Dijkstra's Shortes Path Algorithm i Jav

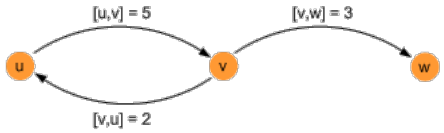
Dijkstra's algorithm is probably the best-known and thus most implemented shortest path algorithm. It is simple, easy to understand and implement, yet ir efficient. By getting familiar with such a sharp tool, a developer can solve efficiently and elegantly problems that would be considered impossibly hard oth my guest as I explore a possible implementation of Dijkstra's shortest path algorithm in Java.

What It Does

Dijkstra's algorithm, when applied to a graph, quickly finds the shortest path from a chosen source to a given destination. (The question "how quickly" is later in this article.) In fact, the algorithm is so powerful that it finds all shortest paths from the source to all destinations! This is known as the *single-sour* paths problem. In the process of finding all shortest paths to all destinations, Dijkstra's algorithm will also compute, as a side-effect if you will, a *spanni* the graph. While an interesting result in itself, the spanning tree for a graph can be found using lighter (more efficient) methods than Dijkstra's.

How It Works

First let's start by defining the entities we use. The graph is made of *vertices* (or nodes, I'll use both words interchangeably), and *edges* which link vertice Edges are directed and have an associated *distance*, sometimes called the weight or the cost. The distance between the vertex *u* and the vertex *v* is n and is always positive.



Dijkstra's algorithm partitions vertices in two distinct sets, the set of *unsettled* vertices and the set of *settled* vertices. Initially all vertices are unsettle algorithm ends once all vertices are in the settled set. A vertex is considered settled, and moved from the unsettled set to the settled set, once its shorte from the source has been found.

We all know that *algorithm + data structures = programs*, in the famous words of Niklaus Wirth. The following data structures are used for this algori

- **d** stores the best estimate of the shortest distance from the source to each vertex
- **π** stores the predecessor of each vertex on the shortest path from the source
- **S** the set of settled vertices, the vertices whose shortest distances from the source have been found
- **Q** the set of unsettled vertices

With those definitions in place, a high-level description of the algorithm is deceptively simple. With *s* as the source vertex:

```
// initialize d to infinity, π and Q to empty
d = ( ∞ )
π = ()
S = Q = ()

add s to Q
d(s) = 0

while Q is not empty
{
     u = extract-minimum(Q)
     add u to S
     relax-neighbors(u)
}
```

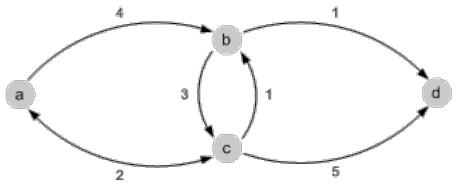
Dead simple isn't it? The two procedures called from the main loop are defined below:

```
relax-neighbors(u)
{
     for each vertex v adjacent to u, v not in S
     {
          if d(v) > d(u) + [u,v]     // a shorter distance exists
          {
               d(v) = d(u) + [u,v]
               π(v) = u
               add v to Q
          }
     }
```

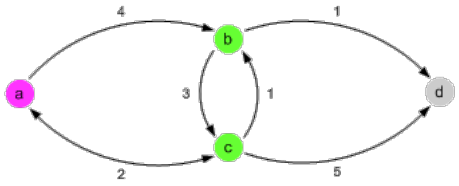
```
}

extract-minimum(Q)
{
    find the smallest (as defined by d) vertex in Q
    remove it from Q and return it
}
```

An Example

So far I've listed the instructions that make up the algorithm. But to really understand it, let's follow the algorithm on an example. We shall run Dikjstra path algorithm on the following graph, starting at the source vertex *a*.



We start off by adding our source vertex *a* to the set *Q*. *Q* isn't empty, we extract its minimum, *a* again. We add *a* to *S*, then relax its neighbors. (I recon follow the algorithm in parallel with this explanation.)

