Transportation planning: The shortest route problem

This problem arises in many practical situations. An example is that we have a map and need to find the shortest path from a given starting point to a given destination point. It is useful for transportation companies to optimize their delivery routes, so as to minimize delivery time/distance. The examples for this class of problems are obvious and the need to solve this type of problem arises very frequently in many areas of engineering and science. In typical problems, the nodes represent points, or cities, from which one can travel to other points via roads. A road connecting two nodes is therefore an edge in the graph. Typically, the weights of the edges are either the length of the path it represents, or the cost of traveling on the path, or the time taken to travel on it, etc.

Example: I want to drive from UST to meet my friend, Mr. KS Li, at his hotel in Hung Hom. A simplified map of the roads gives me several alternate routes, from which I must find the shortest route.

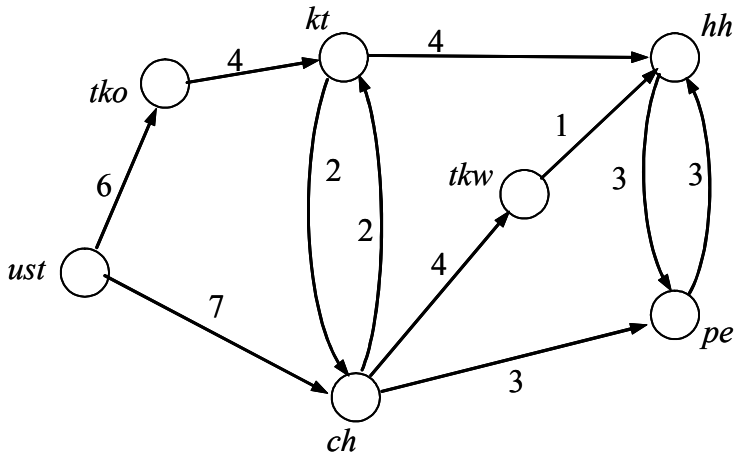


Figure 1. Shortest path example

Of course, one can just list all possibilities in this simple example; we are interested in a general method to solve all such problems. However, once you can model the problem in terms of Graphs, as above, there is simple method that can solve for the best solution. The method is called ***Dijkstra's method***. In fact, many different types of problems that initially appear to be quite different from each other can be modeled as shortest path problems -- and Dijkstra's method works well for most of them.

Notations:
G(E, V) is a ***weighted, directed graph***, such that ***each weight is a positive number***. We shall denote the start node (also called the source) as *s*. So we want to find the minimum total weight path from *s* to some destination node, *z*, in *G*.

Dijkstra's algorithm:
The algorithm uses a simple idea called *relaxation*.

