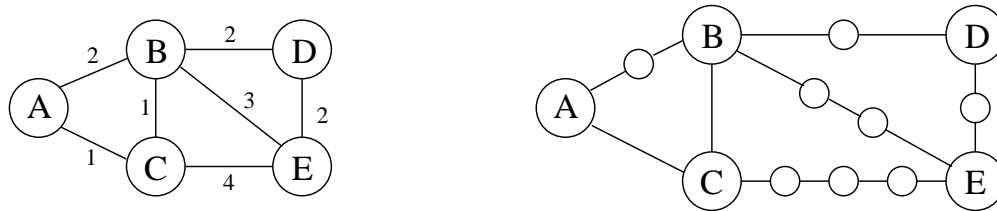


**Figure 4.6** Breaking edges into unit-length pieces.

## 4.4 Dijkstra's algorithm

### 4.4.1 An adaptation of breadth-first search

Breadth-first search finds shortest paths in any graph whose edges have unit length. Can we adapt it to a more general graph  $G = (V, E)$  whose edge lengths  $l_e$  are *positive integers*?

#### A more convenient graph

Here is a simple trick for converting  $G$  into something BFS can handle: break  $G$ 's long edges into unit-length pieces, by introducing “dummy” nodes. Figure 4.6 shows an example of this transformation. To construct the new graph  $G'$ ,

For any edge  $e = (u, v)$  of  $E$ , replace it by  $l_e$  edges of length 1, by adding  $l_e - 1$  dummy nodes between  $u$  and  $v$ .

Graph  $G'$  contains all the vertices  $V$  that interest us, and the distances between them are exactly the same as in  $G$ . Most importantly, the edges of  $G'$  all have unit length. Therefore, we can compute distances in  $G$  by running BFS on  $G'$ .

#### Alarm clocks

If efficiency were not an issue, we could stop here. But when  $G$  has very long edges, the  $G'$  it engenders is thickly populated with dummy nodes, and the BFS spends most of its time diligently computing distances to these nodes that we don't care about at all.

To see this more concretely, consider the graphs  $G$  and  $G'$  of Figure 4.7, and imagine that the BFS, started at node  $s$  of  $G'$ , advances by one unit of distance per minute. For the first 99 minutes it tediously progresses along  $S - A$  and  $S - B$ , an endless desert of dummy nodes. Is there some way we can snooze through these boring phases and have an alarm wake us up whenever something *interesting* is happening—specifically, whenever one of the real nodes (from the original graph  $G$ ) is reached?

We do this by setting two alarms at the outset, one for node  $A$ , set to go off at time  $T = 100$ , and one for  $B$ , at time  $T = 200$ . These are *estimated times of arrival*, based upon the edges currently being traversed. We doze off and awake at  $T = 100$  to find  $A$  has been discovered. At

this point, the estimated time of arrival for  $B$  is adjusted to  $T = 150$  and we change its alarm accordingly.

More generally, at any given moment the breadth-first search is advancing along certain edges of  $G$ , and there is an alarm for every endpoint node toward which it is moving, set to go off at the estimated time of arrival at that node. Some of these might be overestimates because BFS may later find shortcuts, as a result of future arrivals elsewhere. In the preceding example, a quicker route to  $B$  was revealed upon arrival at  $A$ . However, *nothing interesting can possibly happen before an alarm goes off*. The sounding of the next alarm must therefore signal the arrival of the wavefront to a real node  $u \in V$  by BFS. At that point, BFS might also start advancing along some new edges out of  $u$ , and alarms need to be set for their endpoints.

The following “alarm clock algorithm” faithfully simulates the execution of BFS on  $G'$ .

- Set an alarm clock for node  $s$  at time 0.
- Repeat until there are no more alarms:

Say the next alarm goes off at time  $T$ , for node  $u$ . Then:

- The distance from  $s$  to  $u$  is  $T$ .
- For each neighbor  $v$  of  $u$  in  $G$ :
  - \* If there is no alarm yet for  $v$ , set one for time  $T + l(u, v)$ .
  - \* If  $v$ 's alarm is set for later than  $T + l(u, v)$ , then reset it to this earlier time.

