Paths in graphs[1]

1 Distances

Depth-first search readily identifies all the vertices of a graph which can be reached from a designated starting point. It also finds explicit paths to these vertices, summarized in its search tree (Figure 1.1). However, these paths might not be the most economical ones possible. In the figure, vertex C is reachable from S by traversing just one edge, while the DFS tree shows a path of length three. This chapter is about algorithms for finding *shortest paths* in graphs.

Path lengths allow us to talk quantitatively about the extent to which different vertices of a graph are separated from each other:

The *distance* between two nodes is the length of the shortest path between them.

To get a concrete feel for this notion, consider a physical realization of a graph which has a ball for each vertex and a piece of string for each edge. If you lift the ball for vertex s high enough, the other balls which get pulled up along with it are precisely the vertices reachable from s. And to find their distances from s, you need only measure how far below s they hang.

In Figure 1.2 for example, vertex B is at distance two from S, and there are two shortest paths to it. When S is held up, the strings along each of these paths become taut. On the other hand, edge (D, E) plays no role in any shortest path and therefore remains slack.

2 Breadth-first search

In Figure 1.2, the lifting of s partitions the graph into layers: s itself; the nodes at distance one from it; the nodes at distance two from it; and so on. A convenient way to compute distances from s to the other vertices is to proceed layer by layer. Once we have picked out the nodes at distance $0, 1, 2, \ldots, d$, the ones at $d + 1$ are easily determined: they are precisely the as-yet-unseen nodes which are adjacent to the layer at distance d. This suggests an iterative algorithm in which two layers are active at any given time: some layer d which has been fully identified, and $d + 1$, which is being discovered by scanning the neighbors of layer d.

[1]Copyright ©2004 S. Dasgupta, C.H. Papadimitriou, and U.V. Vazirani.

Figure 1.1 (a) A simple graph and (b) its depth-first search tree.

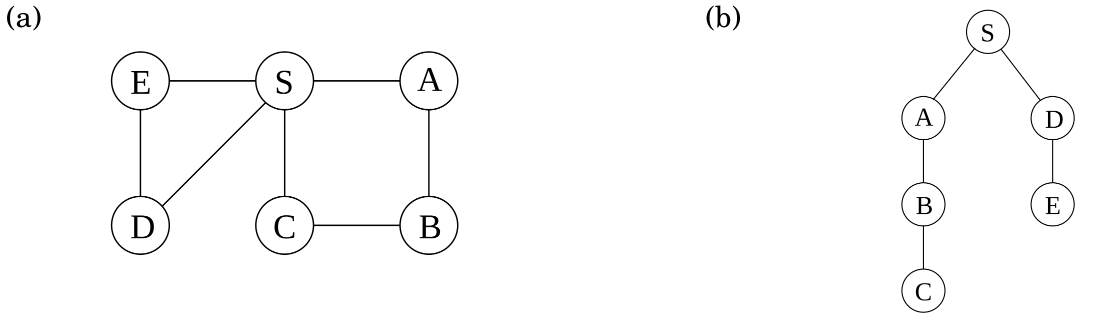


Figure 1.2 A physical model of a graph

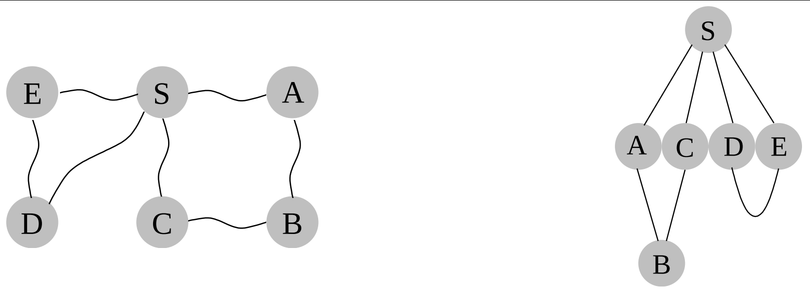


Figure 2.1 Breadth-first search.

```
procedure bfs(G, s)
Input:    Graph $G = (V, E)$, directed or undirected; vertex $s \in V$
Output:   For all vertices $u$ reachable from $s$, dist($u$) is set
          to the distance from $s$ to $u$.

for all $u \in V$:
   dist($u$) $= \infty$

dist($s$) $= 0$
$Q = [s]$ (queue containing just $s$)
while $Q$ is not empty:
   $u =$ eject($Q$)
   for all edges $(u, v) \in E$:
      if dist($v$) $= \infty$:
         inject($Q, v$)
         dist($v$) $=$ dist($u$) $+ 1$
```

Breadth-first search (Figure 2.1) directly implements this simple reasoning. Initially the queue Q consists only of s, the one node at distance zero. And for each subsequent distance $d = 1, 2, 3, \ldots$, there is a point in time at which Q contains all the nodes at distance d and nothing else. As these nodes are processed (ejected off the front of the queue), their as-yet-unseen neighbors are injected into the end of the queue.

