**9.3** Shortest**-Path Algorithms**

**391**

*V4*

**V3**

V2

**V1**

**V9**

**V8**

**V7**

V6

V5

Figure 9.17 A bad case for unweighted shortest-path algorithm using Figure 9.16

By tracing back through the *p*, variable, the actual path can be printed. We will see how when we discuss the weighted case.

The running time of the algorithm is *O*(|V|2), because of the doubly nested for loops. An obvious inefficiency is that the outside loop continues until NUM\_VERTICES**-1**, even if all the vertices become *known* much earlier. Although an extra test could be made to avoid this, it does not affect the worst-case running time, as can be seen by generalizing what happens when the input is the graph in Figure 9.17 with start vertex v9.

We can remove the inefficiency in much the same way as was done for topological sort. At any point in time, there are only two types of unknown vertices that have *dy* #∞. Some have *dy* **= currDist**, and the rest have *dy* = **currDist +** 1. Because of this extra structure, it is very wasteful to search through the entire table to find a proper vertex.

A very simple but abstract solution is to keep two boxes. Box #1 will have the unknown vertices with *dy* = **currDist**, and box #2 will have *dy* = **currDist + 1.** The test to find an appropriate vertex v can be replaced by finding any vertex in box #1. After updating *w* (inside the innermost **if** block), we can add w to box #2. After the outermost for loop terminates, box #1 is empty, and box #2 can be transferred to box #1 for the next pass of the for loop.

We can refine this idea even further by using just one queue. At the start of the pass, the queue contains only vertices of distance currDist. When we add adjacent vertices of distance currDist **+ 1**, since they enqueue at the rear, we are guaranteed that they will not be processed until after all the vertices of distance **currDist** have been processed. After the last vertex at distance currDist dequeues and is processed, the queue only contains vertices of distance currDist **+** 1, so this process perpetuates. We merely need to begin the process by placing the start node on the queue by itself.

The refined algorithm is shown in Figure 9.18. In the pseudocode, we have assumed that the start vertex, s, is passed as a parameter. Also, it is possible that the queue might empty prematurely, if some vertices are unreachable from the start node. In this case, **a** distance of **INFINITY** will be reported for these nodes, which is perfectly reasonable. Finally, the **known** data member is not used; once a vertex is processed it can never enter the queue again, so the fact that it need not be reprocessed is implicitly marked. Thus, the **known** data member can be discarded. Figure 9.19 shows how the values on the graph we have been using are changed during the algorithm (it includes the changes that would occur to **known** if we had kept it).

Using the same analysis as was performed for topological sort, we see that the running time is *O*(|E| + |V|), as long as adjacency lists are used.

**9.3.2 Dijkstra's Algorithm**

If the graph is weighted, the problem (apparently) becomes harder, but we can still use the ideas from the unweighted case.

392

Chapter **9 Graph** Algorithms

**void Graph:: unweighted ( Vertex s )**

**{**

**Queue<Vertex> q;**

**for each Vertex v**

**v.dist =** INFINITY**;**

**s.dist** = **0;**

**q.enqueue(s)**;

**while(**!**q.isEmpty()**)

**{**

**Vertex v** = **q.dequeue()**;

**for each Vertex w adjacent to v**

**if(w.dist ==** INFINITY **)**

}

**{**

**w.dist = v.dist + 1;**

}

**w.path = v;**

q.enqueue**(w);**

**Figure 9.18** Psuedocode for unweighted shortest-path algorithm

We keep all of the same information *as* before. Thus, each vertex is marked as either *known* or unknown. A tentative distance *dy* is kept for each vertex, as before. This dis- tance turns out to be the shortest path length from s to v using only known vertices as intermediates. As before, we record *pv*, which is the last vertex to cause a change to *dy*.

The general method to solve the single-source shortest-path problem is known as **Dijkstra's algorithm**. This thirty-year-old solution is a prime example of a **greedy algo- rithm.** Greedy algorithms generally solve a problem in stages by doing what appears to be the best thing at each stage. For example, to make change in U.S. currency, most people count out the quarters first, then the dimes, nickels, and pennies. This greedy algo- rithm gives change using the minimum number of coins. The main problem with greedy algorithms is that they do not always work. The addition of a 12-cent piece breaks the coin-changing algorithm for returning 15 cents, because the answer it gives (one 12-cent piece and three pennies) is not optimal (one dime and one nickel).