**Dijkstra’s algorithm.** The alarm clock algorithm computes distances in any graph with positive integral edge lengths. It is almost ready for use, except that we need to somehow implement the system of alarms. The right data structure for this job is a *priority queue* (usually implemented via a *heap*), which maintains a set of elements (nodes) with associated numeric key values (alarm times) and supports the following operations:

*Insert.* Add a new element to the set.

*Decrease-key.* Accommodate the decrease in key value of a particular element.<sup>1</sup>

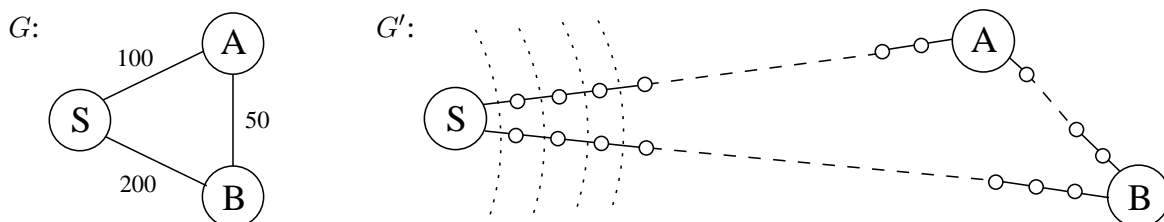
---

<sup>1</sup>The name *decrease-key* is standard but is a little misleading: the priority queue typically does not itself change key values. What this procedure really does is to notify the queue that a certain key value has been decreased.

---

**Figure 4.7** BFS on  $G'$  is mostly uneventful. The dotted lines show some early “wavefronts.”

---



*Delete-min.* Return the element with the smallest key, and remove it from the set.

*Make-queue.* Build a priority queue out of the given elements, with the given key values. (In many implementations, this is significantly faster than inserting the elements one by one.)

The first two let us set alarms, and the third tells us which alarm is next to go off. Putting this all together, we get Dijkstra's algorithm (Figure 4.8).

In the code,  $\text{dist}(u)$  refers to the current alarm clock setting for node  $u$ . A value of  $\infty$  means the alarm hasn't so far been set. There is also a special array, `prev`, that holds one crucial piece of information for each node  $u$ : the identity of the node immediately before it on the shortest path from  $s$  to  $u$ . By following these back-pointers, we can easily reconstruct shortest paths, and so this array is a compact summary of all the paths found. A full example of the algorithm's operation, along with the final shortest-path tree, is shown in Figure 4.9.

In summary, we can think of Dijkstra's algorithm as just BFS, except it uses a priority queue instead of a regular queue, so as to prioritize nodes in a way that takes edge lengths into account. This viewpoint gives a concrete appreciation of how and why the algorithm works, but there is a more direct, more abstract derivation that doesn't depend upon BFS at all. We now start from scratch with this complementary interpretation.

---

**Figure 4.8** Dijkstra's shortest-path algorithm.

---

`procedure dijkstra( $G, l, s$ )`

Input:     Graph  $G = (V, E)$ , directed or undirected;  
           positive edge lengths  $\{l_e : e \in E\}$ ; vertex  $s \in V$

Output:    For all vertices  $u$  reachable from  $s$ ,  $\text{dist}(u)$  is set  
           to the distance from  $s$  to  $u$ .

for all  $u \in V$ :  
     $\text{dist}(u) = \infty$   
     $\text{prev}(u) = \text{nil}$   
 $\text{dist}(s) = 0$