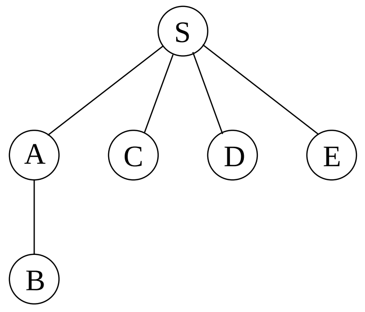
Let's try out this algorithm on our earlier example (Figure 1.1), to confirm that it does the right thing. If S is the starting point and the nodes are ordered alphabetically, they get visited in the sequence shown in Figure 2.2. The breadth-first search tree, on the right, contains the edges through which each node is initially discovered. Unlike the DFS tree we saw earlier, it has the property that all of its paths from A are the shortest possible. It is therefore a *shortest-path tree*.

Establishing correctness. We have developed the basic intuition behind breadth-first search. In order to check that the algorithm works correctly, we need to make sure that it faithfully executes this intuition. What we expect, precisely, is that:

For each $d = 0, 1, 2, \ldots$, there is a moment at which (1) all nodes at distance $\leq d$

Figure 2.2 The result of breadth-first search on the graph of Figure 1.1.

Order of visitation	Queue contents after processing node
	$[S]$
S	$[A\ C\ D\ E]$
A	$[C\ D\ E\ B]$
C	$[D\ E\ B]$
D	$[E\ B]$
E	$[B]$
B	$[\,]$



from s have their distances correctly set; (2) all other nodes have their distances set to ∞; and (3) the queue contains exactly the nodes at distance d.

This has been phrased with an inductive argument in mind. We have already discussed both the base case and the inductive step. Can you fill in the details?

The overall running time of this algorithm is linear, $O(|V| + |E|)$, for the exactly same reasons as depth-first search. Each vertex is put on the queue exactly once, when it is first encountered, so there are $2\,|V|$ queue operations. The rest of the work is done in algorithm's innermost loop. This loop looks at each edge once (in directed graphs) or twice (in undirected graphs), and therefore takes $O(|E|)$ time.

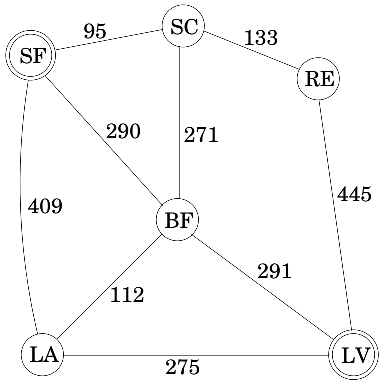
Now that we have both BFS and DFS before us: how do their exploration styles compare? Depth-first search makes deep incursions into a graph, retreating only when it runs out of new nodes to visit. This strategy gives it the wonderful, subtle, and extremely useful properties we saw in the previous chapter. But it also means that DFS can end up taking a long and convoluted route to a vertex which is actually very close by, as in Figure 1.1. Breadth-first search makes sure to visit vertices in increasing order of their distance from the starting point. This is a broader, shallower search, rather like the propagation of a wave upon water. And it is achieved using almost exactly the same code as DFS – but with a queue in place of a stack.

Also notice one stylistic difference from DFS: since we are only interested in distances from s, we do not restart the search in other connected components. Nodes not reachable from s are simply ignored.

3 Lengths on edges

Breadth-first search treats all edges as having the same length. This is rarely true in applications where shortest paths are to be found. For instance, suppose you are driving from San Francisco to Las Vegas, and want to find the quickest route. Figure 3.1 shows the major highways you might conceivably use. Picking the right combination of them is a shortest-path problem in which the length of each edge (each stretch of highway) is important. For the remainder of this chapter, we will deal with this more general scenario, annotating every edge

Figure 3.1 Edge lengths often matter.



$e \in E$ with a length l_e. If $e = (u, v)$, we will sometimes also write $l(u, v)$.

These l_e's do not have to correspond to physical lengths. They could denote time (driving time between cities) or money (cost of taking a bus), or any other quantity that we would like to conserve. In fact, there are cases in which we need to use negative lengths, but we will briefly overlook this particular complication.

4 Dijkstra's algorithm

4.1 An adaptation of breadth-first search

Breadth-first search finds shortest paths in graphs 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 4.1 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 one, by adding $l_e - 1$ dummy nodes between u and v.