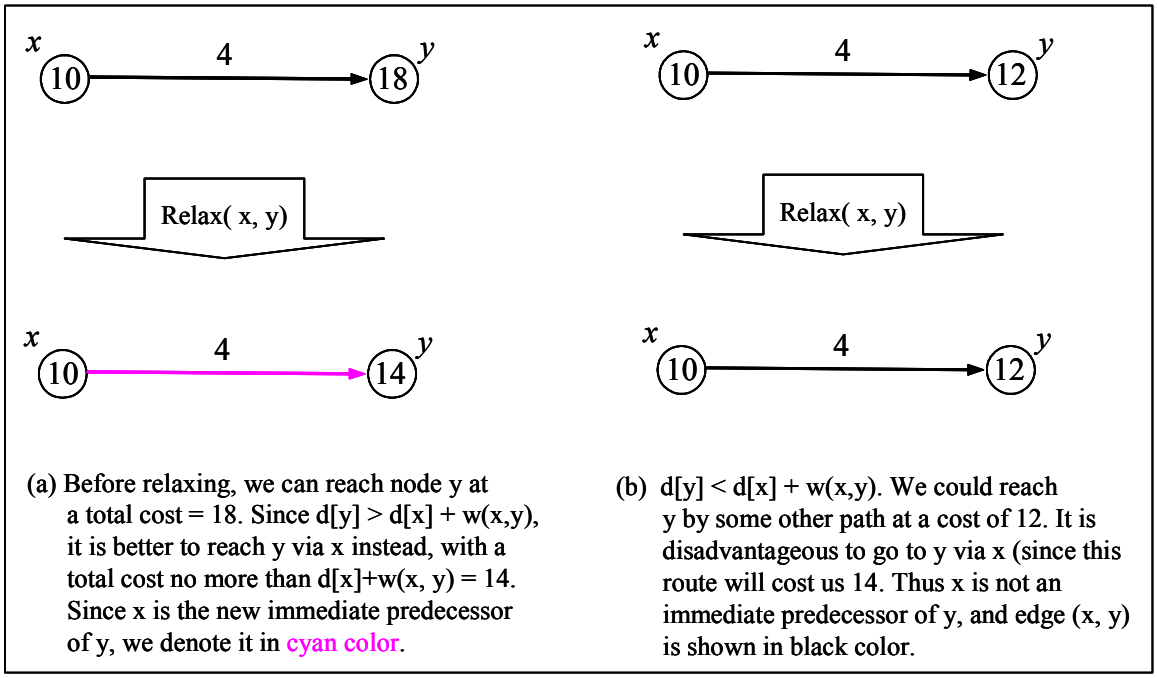
Relaxation:
At each step of the method, we associate with each node *v* a number, *d[v]*, which is the upper bound on the shortest path distance of node *v* from the source *s*.

Thus, in the beginning, we set $d[s] = 0$, and $d[v] = \infty$ for all $v \neq s$. Certainly the distance of the start node from itself is zero; also, the distance of any other node from the start node is less than ∞.

Now assume that there is an edge, (s, u), with cost 10. Then d[u] can be decreased from ∞ to 10. This is because of the following: we can reach from s to u in two ways:
(i) directly; $s \rightarrow u$ in which case it will cost us 10 units to traverse this edge.
(ii) via some other path of the form: $s \rightarrow v_k \rightarrow \ldots \rightarrow v_r \rightarrow u$. In this case, either the total cost of the edges on these routes is less than 10 (say the sum of weights of these edges is 6), then certainly we can reach from s to u with cost $6 < 10$; on the other hand, if the total weights of these edges add up to, say 14; then it is better if we just go directly from $s \rightarrow u$ with weight 10.
In either case, 10 is certainly an upper bound for the total weight to reach u from s. This idea can be extended to any node v:
Assume there is an edge $(u, v) \in E$, with weight $w(u, v)$. Let $d[u]$ and $d[v]$ be the currently known upper bounds on u and v respectively.
Then we can relax the upper bound on v as follows:

if $d[v] > d[u] + w(u, v)$, then set $d[v] = d[u] + w(u, v)$;
otherwise, $d[v]$ is unchanged.

Figure 2 shows examples of the process of relaxation, where the current known upper bound for each node is the number written inside the node.



(a) Before relaxing, we can reach node y at a total cost = 18. Since d[y] > d[x] + w(x,y), it is better to reach y via x instead, with a total cost no more than d[x]+w(x, y) = 14. Since x is the new immediate predecessor of y, we denote it in cyan color.

(b) d[y] < d[x] + w(x,y). We could reach y by some other path at a cost of 12. It is disadvantageous to go to y via x (since this route will cost us 14. Thus x is not an immediate predecessor of y, and edge (x, y) is shown in black color.

Figure 2. Examples of relaxation

When we do a relaxation, we also will keep track of the edge that caused a decrease in the upper bound for v; this is the ***immediate predecessor*** of v in the (current) shortest known path from s to v, and we denote it as $\pi[v]$.