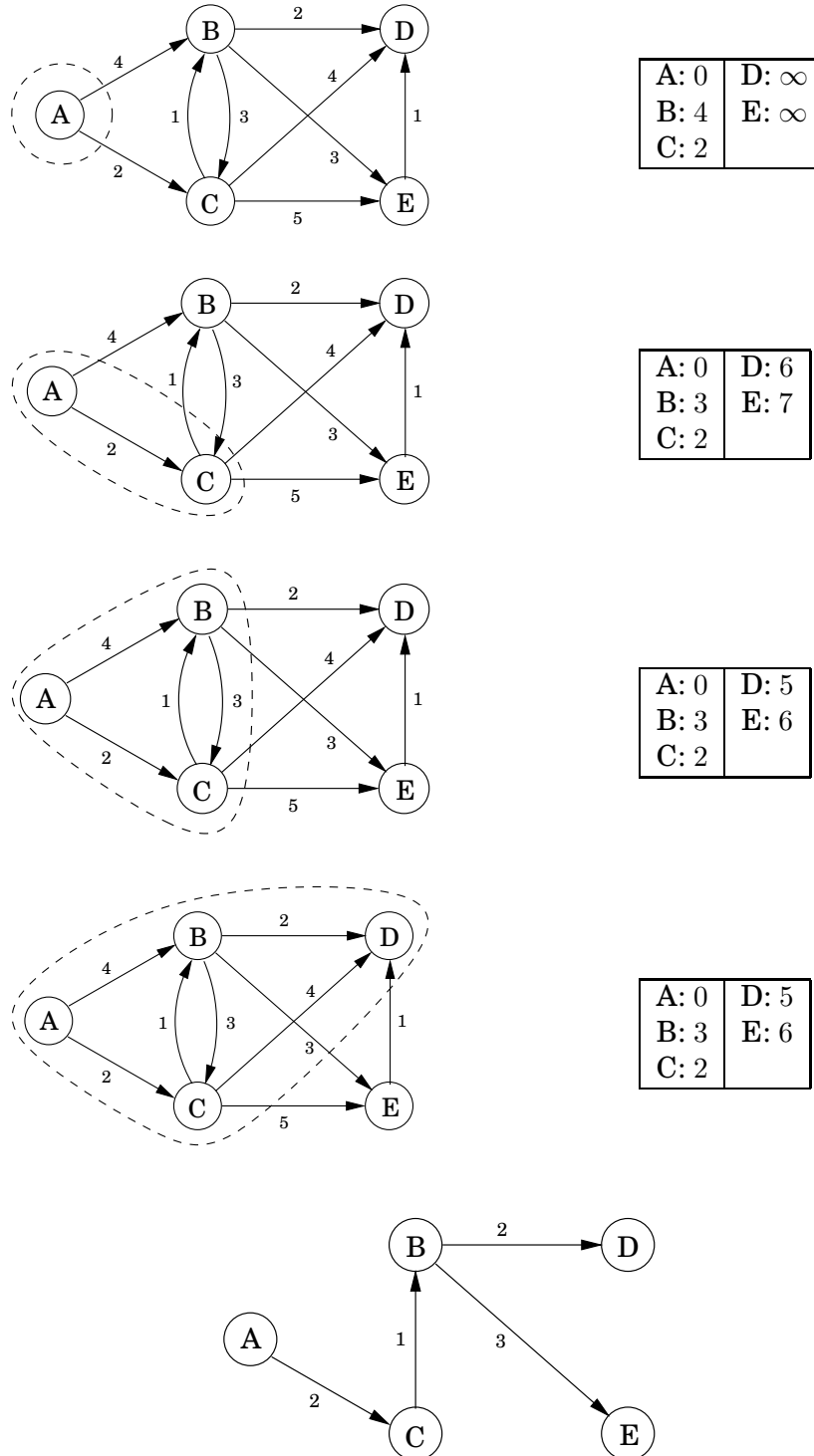
$H = \text{makequeue}(V)$  (using  $\text{dist}$ -values as keys)