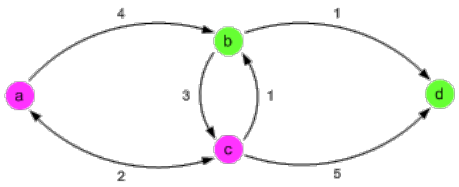


Those neighbors, vertices adjacent to *a*, are *b* and *c* (in green above). We first compute the best distance estimate from *a* to *b*. *d*(*b*) was initialized therefore we do:

```
d(b) = d(a) + [a,b] = 0 + 4 = 4
```

$\pi(b)$ is set to *a*, and we add *b* to *Q*. Similarily for *c*, we assign *d*(*c*) to 2, and $\pi(c)$ to *a*. Nothing tremendously exciting so far.

The second time around, *Q* contains *b* and *c*. As seen above, *c* is the vertex with the current shortest distance of 2. It is extracted from the queue and a the set of settled nodes. We then relax the neighbors of *c*, which are *b*, *d* and *a*.

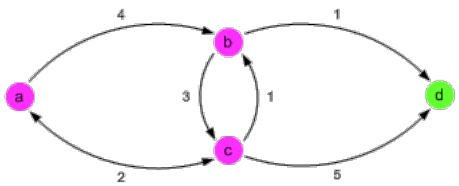


a is ignored because it is found in the settled set. But it gets interesting: the first pass of the algorithm had concluded that the shortest path from *a* to *b* Looking at *c*'s neighbor *b*, we realize that:

```
d(b) = 4 > d(c) + [c,b] = 2 + 1 = 3
```

Ah-ah! We have found that a shorter path going through *c* exists between *a* and *b*. *d*(*b*) is updated to 3, and $\pi(b)$ updated to *c*. *b* is added again to *Q*. Th adjacent vertex is *d*, which we haven't seen yet. *d*(*d*) is set to 7 and $\pi(d)$ to *c*.

The unsettled vertex with the shortest distance is extracted from the queue, it is now *b*. We add it to the settled set and relax its neighbors *c* and *d*.



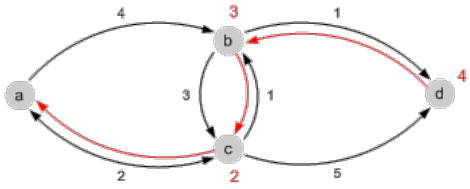
We skip *c*, it has already been settled. But a shorter path is found for *d*:

```
d(d) = 7 > d(b) + [b,d] = 3 + 1 = 4
```

Therefore we update *d*(*d*) to 4 and $\pi(d)$ to *b*. We add *d* to the Q set.

At this point the only vertex left in the unsettled set is *d*, and all its neighbors are settled. The algorithm ends. The final results are displayed in red below:

- π - the shortest path, in predecessor fashion
- d - the shortest distance from the source for each vertex



This completes our description of Dijkstra's shortest path algorithm. Other shortest path algorithms exist (see the References section at the end of this a Dijkstra's is one of the simplest, while still offering good performance in most cases.

Implementing It in Java

The Java implementation is quite close to the high-level description we just walked through. For the purpose of this article, my Java implementation o shortest path finds shortest routes between cities on a map. The `RoutesMap` object defines a weighted, oriented graph as defined in the introduction sec article.

```
public interface RoutesMap
{
    int getDistance(City start, City end);
    List<City> getDestinations(City city);
}
```

Data Structures

We've listed above the data structures used by the algorithm, let's now decide how we are going to implement them in Java.