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'.

Figure 4.1 Breaking edges into unit-length pieces.

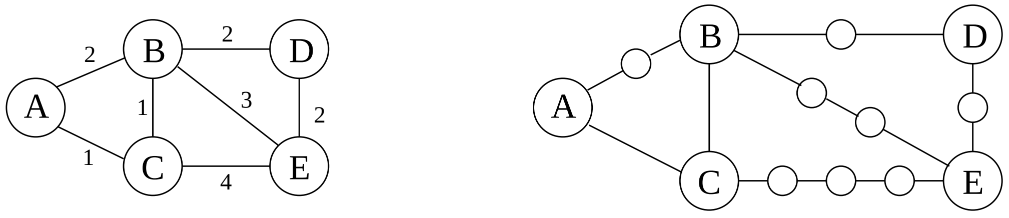
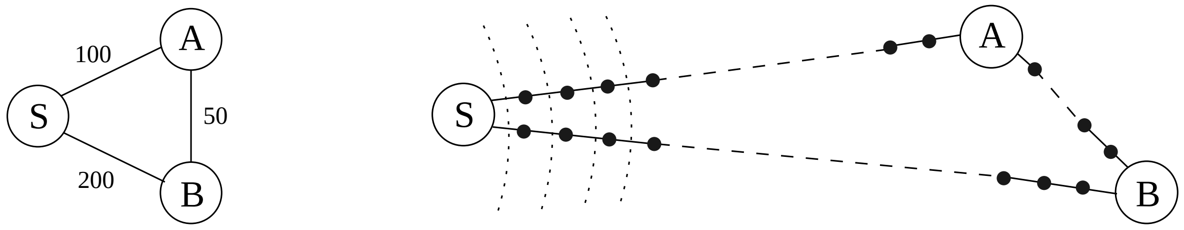


Figure 4.2 BFS on G' is mostly uneventful. The dotted lines show some early "wavefronts".



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 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 4.2, and imagine that 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 BFS is advancing along certain edges of G, and there is an alarm for every endpoint-node towards 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 interesting events. In the example above, 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 coincide with the next discovery of 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 "alarm clock algorithm" is designed to faithfully simulate the execution of BFS on G'. We will defer a rigorous justification till the next section, when we revisit the same procedure from a different viewpoint. Here, meanwhile, is the full pseudocode.

- 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.

Figure 4.3 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 $s$, dist($u$) is set
          to the distance from $s$ to $u$.