Dijkstra's algorithm proceeds in stages, just like the unweighted shortest-path algo- rithm. At each stage, Dijkstra's algorithm selects a vertex, v, which has the smallest *dy* among all the unknown vertices and declares that the shortest path from s to v is *known*. The remainder of a stage consists of updating the values of *dw*.

=

In the unweighted case, *we* set *dw dv* + 1 if *dw* = ∞. Thus, we essentially lowered the value of *d1*, if vertex v offered a shorter path. If we apply the same logic to the weighted

**9.3 Shortest**-**Path** Algorithms

**393**

Initial State

v3 Dequeued

vi Dequeued

v6 Dequeued

ν *known*

*dv*

Pv *known*

*dy*

Pv

*known*

dv

Pv

*known*

dv

Pv

V1

F

حد تک ممے ہیں

V4

**V2**

V3

V5

V6

V7

**F**

F

F

F

8 8 8 8 0 8 8

0

F

1

V3

Τ

0

F

0

T

0

F

0

F

**8** 8 0 8

**0**

**0**

Т

0

0

F

1

V3

~ 8

HFT F F F

H2O

1

V3

Τ

2

**V1**

0 0

**V1**

**0**

Τ

**1**

V3

Τ

0

F

8

0

F

0

1202 8

TFT F

FTF

V3

V1

0

V1

**0**

1

V3

8

F

**F**

Q:

V3

v2 Dequeued

**V1,** V6

V4 Dequeued

V6, V2, V4

V2, V4

*V5* Dequeued

V7 Dequeued

ν

*known*

*dv*

*Pv*

known

*dv*

*Pv*

*known*

dv

Pv

known

dv

*Pv*

V2

سے منہ سب مجھ میں ہو گا

T

1

V3

T

1

V3

T

T 2

V1

T

2

V1

T

**V3**

T

V4

T F

0

0

T

**0**

**0**

Τ

2

V1

T

2

V1

V5

**V6**

V7

Q:

FTF

F 3 V2

F

3

V2

1

**V3**

T

1

V3

0

F

3

V4

HH

HHHTF

**1** V3

T

1

V3

2

V1

T

2

V1

**0**

0

T

**0**

**0**

Τ

2

**V1**

T

2

V1

Τ

3

**V2**

T

3

V2

Τ

1

**V3**

T

1

V3

3

V4

T

3

V4

V4, V5

V5, V7

V7

empty

Figure **9.19** How the data change during the unweighted shortest-path algorithm

=

case, then we should set *dw dy***+**Cv,w if this new value for *dw* would be an improvement. Put simply, the algorithm decides whether or not it is a good idea to use v on the path to *w.* The original cost, *dw*, is the cost without using v; the cost calculated above is the cheapest path using v (and only *known* vertices).

The graph in Figure 9.20 is our example. Figure 9.21 represents the initial config- uration, assuming that the start node, s, is v1. The first vertex selected is v1, with path length 0. This vertex is marked *known*. Now that vi is known, some entries need to be adjusted. The vertices adjacent to v1 are v2 and v4. Both these vertices get their entries adjusted, as indicated in Figure 9.22.

Next, **v4** is selected and marked *known*. Vertices v3, V5, V6, and v7 are adjacent, and it turns out that all require adjusting, as shown in Figure 9.23.

Next, v2 is selected. v4 is adjacent but already known, so no work is performed on it. v5 is adjacent but not adjusted, because the cost of going through v2 is 2 + 10 = 12 and a path of length 3 is already known. Figure 9.24 shows the table after these vertices are **selected**.