S, the settled nodes set

This one is quite straightforward. The Java Collections feature the `Set` interface, and more precisely, the `HashSet` implementation which offers co performance on the `contains` operation, the only one we need. This defines our first data structure.

```
private final Set<City> settledNodes = new HashSet<City>();

private boolean isSettled(City v)
{
    return settledNodes.contains(v);
}
```

Notice how my data structure is declared as an abstract type (`Set`) instead of a concrete type (`HashSet`). Doing so is a good software engineering pra allows to change the actual type of the collection without any modification to the code that uses it.

d, the shortest distances list

As we've seen, one output of Dijkstra's algorithm is a list of shortest distances from the source node to all the other nodes in the graph. A straightforwa implement this in Java is with a `Map`, used to keep the shortest distance value for every node. We also define two accessors for readability, and to enca default infinite distance.

```
private final Map<City, Integer> shortestDistances = new HashMap<City, Integer>();

private void setShortestDistance(City city, int distance)
{
    shortestDistances.put(city, distance);
}

public int getShortestDistance(City city)
{
    Integer d = shortestDistances.get(city);
    return (d == null) ? INFINITE_DISTANCE : d;
}
```

You may notice I declare this field `final`. This is a Java idiom used to flag aggregation relationships between objects. By marking a field `final`, I convey that it is part of a aggregation relationship, enforced by the properties of `final`—the encapsulating class cannot exist without this field.

π, the predecessors tree

Another output of the algorithm is the predecessors tree, a tree spanning the graph which yields the actual shortest paths. Because this is the *predecesso* shortest paths are actually stored in reverse order, from destination to source. Reversing a given path is easy with `Collections.reverse()`.

The predecessors tree stores a relationship between two nodes, namely a given node's predecessor in the spanning tree. Since this relationship is one-t

akin to a *mapping* between nodes. Therefore it can be implemented with, again, a `Map`. We also define a pair of accessors for readability.

```
private final Map<City, City> predecessors = new HashMap<City, City>();

private void setPredecessor(City a, City b)
{
    predecessors.put(a, b);
}

public City getPredecessor(City city)
{
    return predecessors.get(city);
}
```

Again I declare my data structure to be of the abstract type `Map`, instead of the concrete type `HashMap`. And tag it `final` as well.

Q, the unsettled nodes set

As seen in the previous section, a data structure central to Dijkstra's algorithm is the set of unsettled vertices *Q*. In Java programming terms, we need able to store the nodes of our example graph, i.e. `City` objects. That structure is then looked up for the city with the current shortest distance given by *d*(

We could do this by using another `Set` of cities, and sort it according to *d*() to find the city with shortest distance every time we perform this operatior complicated, and we could leverage `Collections.min()` using a custom `Comparator` to compare the elements according to *d*().

But because we do this at every iteration, a smarter way would be to keep the set ordered at all times. That way all we need to do to get the city with distance is to get the first element in the set. New elements would need to be inserted in the right place, so that the set is always kept ordered.

A quick search through the Java collections API yields the `PriorityQueue`: it can sort elements according to a custom comparator, and provides co access to the smallest element. This is precisely what we need, and we'll write a comparator to order cities (the set elements) according to the curre distance. Such a comparator is included below, along with the `PriorityQueue` definition. Also listed is the small method that extracts the node with t distance.

```
private final Comparator<City> shortestDistanceComparator = new Comparator<City>()
    {
        public int compare(City left, City right)
        {
            int shortestDistanceLeft = getShortestDistance(left);
            int shortestDistanceRight = getShortestDistance(right);

            if (shortestDistanceLeft > shortestDistanceRight)
            {
                return +1;
            }
            else if (shortestDistanceLeft < shortestDistanceRight)
            {
                return -1;
            }
            else // equal
            {
                return left.compareTo(right);
            }
        }
    };

private final PriorityQueue<City> unsettledNodes = new PriorityQueue<City>(INITIAL_CAPACITY, shortestDistanceCompa

private City extractMin()
{
    return unsettledNodes.poll();
}
```

One important note about the comparator: it is used by the `PriorityQueue` to determine both object ordering and identity. If the comparator returr elements are equal, the queue infers they are the same, and it stores only one instance of the element. To prevent losing nodes with equal shortest distance compare the elements themselves (third block in the `if` statement above).

Having powerful, flexible data structures at our disposal is what makes Java such an enjoyable language (that, and garbage collection of course).