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

$H =$ makeheap($V$)  (using dist-values as keys)
while $H$ is not empty:
   $u =$ deletemin($H$)
   for all edges $(u, v) \in E$:
      if dist($v$) $>$ dist($u$) $+ l(u, v)$:
         dist($v$) $=$ dist($u$) $+ l(u, v)$
         prev($v$) $= u$
         decreasekey($H, v$)
```

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 *heap* (sometimes called a *priority queue*), 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.[2]

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

Make-heap. Build a heap out of the given elements.

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 4.3).

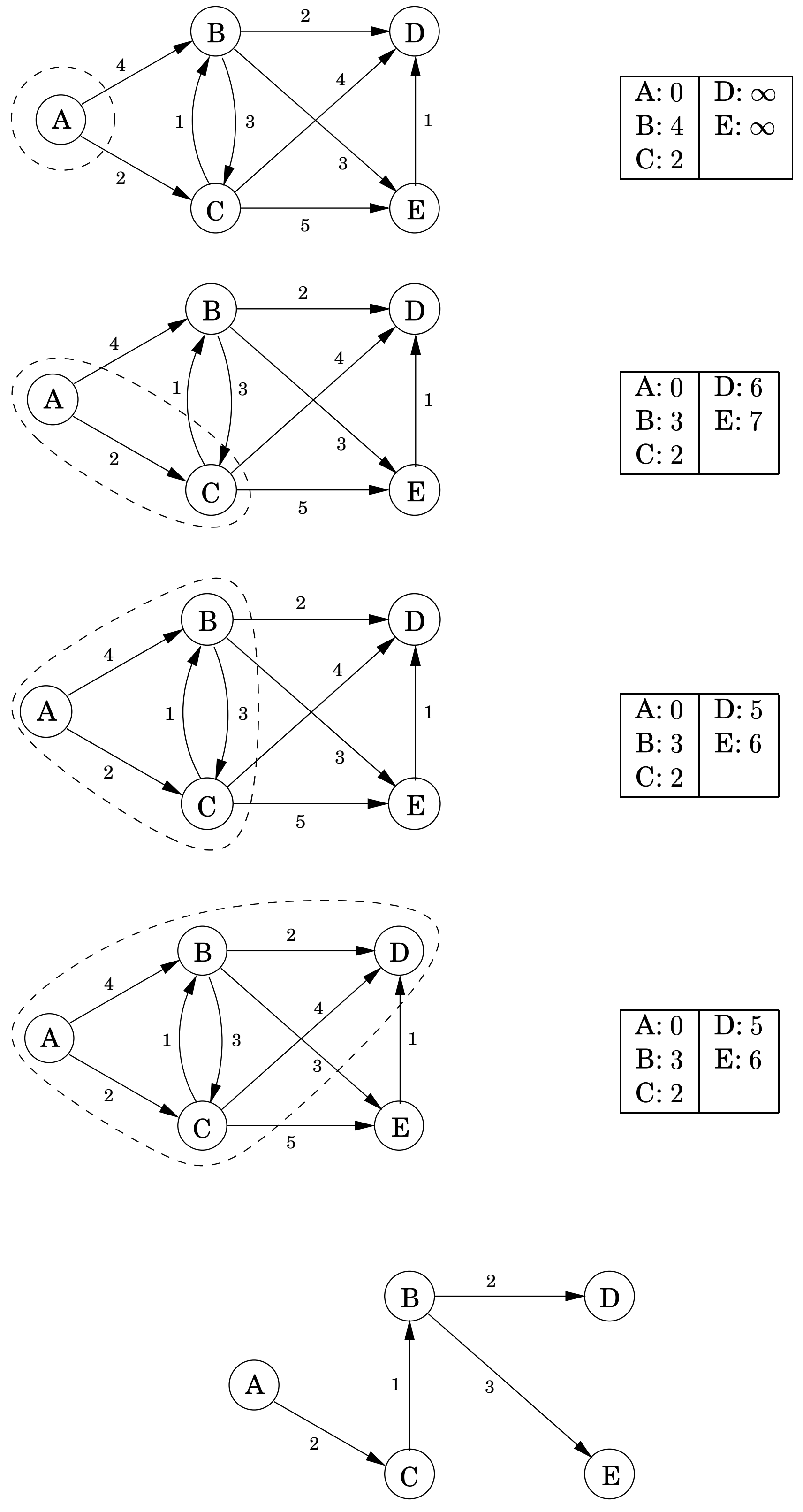
In the code, dist(u) refers to the current alarm clock setting for node u. A value of ∞ means the alarm hasn't so far been set. There is also a special array, prev, which holds one crucial piece of information for each node u: the identity of the node immediately before it on the shortest path from s 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 4.4.

Our particular view of Dijkstra's algorithm, that it is a simulation of breadth-first search on a suitably modified graph, gives a concrete appreciation of how and why it works. But

[2]The name *decrease-key* is standard but is a little misleading: the heap does not itself change key values. What this procedure really does is to notify the heap that a certain key value has been decreased.

there is a more direct, more abstract derivation, which doesn't depend upon BFS at all. We now start from scratch with this complementary interpretation.

Figure 4.4 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.



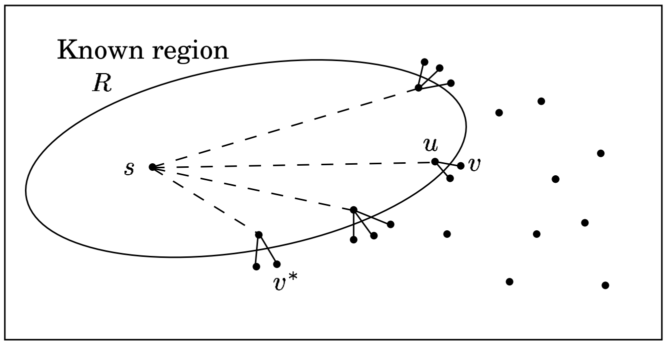
A: 0	D: ∞
B: 4	E: ∞
C: 2	

A: 0	D: 6
B: 3	E: 7
C: 2	

A: 0	D: 5
B: 3	E: 6
C: 2	

A: 0	D: 5
B: 3	E: 6
C: 2	

Figure 4.5 Single-edge extensions of known shortest paths.



4.2 An alternative derivation

Here's a plan for computing shortest paths: expand outwards from the starting point s, steadily growing the region of the graph to which distances and 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, the next addition to it should be *the node outside R which is closest to s*. How do we identify this node?