394

Chapter **9 Graph** Algorithms

V3

**5**

2

**V2**

*V1*

1

**3**

10

2

**2**

**V5**

V4

**8**

**4**

6

**1**

V7

V6

Figure **9.20** The directed graph *G* (again)

ν

**known**

*dv*

*Pv*

V1

F

V2

F

V3

F

V4

V5

11 I

F

F

V6

F

V7

F

8 8 8 8 8 8 0

0

0

0

0

0

0

**Figure 9.21** Initial configuration of table used in Dijkstra's algorithm

V *known*

dv

*Pv*

V1

Τ

V2

F

V3

F

ON 8

HF

**0**

0

2

**V1**

**0**

V4

F

1

V1

V5

F

**LL**

V6

F

V7

F

888

**0**

0

**Figure 9.22** After v1 is declared *known*

The next vertex selected is v5 at cost 3. v7 is the only adjacent vertex, but it is not adjusted, because 3 +6 > 5. Then v3 is selected, and the distance for v6 is adjusted down to 3+ 5 = 8. The resulting table is depicted in Figure 9.25.

Next, v7 is selected; v6 gets updated down to 5 + 1 = 6. The resulting table is Figure 9.26.

ν ***known***

*dv*

*Pv*

V1

T

0

0

V2

F

2

V1

**V3**

F

3

V4

V4

T

1

V1

V5

F

3

V4

F

9

V4

V6

F

5

V7

V4

**Figure 9.23** After v4 is declared *known*

V *known*

dv

*Pv*

V1

Τ

V2

**V3**

V4

V5

THE H I F

0

0

T 2

**V1**

F

3

V4

1

V1

F

3

V4

9

V4

V6

V7

F

5

V4

**Figure 9.24** After v2 is declared *known*

ν *known*

dv

*Pv*

**V1**

T

V2

T

**V3**

V4

V5

V6

HHHHHF

**0**

0

2

V1

T

3

V4

T

1

V1

T

3

V4

8

**V3**

V7

F

5

V4

**9.3 Shortest**-**Path** Algorithms

Figure **9.25** After v5 and then v3 are declared known

Finally, v6 is selected. The final table is shown in Figure 9.27. Figure 9.28 graphically shows how edges are marked *known* and vertices updated during Dijkstra's algorithm.

To print out the actual path from a start vertex to some vertex v, we can write a recursive routine to follow the trail left in the p variables.

We **now** give pseudocode to implement Dijkstra's algorithm. Each Vertex stores various data members that are used in the algorithm. This is shown in Figure 9.29.

395

396

**Chapter 9 Graph Algorithms**

**ν**

known

dv

*Pv*

T

**0**

**0**

V1

12

T

2

**V1**

V3

T

3

V4

T

1

V1

V4

V5

T

3

**V4**

V6

V7

F H

6

V7

T

5

V4

Figure **9.26** After v7 is declared known

ν

**known**

*dv*

*Pv*

V1

T

0

0

V2

T

2

V1

**V3**

T

3

V4

V4

T

1

V1

V5

T

3

V4

V6

T

6

V7

V7

T

5

V4

**Figure 9.27** After v6 is declared *known* and algorithm terminates

The path can be printed out using the recursive routine in Figure 9.30. The routine recursively prints the path all the way up to the vertex before v on the path, and then just prints v. This works because the path is simple.

Figure 9.31 shows the main algorithm, which is just a **for** loop to fill up the table using the greedy selection rule.

A proof by contradiction will show that this algorithm always works as long as no edge has a negative cost. If any edge has negative cost, the algorithm could produce the wrong answer (see Exercise 9.7(a)). The running time depends on how the vertices are manipulated, which we have yet to consider. If we use the obvious algorithm of sequentially scanning the vertices to find the minimum *d,,* each phase will take O(|V|) time to find the minimum, and thus *O*(|V|2) time will be spent finding the minimum over the course of the algorithm. The time for updating *dw* is constant per update, and there is at most one update per edge for a total of O(*|E*|). Thus, the total running time is *O*(*|*E|+|V|2) = *O*(|V|2). If the graph is dense, with |E| = ✪(|V|2), this algorithm is not only simple but also essentially optimal, since it runs in time linear in the number of edges.