We start with two lists:
V = set of all nodes
S = set of nodes, u_i, for which $d[u_i]$ = *final* shortest path distance from *s* to u_i

Initially, S is empty. At any stage, consider all the nodes u_i, for which the shortest distance from *s* has not yet been determined. We ***pick the one with the least*** $d[u_i]$. For this node, d[u_i] must be the final shortest path distance from source *s* (we shall prove this later) – so we relax every edge going out from this node, and add this node to the set S.

Since at each step, we select the node of S that has the least $d[u_i]$, this step is a ***greedy selection*** rule. The entire method can be written as follows.

INPUTS: a graph, *G(V, E)*, and a start node, *s*.

Step 1. Set d[v_i] = ∞ for each node
Step 2. For the source, s, set d[s] = 0;
Step 3. Make two lists: S = {}; Q = V
Step 4. Find the node, u, in Q with minimum d[u]
Step 5. Remove *u* from *Q*, and add *u* to *S*;
Step 6. For each edge (u, v) going out from u
if (d[v] > d[u] + w(u, v)) then
Set d[v] = d[u] + w(u, v), and
Set u as the immediate predecessor of v
Step 7. If set Q still has more nodes, go to Step 4.
Step 8. Done !

At the end of the procedure, each node should have a finite number, *d[u]* that shows its final, minimum distance from the source node *s*. Further, to find the shortest path, we just need to trace back the immediate predecessors from the destination node back to the start node. Since each node has exactly one immediate predecessor, we are guaranteed to get a unique path back to the source.

The following figures show how the method works for our example problem. In the figures, I denote the nodes in the set *S* by cyan color. Likewise, each immediate predecessor edge is marked in cyan color. Below, I will sometimes write relax a node, which means relax each edge going out of that node.

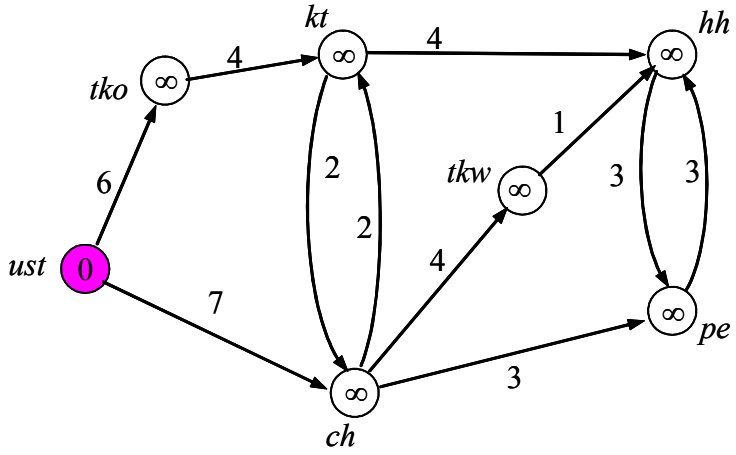


Figure 3. Dijkstra's method: select the node with lowest d[u] (*ust*) and add it to S

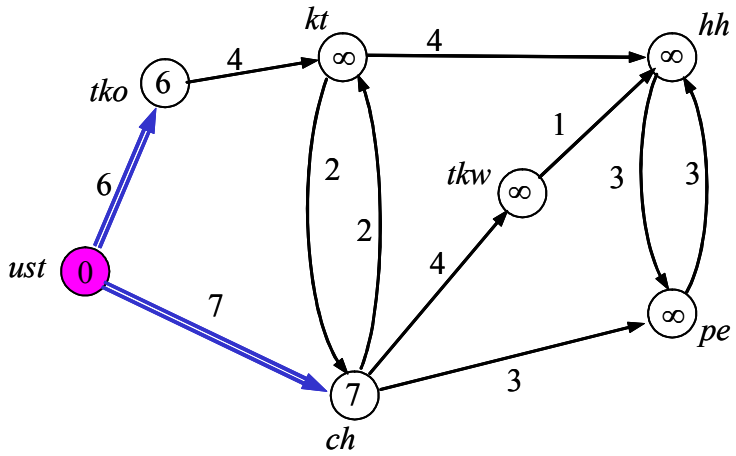


Figure 4. Node ‘ust’ was just added to set S -- relax each edge going out of ‘ust’