Property. If all edge lengths are positive, then

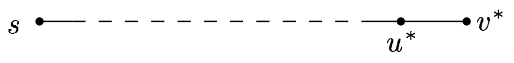
$$\min_{v \notin R} \text{distance}(s, v) \;=\; \min_{u \in R} \; \min_{v \notin R : (u,v) \in E} \text{distance}(s, u) + l(u, v). \qquad (1)$$

The left-hand side of this equation refers to the node we want. Call it v^*. The right-hand side tells us how to find it: by looking at *single-edge extensions* of known shortest paths,

"shortest path to a node in R, followed by an edge leading out of R" (Figure 4.5),

and picking the shortest one.

These single-edge extensions are all legitimate paths. To prove the property, we only need to check that the shortest path from s to v^* is amongst them. Let's take a closer look at this path, particularly at its second-last node, which we'll call u^*:



Since this is a shortest path, there is no scope for further shrinking any part of it. So the initial portion must be the shortest path from s to u^*.

Is u^* in R? Well, we've defined v^* to be the closest node to s outside R. Since u^* is even closer to s, by a distance of $l(u^*, v^*) > 0$, it must be in R. Therefore the shortest path to v^* has exactly the right form: it consists of the shortest path to $u^* \in R$, plus the single edge (u^*, v^*).

We now have an algorithm. 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, `dist`(v) is the length of the shortest current extension leading to v, as in the right-hand side of (1). It is ∞ for nodes not adjacent to R.

Figure 4.6 The complexity of Dijkstra's algorithm, under different heap implementations.

Heap implementation	`deletemin`	`insert/` `decreasekey`	$\lvert V\rvert \times$ `deletemin` + $(\lvert V\rvert + \lvert E\rvert) \times$ `insert`
Linked list	$O(\lvert V\rvert)$	$O(1)$	$O(\lvert V\rvert^2)$
Binary heap	$O(\log \lvert V\rvert)$	$O(\log \lvert V\rvert)$	$O(\lvert E\rvert \log \lvert V\rvert)$
d-ary heap	$O(\frac{d \log \lvert V\rvert}{\log d})$	$O(\frac{\log \lvert V\rvert}{\log d})$	$O((\lvert V\rvert \cdot d + \lvert E\rvert)\frac{\log \lvert V\rvert}{\log d})$
Fibonacci heap	$O(\log \lvert V\rvert)$	$O(1)$ (amortized)	$O(\lvert V\rvert \log \lvert V\rvert + \lvert E\rvert)$