If the graph is sparse, with |*E*|

=

(VI), this algorithm is too slow. In this case, the distances would need to *be* kept in a priority queue. There are actually two ways to do this; both are similar.

**V3**

**9.3** Shortest**-Path** Algorithms

397

**V5**

∞

**\***

*V*

1

**4**

**0**

**2**

**V2**

**3**

10

**2**

**2**

180

V4

**8**

**4**

6

1

V6

*V7*

∞

**V1**

2

**V6**

**0**

**2**

12

1

**3**

**10**

2

**V3**

**V5**

**8**

V4

1

18+

**4**

V7

∞

५४

2

**\***

**V3**

**3**

**4**

**5**

V1

2

**8**

12

**0**

1

**3**

10

2

\*

V4

**V5**

4

**6**

1

V6

**V7**

**5**

**3**

A

**2**

V1

*V2*

4

1

**3**

**10**

2

2

**\***

**V3**

VA

**25**

**8**

**6**

1

V6

**V75**

**V3**

**\***

**5**

**\***

\*

**2**

**4**

1

**3**

10

2

2

**\***

***4***

**15'**

**5**

**8**

**4**

**6**

1

**V6**

\*

**6**

***V7***

5

**V3**

A

5

2

12

1

1

**3**

10

**2**

**2**

V5

V4

**8**

4

6

1

**V7**

V6

\*

**3**

w

5

2

1

2

**8**

V6 **8**

**3**

2

**10**

♡

να

4*.*

**\***

ν

**4**

6

1

V7

4.

**V1**

2

**0**

**1**

2

**\***

**3**

10

**2**

\*

***(V5***

**3**

**8**

**4**

6

1

V6

**X**

\*

**V7**

\*

**V3**

**3**

Figure **9.28** Stages of Dijkstra's algorithm

Selection of the vertex v is a **deleteMin** operation, since once the unknown minimum vertex is found, it is no longer unknown and must be removed from future consideration. The update of w's distance can be implemented two ways.

One way treats the update as a **decreaseKey** operation. The time to find the minimum is then O(log |*V*|), as is the time to perform updates, which amount to **decreaseKey** operations. This gives a running time of *O*(|E| log |V| + |V| log |V|) : *O*(*|E*| log |*V*|), an improvement

=

398

Chapter **9 Graph** Algorithms

**/\*\***

**\* PSEUDOCODE sketch of the Vertex structure.**

**\* In real C++, path would be of type Vertex \***

?

**\* and many of the code fragments that we describe**

\*

**\* require either a dereferencing or use the**

**\* -> operator instead of the . operator.**

**\* Needless to say, this obscures the basic algorithmic ideas.**

**\*/**

**struct Vertex**

{

**List**

**bool**

**adj;**

// **Adjacency list**

**known;**

**DistType dist;**

// **DistType is probably** int

**Vertex path;**

*//* **Probably Vertex \*, as mentioned above**

**}**;

// **Other data and member functions as needed**

**Figure 9.29 Vertex** class for Dijkstra's algorithm (pseudocode)

**/\*\***

**\* Print shortest path to v after dijkstra has run**.

**\* Assume that the path exists.**

**\*/**

**void Graph::printPath( Vertex** v **)**

{

**if( v.path = NOT\_A\_VERTEX )**

{

**printPath(v.path )**;

**cout << to "**;

**}**

**cout** << **v;**

}

Figure **9.30** Routine to print the actual shortest path

over the previous bound for sparse graphs. Since priority queues do not efficiently support the **find** operation, the location in the priority queue of each value of *di* will need to be maintained and updated whenever ***di*** changes in the priority queue. If the priority queue is implemented by a binary heap, this will be messy. If a pairing heap (Chapter 12) is used, the code is not too bad.

