32

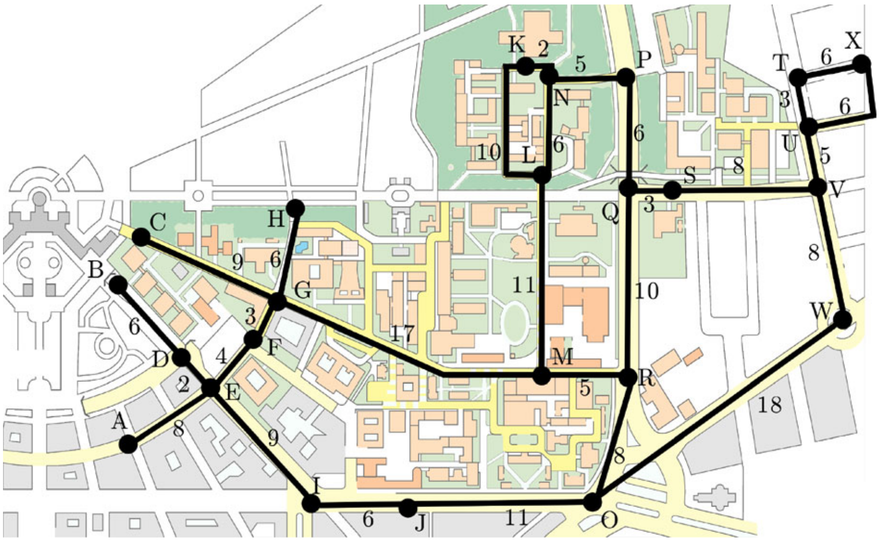
Shortest Paths

Peter Sanders and Johannes Singler

Karlsruher Institut für Technologie, Karlsruhe, Germany

I have just moved to Karlsruhe and into my first flat. Such a big city is rather complicated. I already have a city map, but how can I find the fastest way to get from A to B? I like cycling but I am notoriously impatient, so I really need the shortest path to the university, to my girlfriend, and so on.

Systematic planning could look like this: I pin the city map on a table and put thin yarn threads along the streets, knotting them at crossroads and junctions. I also knot all possible start points, end points and dead ends.



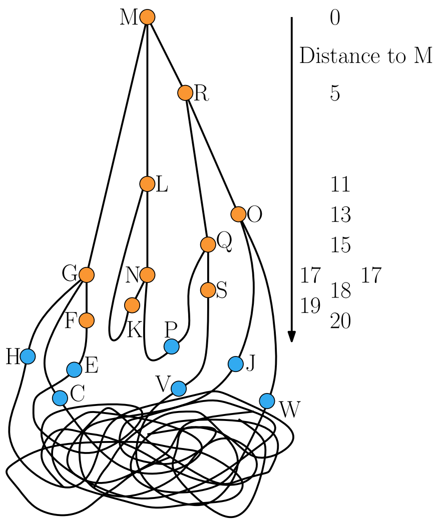
Copyright Karlsruher Institut für Technologie, Institut für Photogrammetrie und Fernerkundung

Here comes the trick: I pick the starting knot and slowly lift it up. One after another the knots leave the table surface. I have labeled the nodes so that

I always know where each knot comes from. At last, all knots hang vertically below the starting knot.

The rest is really easy: To find the shortest path, I only have to find the end knot and trace the straight threads back to the start. The distance between both points can then be found with a measuring tape. The path found this way must indeed be the shortest, because if there was a shorter one, it would have kept the start and end closer together.

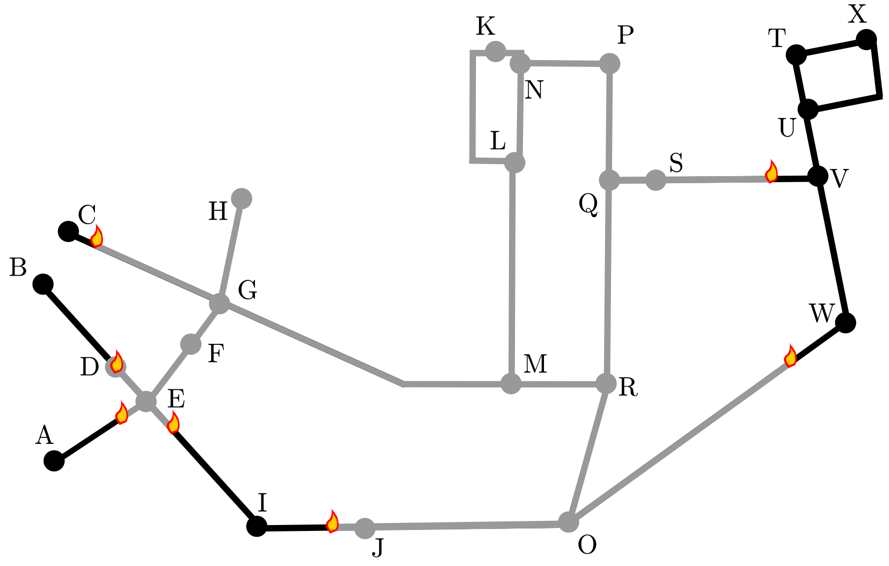
Suppose, for example, I need the shortest path from the cafeteria (M) to the computing center (F). I pick knot M, lifting all other knots off the table. The figure below shows the situation at the moment when knot F hangs in the air for the first time. To make the figure clearer, the knots are pulled apart horizontally a bit. The orange knots are hanging, the numbers on the right indicate the distance to M using the thread length from the first figure.