```
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($\cdot$)
   for all edges $(v,z) \in E$:
      if dist($z$) $>$ dist($v$) $+ l(v,z)$:
         dist($z$) $=$ dist($v$) $+ l(v,z)$
   Add $v$ to $R$
```

Incorporating heap operations gives us back Dijkstra's algorithm (Figure 4.3).

4.3 Running time

At the level of abstraction of Figure 4.3, Dijkstra's algorithm is structurally identical to breadth-first search. However, it is slower because the heap accesses are computationally more demanding than the constant-time `eject`'s and `inject`'s of BFS. Treating `makeheap` as a sequence of $|V|$ `insert` operations, we get a total of $|V|$ `deletemin`, $|V|$ `insert`, and $|E|$ `decreasekey` operations. The time needed for these varies by heap implementation (Figure 4.6), as we will shortly discuss. In all our implementations, however, `insert` and `decreasekey` have the same complexity, so we will treat them as one.

Let's try to understand the various possible running times. Even a naive implementation – a linked list containing elements in no particular order – gives a respectable time complexity of $O(|V|^2)$. With a binary heap, we get $O(|E| \log |V|)$. 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 linked list 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 which is a function of d. The optimal choice is $d \approx |E|/|V|$, the *average degree* of the graph; to avoid having to worry about the case $|E| \approx 0$, we will take $d = |E|/|V| + 2$. 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 Figure 4.6 gives running times using the sophisticated *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.

5 Heap implementations

5.1 Linked list

The simplest implementation of a heap is as an unordered linked list. An `insert` is achieved quickly, adding the new element to the front of the list in $O(1)$ time. A `decreasekey` is even faster, since it requires no action at all. To `deletemin`, on the other hand, requires a linear-time scan of the list.

5.2 Binary heap

Here elements are stored in a *complete* binary tree, namely a binary tree in which each level is filled in left-to-right, and must be full before the next level is started (Figure 5.1). In addition, the key value of any node of the tree is less than or equal to that of its children, so in particular the root always contains the smallest element.

To `insert`, place the new element at the bottom of the tree (in the first available position), and let it "bubble up". That is, if it is smaller than its parent, swap the two, and repeat. The number of swaps is at most the height of the tree, which is $\lceil \log_2 n \rceil$ when there are n elements. A `decreasekey` is accomplished similarly.

To `deletemin`, return the root value. Now this element has to be removed from the heap, so take the last node in the tree (in the rightmost position in the bottom row), and place it at the root. Let it "sift down": if it is bigger than either child, swap it with the smaller child, and repeat. Again this takes $O(\log n)$ time.

5.3 d-ary heap

A d-ary heap is identical to a binary heap, except that nodes have d children instead of just two. This reduces the height of a tree with n elements to $\lceil \log_d n \rceil = (\log n)/(\log d)$. Inserts are therefore speeded up by a factor of $O(\log d)$. Deletemins, however, take a little longer, $O(d \log_d n)$ (do you see why?).

The regularity of a complete d-ary tree makes it easy to represent using an array. The tree nodes have a natural ordering: row by row, starting at the root, and left-to-right within each row. If there are n nodes, this ordering specifies their positions $0, 1, \ldots, n-1$ within the array. Moving up and down the tree is easily simulated on the array, using the fact that node number j has parent $\lceil j/d \rceil$ and children $\{jd+2, \ldots, \min\{n-1, jd+d+1\}\}$ (see exercise and figure).

The array h is assumed to support the following constant-time operations: $|h|$, which returns the number of elements in the array; and h^{-1}, which returns the position of an element within the array.[3] Figure 5.2 gives pseudcode for a d-ary heap in terms of these primitives.

[3]This can always be achieved by maintaining the values of h^{-1} as an auxiliary array.

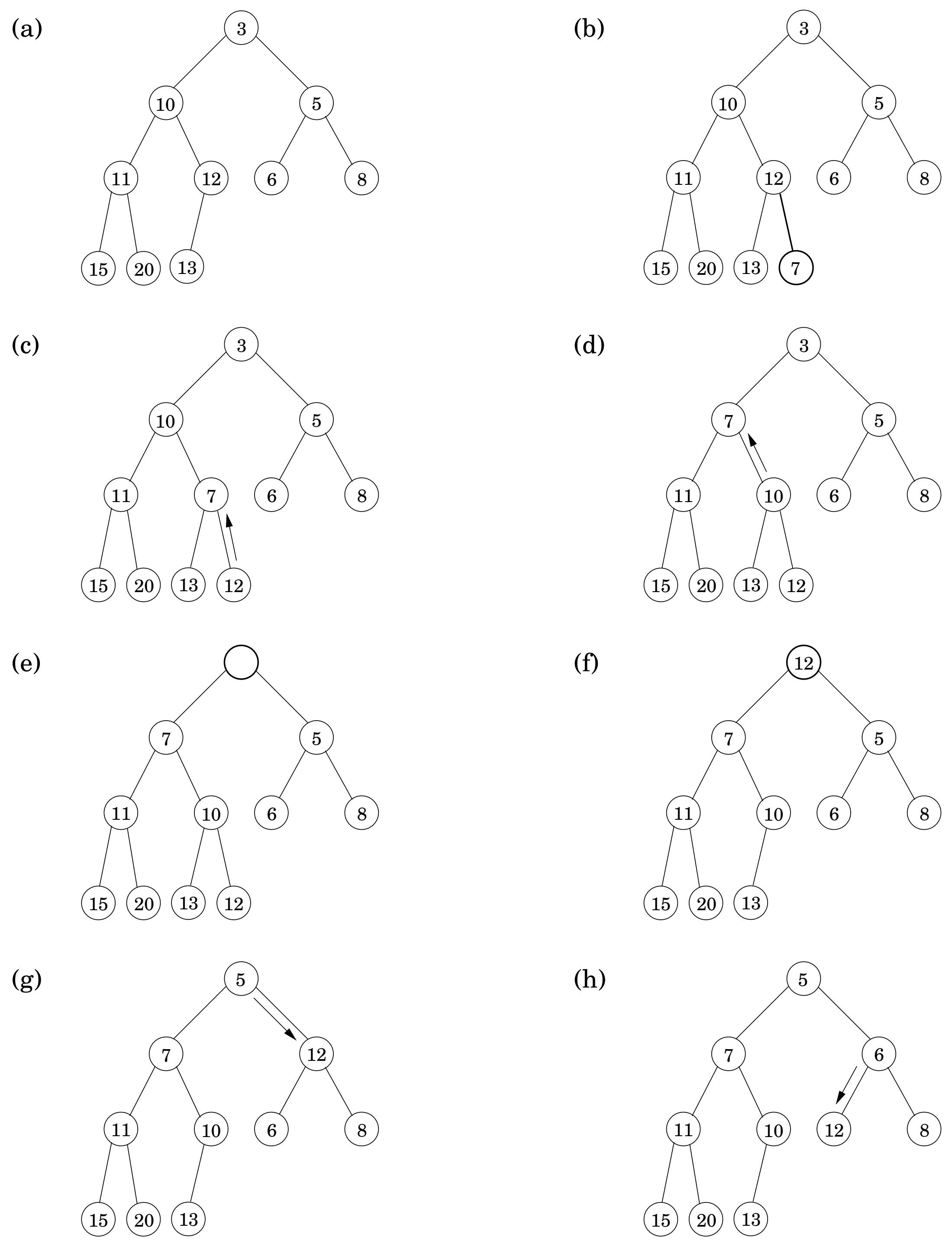
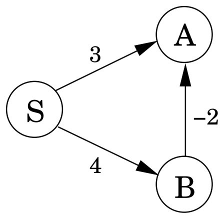


Figure 5.1 (a) A binary heap with ten elements. Only the key values are shown. (b)–(d) The intermediate “bubble-up” steps in inserting an element with key 7. (e)–(g) The “sift-down” steps in a delete-min operation.

Figure 5.2 Operations on a d-ary heap, modeled on an exposition by R.E. Tarjan.