An alternate method is to insert w and *the* new value *dw* into the priority queue every time w's distance changes. Thus, there may be more than one representative for each vertex in the priority queue. When the **deleteMin** operation removes the smallest vertex from the priority queue, it must be checked to make sure that it is not already *known* and, if

**void Graph::dijkstra( Vertex s )**

{

**for each Vertex v**

}

=

**v.dist** INFINITY**; v.known**

-

**false;**

}

}

**s.dist**

= **0;**

**while there is an unknown distance vertex )**

{

**Vertex** v **= smallest unknown distance vertex;**

**v.known**

**true;**

**for each Vertex w adjacent to v**

**if(w.known)**

**{**

**DistType cvw = cost of edge** from **v to w;**

**if( v.dist + cvw <w.dist )**

{

// **Update w**

**decrease(w.dist to v.dist + cvw); w.path = v;**

}

**Figure 9.31** Pseudocode for Dijkstra's algorithm

**9.3 Shortest**-**Path** Algorithms

399

it is, it is simply ignored and another **deleteMin** is performed. Although this method is superior from a software point of view, and is certainly much easier to code, the size of the priority queue could get to be as large as [*E*]. This does not affect the asymptotic time bounds, since |*E*| ≤ |V|2 implies that log |E| ≤ 2 log |V|. Thus, **we still** get an *O*(|*E*| log |V|) algorithm. However, the space requirement does increase, and this could be important in some applications. Moreover, because this method requires |*E*| deleteMins instead of only *IV*, it is likely to be slower in practice.

Notice that for the typical problems, such as computer mail and mass transit com- mutes, the graphs are typically very sparse because most vertices have only a couple of edges, so it is important in many applications to use a priority queue to solve this problem.

There are better time bounds possible using Dijkstra's algorithm if different data struc- tures are used. In Chapter 11, we will see another priority queue data structure called the

400

**Chapter 9 Graph Algorithms**

Fibonacci heap. When this is used, the running time is O(|*E*|+|V| log |*V*|). Fibonacci heaps have good theoretical time bounds but a fair amount of overhead, so it is not clear whether using Fibonacci heaps is actually better in practice than Dijkstra's algorithm with binary heaps. To date, there are no meaningful average-case results for this problem.

9.3.3 **Graphs** with **Negative** Edge **Costs**

**If** the graph has negative edge costs, then Dijkstra's algorithm does not work. The problem is that once a vertex, ***u***, is declared *known*, it is possible that from some other *unknown* vertex, v, there is a path back to *u* that is very negative. In such a case, taking a path from s to v back to u is better than going from s to u without using v. Exercise 9.7(a) asks you to construct an explicit example.

A tempting solution is to add a constant A to each edge cost, thus removing negative edges, calculate a shortest path on the new graph, and then use that result on the original. The naive implementation of this strategy does not work because paths with many edges become more weighty than paths with few edges.

A combination of the weighted and unweighted algorithms will solve the problem, but at the cost of a drastic increase in running time. We forget about the concept of known vertices, since our algorithm needs to be able to change its mind. We begin by placing s on a queue. Then, at each stage, we dequeue a vertex v. *We* find all vertices *w* adjacent to v such that *dw* > dy + Cv,w. We update ***dw*** and pw, and place *w* on a queue if it is not already there. A bit can be set for each vertex to indicate presence in the queue. We repeat the

process until the queue is empty. Figure 9.32 (almost) implements this algorithm. Although the algorithm works if there are no negative-cost cycles, it is no longer true that the code in the inner **for** loop is executed once per edge. Each vertex can dequeue at most |V| times, so the running time is O(|*E*| · |V|) if adjacency lists are used (Exercise 9.7(b)). This is quite an increase from Dijkstra's algorithm, so it is fortunate that, in practice, edge costs are nonnegative. If negative-cost cycles are present, then the algorithm as written will loop indefinitely. By stopping the algorithm after any vertex has dequeued IV + 1 times, we can guarantee termination.

**9.3.4 Acyclic Graphs**

If the graph is known to be acyclic, we can improve Dijkstra's algorithm by changing the order in which vertices are declared *known*, otherwise known as the vertex selection rule. The new rule is to select vertices in topological order. The algorithm can be done in one pass, since the selections and updates can take place as the topological sort is being performed.

This selection rule works because when a vertex v is selected, its distance, *dy*, can no longer be lowered, since by the topological ordering rule it has no incoming edges emanating from *unknown* nodes.