It is obvious that the shortest path from M to F leads via G. Between L and K, the thread is already sagging with no chance of hanging any straighter. This means there is no shortest path from M using this connection.

I have tried this method successfully for the campus and its surroundings. But I failed miserably with my first trial run for the whole city of Karlsruhe, which produced nothing but a heap of tangled threads. It took me half the night to disentangle them and lay them out on the city map again.

My younger brother drops by the next day. “No problem,” he says, “I will solve the problem with superior technology!” He turns to his chemical kit and soaks the threads in a mysterious liquid. Good grief! He ignites the web at the starting point. Seconds later, the room disappears in a cloud of smoke. This pyromaniac turned the threads into fuses. He explains proudly: “All threads are burning at the same speed. So the time before a knot catches fire is proportional to the distance from the starting point. Besides, the direction from which a knot catches fire contains the same information as the straight threads of my hanging web approach." Great! Unfortunately, he forgot to record the inferno, so we have only ashes left. Even with a video tape I would have to start over with every new starting point. Below, there is a snapshot of the threads after the flames from starting point M have burned part of the way (gray).



I throw my brother out and start to think. I have to get over my fear of abstraction and make the problem clear to my stupid computer. This does have certain advantages: threads that do not exist cannot get tangled up or burn. My professor told me that back in 1959 a certain Mr. Dijkstra developed an algorithm that solves the shortest-path problem in a way that is quite similar to the thread method. Neatly enough, Dijkstra's algorithm can be described in thread terminology.

Dijkstra's Algorithm

Mainly, it is about simulating the thread algorithm. For every knot, a computer implementation must know the threads starting from it and their respective lengths. It also administrates a table d which estimates the distance from the starting point. The distance $d[v]$ is the length of the shortest connection from the starting knot to v using only *hanging* knots. As long as there are no "hanging connections," $d[v]$ is infinite. Therefore, in the beginning, $d[\text{starting knot}] = 0$, and $d[v] = \text{infinite}$ for all other knots.

The pseudocode given here describes the calculation of all knots' distances to the starting knot:

Dijkstra's Algorithm in Thread Terminology

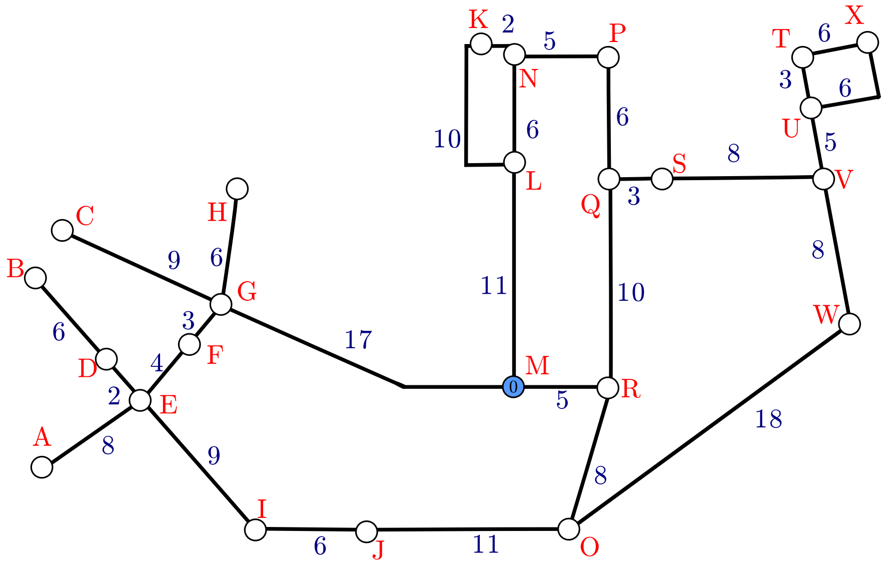
```
1  all knots are waiting, all $d[v]$ are infinite, only $d[\textit{starting knot}] = 0$
2  **while** there are waiting knots **do**
3      $v :=$ the waiting knot with the smallest $d[v]$
4      Turn $v$ hanging
5      **for all** threads from $v$ to a neighbor $u$ of the length $\ell$ **do**
6          **if** $d[v] + \ell < d[u]$, **then** $d[u] := d[v] + \ell$
               //  found shorter path to $u$, leads via $v$
```