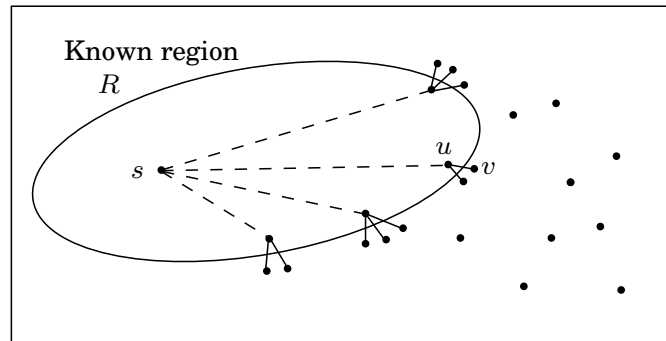
while  $H$  is not empty:

$u = \text{deletemin}(H)$   
    for all edges  $(u, v) \in E$ :  
        if  $\text{dist}(v) > \text{dist}(u) + l(u, v)$ :  
             $\text{dist}(v) = \text{dist}(u) + l(u, v)$   
             $\text{prev}(v) = u$   
             $\text{decreasekey}(H, v)$

---

**Figure 4.9** A complete run of Dijkstra's algorithm, with node *A* as the starting point. Also shown are the associated *dist* values and the final shortest-path tree.

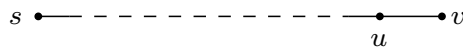


**Figure 4.10** Single-edge extensions of known shortest paths.

#### 4.4.2 An alternative derivation

Here's a plan for computing shortest paths: expand outward from the starting point  $s$ , steadily growing the region of the graph to which distances and shortest paths are known. This growth should be orderly, first incorporating the closest nodes and then moving on to those further away. More precisely, when the “known region” is some subset of vertices  $R$  that includes  $s$ , the next addition to it should be *the node outside  $R$  that is closest to  $s$* . Let us call this node  $v$ ; the question is: how do we identify it?

To answer, consider  $u$ , the node just before  $v$  in the shortest path from  $s$  to  $v$ :



Since we are assuming that all edge lengths are positive,  $u$  must be closer to  $s$  than  $v$  is. This means that  $u$  is in  $R$ —otherwise it would contradict  $v$ 's status as the closest node to  $s$  outside  $R$ . So, the shortest path from  $s$  to  $v$  is simply *a known shortest path extended by a single edge*.

But there will typically be many single-edge extensions of the currently known shortest paths (Figure 4.10); which of these identifies  $v$ ? The answer is, *the shortest of these extended paths*. Because, if an even shorter single-edge-extended path existed, this would once more contradict  $v$ 's status as the node outside  $R$  closest to  $s$ . So, it's easy to find  $v$ : it is the node outside  $R$  for which the smallest value of  $\text{distance}(s, u) + l(u, v)$  is attained, as  $u$  ranges over  $R$ . In other words, *try all single-edge extensions of the currently known shortest paths, find the shortest such extended path, and proclaim its endpoint to be the next node of  $R$* .

We now have an algorithm for growing  $R$  by looking at extensions of the current set of shortest paths. Some extra efficiency comes from noticing that on any given iteration, the only *new* extensions are those involving the node most recently added to region  $R$ . All other extensions will have been assessed previously and do not need to be recomputed. In the following pseudocode,  $\text{dist}(v)$  is the length of the currently shortest single-edge-extended path leading to  $v$ ; it is  $\infty$  for nodes not adjacent to  $R$ .

```

Initialize dist(s) to 0, other dist( $\cdot$ ) values to  $\infty$ 
R = { } (the ``known region'')
while R  $\neq$  V:
    Pick the node v  $\notin$  R with smallest dist( $\cdot$ )
    Add v to R
    for all edges (v, z)  $\in$  E:
        if dist(z) > dist(v) + l(v, z):
            dist(z) = dist(v) + l(v, z)

```

Incorporating priority queue operations gives us back Dijkstra's algorithm (Figure 4.8).