There is no need for a priority queue with this selection rule; the running time is *O*(|*E*| + |V|), since the selection takes constant time.

An acyclic graph could model some downhill skiing problem-we want to get from point a to b, but can only go downhill, so clearly there are no cycles. Another possible

**void Graph::weighted Negative( Vertex s )**

{

**Queue<Vertex> q**;

**for each Vertex v**

**v.dist =** INFINITY**;**

**s.dist = 0;**

**q.enqueue(s);**

**while(!q.isEmpty() )**

{

**Vertex** v **=**

**q.dequeue()**;

**for each Vertex w adjacent to v**

**if( v.dist** + **cvw** <**w.dist )**

}

}

**{**

// **Update w**

**w.dist** = **v.dist + cvw;**

**w.path = v;**

**if( w is not already in** q **)**

**q.enqueue( w )**;

**9.3 Shortest**-Path **Algorithms**

401

Figure **9.32** Pseudocode for weighted shortest-path algorithm with negative edge costs

application might be the modeling of (nonreversible) chemical reactions. We could have each vertex represent a particular state of an experiment. Edges would represent a transi- tion from one state to another, and the edge weights might represent the energy released. **If** only transitions from a higher energy state to a lower are allowed, the graph is acyclic.

A more important use of acyclic graphs is **critical path analysis**. The graph in Figure 9.33 will serve as our example. Each node represents an activity that must be per- formed, along with the time it takes to complete the activity. This graph is thus known as an *activity-node* graph. The edges represent precedence relationships: An edge (v, *w)* means that activity v must be completed before activity w may begin. Of course, this implies that the graph must be acyclic. We assume that any activities that do not depend (either *directly* or indirectly) on each other can be performed in parallel by different servers.

This type

of a graph could be (and frequently is) used to model construction projects. In this case, there are several important questions which would be of interest to answer. First, what is the earliest completion time for the project? We can see from the graph that 10 time units are required along the path *A*, *C, F*, H. Another important question is to deter- mine which activities can be delayed, and by how long, without affecting the minimum completion time. For instance, delaying any of *A*, *C, F*, or *H* would push the completion

402

**Chapter 9 Graph Algorithms**

**(A (3)**

【C **(3)**

**start**

F **(3)**

(D **(2)**

**(**H (1)

**-finish**

**B (2)**

G (2)

E (1)

**(K** (4)

**Figure 9.33** Activity-node graph

time past 10 units. On the other hand, activity *B* is less critical and can be delayed up to two time units without affecting the final completion time.

To perform these calculations, we convert the activity-node graph to an **event-node graph**. Each event corresponds to the completion of an activity and all its dependent activ- ities. Events reachable from a node v in the event-node graph may not commence until after the event v is completed. This graph can be constructed automatically or by hand. Dummy edges and nodes may need to be inserted in the case where an activity depends on several others. This is necessary in order to avoid introducing false dependencies (or false lack of dependencies). The event-node graph corresponding to the graph in Figure 9.33 is shown in Figure 9.34.

To find the earliest completion time of the project, we merely need to find the length of the *longest* path from the first event to the last event. For general graphs, the longest-path problem generally does not make sense, because of the possibility of **positive**-cost **cycles**. These are the equivalent of negative-cost cycles in shortest-path problems. If positive-cost cycles are present, we could ask for the longest *simple* path, but no satisfactory solution is known for this problem. Since the event-node graph is acyclic, we need not worry about cycles. In this case, it is easy to adapt the shortest-path algorithm to compute the earliest

**C/3**

**2**

**4**

0

**A/3**

F**/3**

0

7'

7

0

**D**/2

**H/1**

1

**6**'

**6**

0

0

10**'**

**10**

**B/**2

**G**/2

0

**8'**

**8**

E/1

**3**

**5**

**9**

**K**/4

**Figure 9.34** Event-node graph

**9.3 Shortest-Path Algorithms**

**403**

completion time for all nodes in the graph. If EC; is the earliest completion time for node i, then the applicable rules are

EC1 **= 0**

ECW

= max *(*EC1 + Cv,w)

**(v**,w)ЄE

Figure 9.35 shows the earliest completion time for each event in our example event-node graph.