Putting It All Together

We have defined our data structures, we understand the algorithm, all that remains to do is implement it. As I mentioned earlier, my implementation is c high-level description given above. Note that when the only shortest path between two specific nodes is asked, the algorithm can be interrupted as s destination node is reached.

```
public void execute(City start, City destination)
{
    initDijkstra(start);

    while (!unsettledNodes.isEmpty())
    {
        // get the node with the shortest distance
```

```
        City u = extractMin();

        // destination reached, stop
        if (u == destination) break;

        settledNodes.add(u);

        relaxNeighbors(u);
    }
}
```

The `DijkstraEngine` class implements this algorithm and brings it all together. See "Implementation Notes" below to download the source code.

A Word About Performance

The complexity of Dijkstra's algorithm depends heavily on the complexity of the priority queue *Q*. If this queue is implemented naively as I first introduced re-ordered at every iteration to find the mininum node), the algorithm performs in $O(n^2)$, where n is the number of nodes in the graph.

With a real priority queue kept ordered at all times, as we implemented it, the complexity averages $O(n \log m)$. The logarithm function stems from the `PriorityQueue` class, a heap implementation which performs in $\log(m)$.

Implementation Notes

The Java source code discussed in this article is available for download as a ZIP file. Extensive unit tests are provided and validate the correctn implementation. Some minimal Javadoc is also provided. As the code makes use of the `assert` facility and generics, it must be compiled with "`javac 1.5`"; the tests require `junit.jar`. I warmly recommend Eclipse for all Java development.

I've received a fair amount of e-mail about this article, which has become quite popular. I'm unfortunately unable to answer all your questions, anc apologize. Keep in mind this article (and the code) is meant as a starting point: the implementation discussed here is hopefully simple, correct, and relativ understand, but is probably not suited to your specific problem. You must tailor it to your own domain.

My goal in writing this article was to share and teach a useful tool, striving for 1- simplicity and 2- correctness. I purposefully shied away from turning th into a full-blown generic Java implementation. Readers after full-featured, industrial-strength Java implementations of Dijkstra's shortest path algorithm s at the "Resources" section below.

Resources

- Any decent computer science curriculum includes at least one course on graph algorithms, which necessarily talks about Dijkstra's shortest path a recommend the excellent McGill University course, especially if my lightning-fast description of the algorithm left you bewildered. Here are som University of Western Australia, at Brigham Young University.
- The excellent *Introduction to Algorithms*, a book by Thomas Cormen et al. is a classic in algorithm litterature.
- The Boost Library implements Dijkstra's shortest path in C++.
- Many other algorithms exist to find the shortest path in a graph. They can resolve situations where Dijkstra's shortest path cannot be applied, e.g edges have negative weights. *A** is another popular shortest path algorithm; it requires a function to calculate the estimated remaining distance to with can greatly improve shortest path discovery. (1, 2)
- Dijkstra's shortest path algorithm is applied to real-world problems: Three Fastest Shortest Path Algorithms on Real Road Networks.
- JGL and JDSL provide full-featured, industrial-strength Java implementations of this algorithm.

Acknowledgments

The following entities have contributed to this article and deserve credit for it. I wish to address my sincere thanks to:

Pat Farrell <pfarrell@pfarrell.com>
: who shared with me his port to Java 5 (generics).

philip dahlquist <dahlquist@kreative.net>
: for typo corrections. Philip reports this document is used "as a reference in a data structures class (CMSc 420) at the university of Maryland, Co USA."

Carl Schwarcz <carl@schwarczclark.com>
: who pointed out a serious error in the original implementation of the shortest distance comparator.

the numerous readers of this article
: whose comments and criticism helped me improve it.

the ESSI faculty and teaching staff
: who taught me powerful algorithms and how to use them.

About the Author

Renaud Waldura is a software engineer specializing in distributed applications development. He started implementing graph algorithms in Java in 1995, l an educational Java applet to illustrate graph search strategies. Visit Renaud's Web site at http://renaud.waldura.com and learn more about how he ca solve your business problem with quality algorithms.

Last modified: 2007-08-20 15:16:45 -0700 (Mon, 20 Aug 2007) $