.

For $i, j, k = 1, 2, \ldots, n$, we define $t_{ij}^{(k)}$ to be 1 if there exists a path in graph G from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k\}$, and 0 otherwise. We construct the transitive closure $G^* = (V, E^*)$ by putting edge (i, j) into E^* if and only if $t_{ij}^{(n)} = 1$. A recursive definition of $t_{ij}^{(k)}$, analogous to recurrence (5), is

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i, j) \notin E, \\ 1 & \text{if } i = j \text{ or } (i, j) \in E. \end{cases}$$

and for $k \geq 1$,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}).$$

We compute the matrices $T^{(k)} = (t_{ij}^{(k)})$ in order of increasing k.

```
1:  procedure TRANSITIVE-CLOSURE(G)
2:      n = |G.V|
3:      let T^(0) = (t_ij^(0)) be a new n × n matrix
4:      for i = 1 to n do
5:          for j = 1 to n do
6:              if i == j or (i, j) ∈ G.E then
7:                  t_ij^(0) = 1
8:              else
9:                  t_ij^(0) = 0
10:             end if
11:         end for
12:     end for
13:     for k = 1 to n do
14:         let T^(k) = (t_ij^(k)) be a new n × n matrix
15:         for i = 1 to n do
```

```
16:             for j = 1 to n do
17:                 t_ij^(k) = t_ij^(k-1) ∨ (t_ik^(k-1) ∧ t_kj^(k-1))
18:             end for
19:         end for
20:     end for
21:     return T^(n)
22: end procedure
```

The above procedure also runs in $\Theta(n^3)$ time. But on some computers, logical operations on single-bit values execute faster than arithmetic operations on integer words of data. Moreover, because the direct transitive-closure algorithm uses only boolean values rather than integer values, its space requirement is less than the Floyd-Warshall algorithms by a factor corresponding to the size of a word of computer storage.