To justify this algorithm formally, we would use a proof by induction, as with breadth-first search. Here's an appropriate inductive hypothesis.

At the end of each iteration of the while loop, the following conditions hold: (1) there is a value  $d$  such that all nodes in  $R$  are at distance  $\leq d$  from  $s$  and all nodes outside  $R$  are at distance  $\geq d$  from  $s$ , and (2) for every node  $u$ , the value  $\text{dist}(u)$  is the length of the shortest path from  $s$  to  $u$  whose intermediate nodes are constrained to be in  $R$  (if no such path exists, the value is  $\infty$ ).

The base case is straightforward (with  $d = 0$ ), and the details of the inductive step can be filled in from the preceding discussion.

### 4.4.3 Running time

At the level of abstraction of Figure 4.8, Dijkstra's algorithm is structurally identical to breadth-first search. However, it is slower because the priority queue primitives are computationally more demanding than the constant-time `eject`'s and `inject`'s of BFS. Since `makequeue` takes at most as long as  $|V|$  `insert` operations, we get a total of  $|V|$  `deletemin` and  $|V| + |E|$  `insert/decreasekey` operations. The time needed for these varies by implementation; for instance, a binary heap gives an overall running time of  $O((|V| + |E|) \log |V|)$ .

**Which heap is best?**

The running time of Dijkstra's algorithm depends heavily on the priority queue implementation used. Here are the typical choices.

Implementation	deletemin	insert/ decreasekey	$ V  \times \text{deletemin} +$ $( V  +  E ) \times \text{insert}$
Array	$O( V )$	$O(1)$	$O( V ^2)$
Binary heap	$O(\log  V )$	$O(\log  V )$	$O(( V  +  E ) \log  V )$
$d$ -ary heap	$O(\frac{d \log  V }{\log d})$	$O(\frac{\log  V }{\log d})$	$O(( V  \cdot d +  E ) \frac{\log  V }{\log d})$
Fibonacci heap	$O(\log  V )$	$O(1)$ (amortized)	$O( V  \log  V  +  E )$

So for instance, even a naive array implementation gives a respectable time complexity of  $O(|V|^2)$ , whereas with a binary heap we get  $O((|V| + |E|) \log |V|)$ . Which is preferable?

This depends on whether the graph is *sparse* (has few edges) or *dense* (has lots of them). For all graphs,  $|E|$  is less than  $|V|^2$ . If it is  $\Omega(|V|^2)$ , then clearly the array implementation is the faster. On the other hand, the binary heap becomes preferable as soon as  $|E|$  dips below  $|V|^2 / \log |V|$ .

The  $d$ -ary heap is a generalization of the binary heap (which corresponds to  $d = 2$ ) and leads to a running time that is a function of  $d$ . The optimal choice is  $d \approx |E|/|V|$ ; in other words, to optimize we must set the degree of the heap to be equal to the *average degree* of the graph. This works well for both sparse and dense graphs. For very sparse graphs, in which  $|E| = O(|V|)$ , the running time is  $O(|V| \log |V|)$ , as good as with a binary heap. For dense graphs,  $|E| = \Omega(|V|^2)$  and the running time is  $O(|V|^2)$ , as good as with a linked list. Finally, for graphs with intermediate density  $|E| = |V|^{1+\delta}$ , the running time is  $O(|E|)$ , linear!

The last line in the table gives running times using a sophisticated data structure called a *Fibonacci heap*. Although its efficiency is impressive, this data structure requires considerably more work to implement than the others, and this tends to dampen its appeal in practice. We will say little about it except to mention a curious feature of its time bounds. Its *insert* operations take varying amounts of time but are guaranteed to *average*  $O(1)$  over the course of the algorithm. In such situations (one of which we shall encounter in Chapter 5) we say that the *amortized* cost of heap *insert*'s is  $O(1)$ .