We can also compute the latest time, LC, that each event can finish without affecting the final completion time. The formulas to do this are

LCn =

ECn

LCv

―

= \_min\_*(LCw* - Cv‚w)

(v,w)*ЄE*

These values can be computed in linear time by maintaining, for each vertex, a list of all adjacent and preceding vertices. The earliest completion times are computed for vertices by their topological order, and the latest completion times are computed by reverse topological order. The latest completion times are shown in Figure 9.36.

The **slack time** for each edge in the event-node graph represents the amount of time that the completion of the corresponding activity can be delayed without delaying the overall completion. It is easy to see that

*Slack*(v, w) = LCw - ECv**-**Cv,w

**3**

**6**

C**/3**

**2**

**4**

0

6

**9**

A**/3**

F/**3**

0 7'

7

0

**3**

D/2

1

**6'**

**6**

0

0

**7**

**B**/2

**0**

G/2

உம் மடம்

0

**8'**

**8**

**3**

0

**E**/1

**3**

5

9

**K**/4

**Figure 9.35** Earliest completion times

**C/**3

2

**4**

**0**

**A/3**

0

F/3

0

7'

7

0

D**/***2*

**H/1**

1

**6'**

**6**

**0**

**0**

(**10'**

10

**B**/2

**G**/2

**10**

0

**8'**

**8**

**E/**1

**3**

5

9

K/4

**Figure 9.36** Latest completion times

404

**Chapter 9 Graph Algorithms**

**3**

C**/3/**0

2

**4**

**6**

A**/3/**0

**F/3/0**

7'

7

**10**

D/2/1

1

**6**'

6

**H/1**/0 (10**')**

**10**

**B**/2*/*2

G/2/2

**10**

**8'**

**8**

E/1/2

**3**

**5**

9

K/4/2

**Figure 9.37** Earliest completion time, latest completion time, and slack

Figure 9.37 shows the slack (as the third entry) for each activity in the event-node graph. For each node, the top number is the earliest completion time and the bottom entry is the latest completion time.

Some activities have zero slack. These are critical activities, which must finish on sched- ule. There is at least one path consisting entirely of zero-slack edges; such a path is a **critical path**.

**9.3.5 All-Pairs Shortest Path**

Sometimes it is important to find the shortest paths between all pairs of vertices in the graph. Although we could just run the appropriate single-source algorithm |V| times, we might expect a somewhat faster solution, especially on a dense graph, if we compute all the information at once.

In Chapter 10, we will see an *O*(|V|3) algorithm to solve this problem for weighted graphs. Although, for dense graphs, this is the same bound as running a simple (non- priority queue) Dijkstra's algorithm [V] times, the loops are so tight that the specialized all-pairs algorithm is likely to be faster in practice. On sparse graphs, of course**,** it is faster to run |V| Dijkstra's algorithms coded with priority queues.

**9.3.6 Shortest** Path **Example**

In this section we write some C++ routines to compute word ladders. In a word ladder each word is formed by changing one character in the ladder's previous word. For instance, we can convert **zero** to **five** by a sequence of one-character substitutions as follows: **zero hero here hire fire five.**

This is an unweighted shortest problem in which each word is a vertex, and two **ver-** tices have edges (in both directions) between them if they can be converted to each other with a one-character substitution.

In Section 4.8, we described and wrote a C**++** routine that would create a **map** in which the keys are words, and the values are **vectors** containing the words that can result from a one-character transformation. As such, this **map** represents the graph, in adjacency list format, and we only need to write one routine to run the single-source unweighted shortest-path algorithm and a second routine to output the sequence of words, after the

1

2

3

*4*

5

6

{

7

8

9

*10*

*11*

12

13

{

*14*

*15*

16

*17*

*18*

19

**if( previousWord [ str ]**

*//* Runs **the shortest path calculation from the adjacency map, returning a vector**

// **that contains the sequence of word changes to get from first to second**. **unordered\_map<string, string>**