```
procedure insert($h, x$)
bubbleup($h, x, |h|$)

procedure decreasekey($h, x$)
bubbleup($h, x, h^{-1}(x)$)

procedure deletemin($h$)
if $h$ is empty:
    return null
else:
    $x = h(0)$
    siftdown($h, h(|h|-1), 0$)
    return $x$

procedure makeheap($S$)
$h =$ empty array of size $|S|$
for $x \in S$:
    $h(|h|) = x$
for $i = |S| - 1$ downto $0$:
    siftdown($h, h(i), i$)
return $h$

procedure bubbleup($h, x, i$)
$p = \lceil i/d \rceil$
while $i \neq 0$ and key$(h(p)) >$ key$(x)$:
    $h(i) = h(p);\quad i = p;\quad p = \lceil i/d \rceil$
$h(i) = x$

procedure siftdown($h, x, i$)
$c =$ minchild$(h, i)$
while $c \neq 0$ and key$(h(c)) <$ key$(x)$:
    $h(i) = h(c);\quad i = c;\quad c =$ minchild$(h, i)$
$h(i) = x$

procedure minchild($h, i$)
if $di + 2 > |h| - 1$:
    return 0 (no children)
else:
    return $\arg\min\{$key$(h(j)) : di + 1 \leq j \leq \min\{|h| - 1, di + d + 1\}\}$
```

Figure 6.1 Dijkstra's algorithm will not work if there are negative edges.



6 A general shortest-paths algorithm

6.1 Negative edges

One of the basic principles behind Dijkstra's algorithm is that the shortest path from the starting point s to any node v must pass exclusively through nodes which are closer than v. This no longer holds in the presence of negative edge lengths. In Figure 6.1, the shortest path from S to A passes through B, a node which is further away!

What needs to be changed in order to accommodate this new complication? To answer this, let's take a particular high-level view of Dijkstra's algorithm. A crucial invariant is that the `dist` values it maintains are always either overestimates or exactly correct. They start off at ∞, and the only way they ever change is by updating along an edge:

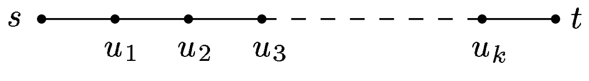
```
procedure update($(u,v) \in E$)
if dist($v$) $>$ dist($u$) $+ l(u,v)$:
    dist($v$) $=$ dist($u$) $+ l(u,v)$
    prev($v$) $= u$
```

This *update* operation is simply an expression of the fact that the distance to v cannot possibly be more than the distance to u, plus $l(u,v)$. It has the following properties.

1. It gives the correct distance to v in the particular case where u is the second-last node in the shortest path to v, and dist(u) is correctly set.

2. It will never make `dist`(v) too small, and in this sense it is *safe*. For instance, a slew of extraneous `update`'s can't hurt.

This operation is extremely useful: it is harmless, and if used carefully, will correctly set distances. In fact, Dijkstra's algorithm is a sequence of `update`'s. We know it doesn't work with negative edges, but is there some other sequence which does? To get a sense of the properties this sequence must possess, let's pick a node t and look at the shortest path to it from s.



This path can have at most $|V|-1$ edges (do you see why?). If the overall sequence of updates performed includes $(s,u_1), (u_1,u_2), (u_2,u_3), \ldots, (u_k,t)$, *in that order*, then by the first property the distance to t will be correctly computed. It doesn't matter what other updates occur on these edges, or what happens in the rest of the graph, because updates are *safe*.

Figure 6.2 A single-source shortest-path algorithm for general graphs.