How is this algorithm linked to the process of slowly lifting up the starting knot? Each iteration of the while-loop corresponds to the transition of knot v from waiting to hanging. The next knot lifted in turn is the waiting knot v of the smallest value $d[v]$. This value is the height to which we have to lift the starting knot to make v hanging. Since other threads lifted to this height later on cannot decrease it, $d[v]$ is the definitive distance from the starting knot to v.

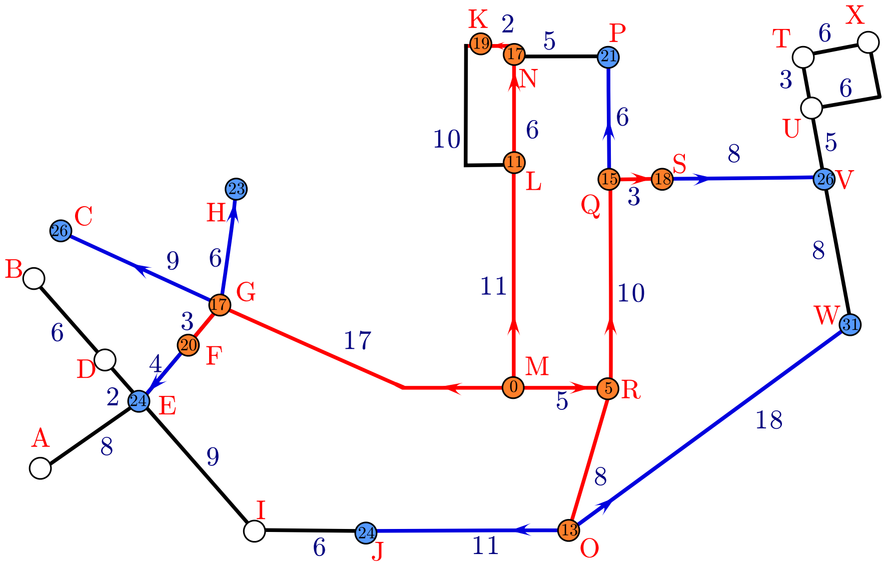
The plus side of Dijkstra's algorithm is that the $d[u]$ values of the other knots can easily be adapted when v becomes hanging: only the threads starting from v have to be considered, which is done by the inner for-loop. A thread between v and a neighboring knot u of the length ℓ builds a connection of the length $d[u] := d[v] + \ell$ from the starting knot to u. If this value is smaller than the last value of $d[u]$, the latter is diminished accordingly. In the end, all knots reachable from the starting knot are hanging and the $d[u]$ values give the lengths of the shortest paths.

In the following, you see some steps in the execution of the algorithm. Hanging knots are orange, blue ones are waiting and the remaining, as of yet unreached knots are white. Inside the circles the current $d[u]$ is given. After the algorithm has finished you can go backwards from the target knot and along the red threads to find the shortest path.

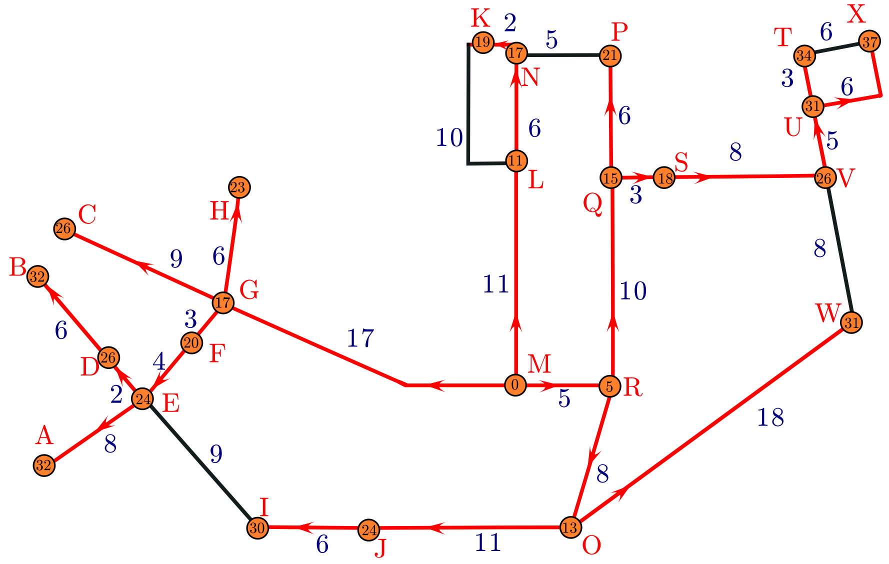
The algorithm starts with the following configuration, when all knots are still lying on the table.