**findChain(const unordered\_map<string, vector<string>> & adjacentWords,**

**const string & first, const string & second)**

**unordered\_map<string, string> previousWord;**

**queue<string> q**;

**q.push( first )**;

**while(**!**q.empty())**

**string current** = **q.front(); q.pop()**; **auto itr adjacentWords.find( current );**

=

**const vector<string>** & **adj for(string & str : adj )**

-

**itr->second;**

|| **||**

==

"" **}**

20

{

21

22

**previousWord[ str** ] **= current; q.push(str);**

23

}

24

}

25

26

27

**previousWord[ first ] =** "";

**return previousWord;**

28

}

29

30

31

32

33

34

35

36

37

38

39

*40*

41

42

43 }

// **After the shortest path calculation has run, computes the vector that** // **contains the sequence of words changes to get from first to second. vector<string> getChain FromPreviousMap(**

{

**const unordered\_map<string, string> & previous, const string & second**)

**vector<string> result;**

**auto & prev**

=

**const\_cast<unordered\_map<**string**,** string**> &>(previous )**;

**for(string current =** second**; current !**= ""**; current = prev[ current ])**

**result.push\_back( current );**

**reverse( begin( result ), end result ) ); return result;**

**Figure 9.38** C++ code to find word ladders

406

Chapter **9 Graph** Algorithms

single-source shortest-path algorithm has completed. These two routines are both shown in Figure 9.38.

The first routine is **findChain**, which takes the **map** representing the adjacency lists and the two words to be connected and returns a **map** in which the keys are words, and the corresponding value is the word prior to the key on the shortest ladder starting at first. In other words, in the example above, if the starting word is **zero**, the value for key five is **fire**, the value for key **fire** is hire, the value for *key* **hire** is here, and so on. Clearly this provides enough information for the second routine, **getChainFromPreviousMap**, which can work its way backward.

**findChain** is a direct implementation of the pseudocode in Figure 9.18, and for sim- plicity, it assumes that first is a key in **adjacentWords** (this is easily tested prior to the call, or we can add extra code at line 16 that throws an exception if this condition is not satis- fied). The basic loop incorrectly assigns a previous entry for **first** (when the initial word adjacent to **first** is processed) so at line 25 that entry is repaired.

getChainFromPrevMap uses the **prev** map and **second,** which presumably is a key in the map and returns the words used to form **the** word ladder by working its way backward through prev. This generates the words backward, so the STL **reverse** algorithm is used to fix the problem. The cast at line 36 is needed because operator [] cannot be applied on an immutable **map**.

It is possible to generalize this problem to allow single-character substitutions that include the deletion of a character or the addition of a character. To compute the adjacency list requires only a little more effort: In the last algorithm in Section 4.8, every time a representative for word w in group *g* is computed, we check if the representative is a word in group g – 1. If it is, then the representative is adjacent to w (it is a single-character deletion), and *w* is adjacent to the representative (it is a single-character addition). It is also possible to assign a cost to a character deletion or insertion (that is higher than a simple substitution), and this yields a weighted shortest-path problem that can be solved with Dijkstra's algorithm.

**9.4 Network Flow Problems**

Suppose we are given a directed graph G = (*V,E*) with edge capacities Cv,w. These capacities could represent the amount **of** water that could flow through a pipe or the amount of traffic that could flow on a street between **two** intersections. *We* have two vertices: s, which we call the **source**, and *t,* which is the **sink**. Through any edge, (v, *w)*, at most cv,w units of "flow" may pass. At any vertex, v, that is not either s or *t,* the total flow coming in must equal the total flow going out. The maximum-flow problem is to determine the maximum amount of **flow** that can pass from s to t. As an example, for the graph in Figure 9.39 on the left the maximum flow is 5, as indicated by the graph on the right. Although this example graph is acyclic, this is not a requirement; our (eventual) algorithm will work even if the graph has a cycle.

As required by the problem statement, no edge carries more flow than its capacity. Vertex *a* has three units of flow coming in, which it distributes to c and *d*. Vertex *d* takes three units of flow from *a* and *b* and combines this, sending the result to t. A vertex can