```
procedure shortest-paths(G, l, s)
Input:    Graph $G = (V, E)$, directed or undirected;
          edge lengths $\{l_e : e \in E\}$; vertex $s \in V$
Output:   For all vertices $u$ reachable from $s$, dist($u$) is set
          to the distance from $s$ to $u$.

for all $u \in V$:
   dist($u$) $= \infty$
   prev($u$) $=$ nil

dist($s$) $= 0$
for $k = 1$ to $|V| - 1$:
   for all $e \in E$:
      update($e$)
```

But still, if we don't know all the shortest paths beforehand, how can we be sure to update the right edges in the right order? Here is an easy solution: simply update *all* the edges, $|V| - 1$ times! The resulting algorithm is shown in Figure 6.2, with an example in Figure 6.3.

As with Dijkstra's algorithm, there *is* a region R for which `dist` values are correct, and this region grows on each iteration. However it grows in a haphazard way and, except at the very beginning (when it is $\{s\}$) and the very end (when it is the whole graph), we never know what it looks like. The overall running time is $O(|V| \cdot |E|)$: the region grows by at least one node at a time, and each of these expansions is achieved through $|E|$ update operations. In contrast, Dijkstra's algorithm exploits the positive edges to grow the region in a carefully controlled way, with the benefit that each expansion is accomplished more economically.

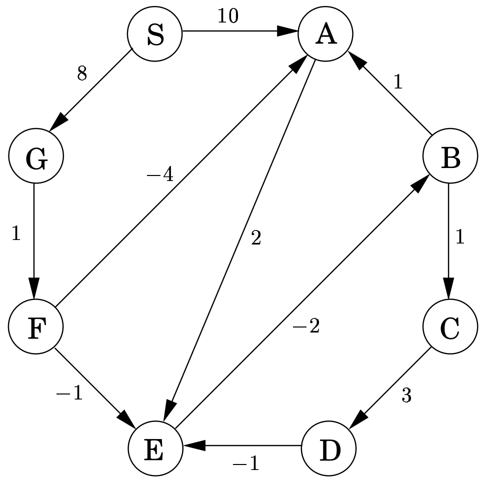
6.2 Negative cycles

If the length of edge (E, B) in Figure 6.3 were changed to -4, the graph would have a *negative cycle* $A \to E \to B \to A$. In such situations, it doesn't make sense to even ask about shortest paths. There is a path of length 2 from A to E. But going round the cycle, there's also a path of length 1, and going round multiple times, we find paths of length $0, -1, -2$, and so on.

The shortest path problem is ill-posed in graphs with negative cycles. As might be expected, our algorithm from the previous section works only in the absence of such cycles. But where did this assumption appear in the analysis? It slipped in when we asserted the *existence* of a shortest path...

Fortunately, it is easy to automatically detect negative cycles and issue a warning. Such a cycle would allow us to endlessly apply rounds of `update` operations, reducing `dist` estimates every time. So instead of stopping after $|V| - 1$ iterations, perform one extra round. There is a negative cycle if and only if some `dist` value is reduced during this final round.

Figure 6.3 The general shortest-paths algorithm illustrated on a sample graph.



	Iteration							
Node	0	1	2	3	4	5	6	7
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

7 Shortest paths in dags

There are two subclasses of graphs which automatically exclude the possibility of negative cycles: graphs without negative edges, and graphs without cycles. We already know how to efficiently handle the former. We will now see how the single-source shortest path problem can be solved in just linear time on directed acyclic graphs.

As before, we need to perform a sequence of updates which includes every shortest path as a subsequence. The key source of efficiency is that

In any path of a dag, the vertices appear in increasing topological order.

Therefore, it is enough to topologically sort the dag by depth-first search, and then visit the vertices in sorted order, updating the edges out of each. The algorithm is given in Figure 7.1.

As a further simplification, it isn't necessary to actually topologically order the graph. Instead, the edges can be updated as they appear in breadth-first search from s. Also notice that our scheme doesn't require edges to be positive. By negating edges, we can use this same algorithm to find the *longest path* in a dag.

Figure 7.1 A single-source shortest-path algorithm for directed acyclic graphs.

```
procedure dag-shortest-paths($G, l, s$)
Input:    Dag $G = (V, E)$;
          edge lengths $\{l_e : e \in E\}$; vertex $s \in V$
Output:   For all vertices $u$ reachable from $s$, dist($u$) is set
          to the distance from $s$ to $u$.

for all $u \in V$:
   dist($u$) $= \infty$
   prev($u$) $=$ nil

dist($s$) $= 0$
Topologically sort $G$ by depth-first search
for each $u \in V$, in topological order:
   for all edges $(u, v) \in E$:
      update($u, v$)
```