The next figure shows the algorithm's state after ten steps, i.e., the same situation as in the hanging thread web in this chapter's first figure.



The end state looks like this: All knots are hanging in the air, and the shortest paths from M lead along the red threads.



With my computer implementation, I can now calculate distances between knots to my heart's content, without having to clean up ashes or disentangle threads. My anger is subsiding slowly, and maybe my brother will not be permanently banned from my flat after all. However, for real route planning, I must expand Dijkstra's algorithm to make the shortest paths themselves available. Whenever $d[u]$ is set to a new value, the program remembers which knot was responsible for this: knot v that has just been lifted. In the end, the route is reconstructed by going backwards, following the predecessor information from the target to the start knot. The pointers contain the same information as the red threads in the figures.

FAQs and Further Reading

Where can I find a more detailed description of Dijkstra's algorithm? There are many good algorithm textbooks explaining everything there is to know, e.g., K. Mehlhorn, P. Sanders. *Algorithms and Data Structures – The Basic Toolbox.* Springer, 2008.
Who was Dijkstra? Edsger W. Dijkstra (http://de.wikipedia.org/wiki/Edsger_Wybe_Dijkstra) was born in 1930 and died in 2002. Not only did he invent the algorithm described above, he also made substantial contributions in the field of systematic programming and to the modelling of parallel processes. In 1972 he was presented with the Turing Award, the most prestigious

award for computer scientists. His famous article "*A note on two problems in connexion with graphs*" was published in 1959 in the journal *Numerische Mathematik*. The "other" problem he mentions is the calculation of minimum spanning trees. If you leave out the "$d[v]+$" in line 6 of our pseudocode you get the Jarník–Prim algorithm from Chap. 33.

What are threads, etc., in "technicalese"? *Knots* are called *nodes* in computer science; instead of threads we have *edges*. Networks of nodes and edges form *graphs*.

Have I not already come across something like this in this book? In computer science, the search for paths and the related problem of finding circles are very important.

- *Depth search* systematically lists certain paths, which is the basis of many algorithms. See for example Chap. 7 (Depth-First Search) and Chap. 9 (Cycles in Graphs).
- The *Eulerian Circles* in Chap. 28 use each edge exactly once.
- The Travelling Salesman Problem described in Chap. 40 is concerned with a round trip between cities that has to be as short as possible. Determining the travelling time between the cities, however, is a shortest path problem.

How to implement the pseudocode efficiently? We need a data structure that supports the following operations: insert nodes, delete nodes with the shortest distance, and change distance. Since this combination of operations is needed quite often, there is a name for it – *priority queue*. Fast priority queues need time at most *logarithmic* in the number of nodes for any of the operations.

Can we do this even faster? Do we really have to look at the whole Western European road network in order to find the shortest path from Karlsruhe to Barcelona? Common sense says otherwise. Current commercial route planners only look at highways when "far apart" from starting points and destinations, but cannot guarantee not to overlook short cuts. In recent years, however, faster procedures have been developed that guarantee optimal solutions, see, for example, the work of our group `http://algo2.iti.kit.edu/routeplanning.php`.

Is this a route planner for road networks only? The problem is much more common than it seems. For instance, Dijkstra's algorithm does not have to know about a node's geographical position, and is not limited to (spatial) distances, but can also use travelling times as thread lengths. This even works for one-way streets or differing travelling times for the trips from A to B and back, since our algorithm only considers the thread length from start to end. The network can also model many other things, e.g., public transport including departure times, or communication channels in the internet. Even problems that, at first sight, look unrelated to paths, can often be rephrased appropriately. For example, the distance between two strings of characters (genome sequences) in Chap. 31 can be interpreted as a distance in a graph.

Nodes are pairs of letters of the two inputs that are matched. Edges encode the operations *delete*, *insert*, *replace* and *transfer*.
Is it possible to have streets of negative length? This can be quite useful, e.g., it is possible to factor in that my favorite ice cream parlor is in a certain street, so I do not mind detours. However, a round trip of negative length is not allowed, otherwise you could go in circles for as long as you like (eating ice cream) while the path is continuously becoming shorter – the concept of *the* shortest path would not make sense anymore. But even if there are no negative circles, Dijkstra's algorithm could fail. The problem is that via a knot already hanging, a thread of negative length could provide improved routes for other nodes. Dijkstra's algorithm does not handle this case. A better-suited alternative is Bellman and Ford's algorithm, which takes more care to cover all cases but is much slower.