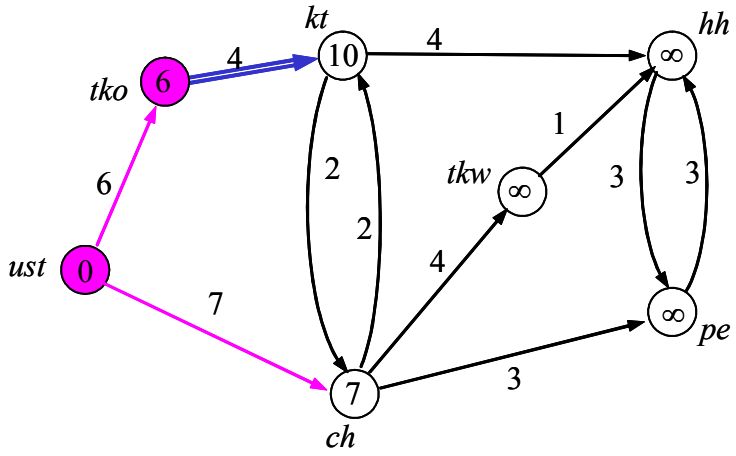


Figure 5. Set the immediate predecessors; Min d[u] node in set Q is ‘tko’, add it to S

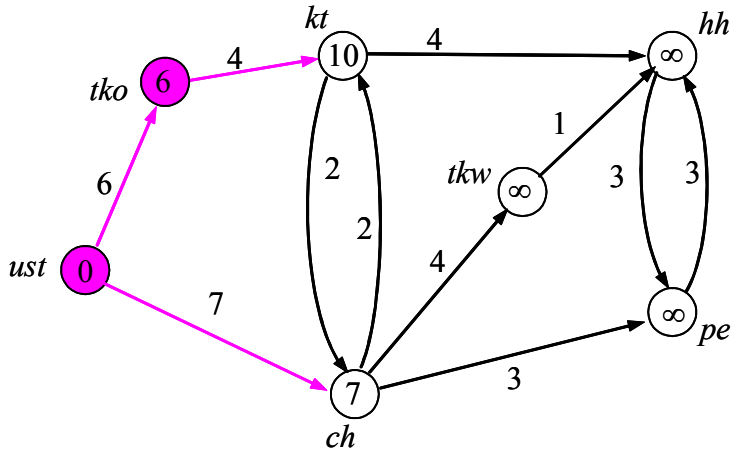


Figure 6. Node ‘tko’ was just added to S, relax it; set the immediate predecessor of node *kt*

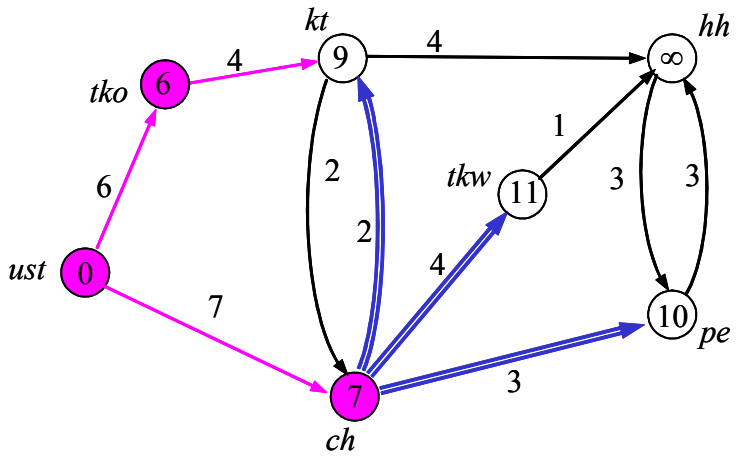


Figure 7. Min d[u] among nodes in Q is now 'ch'; add 'ch' to S; relax it

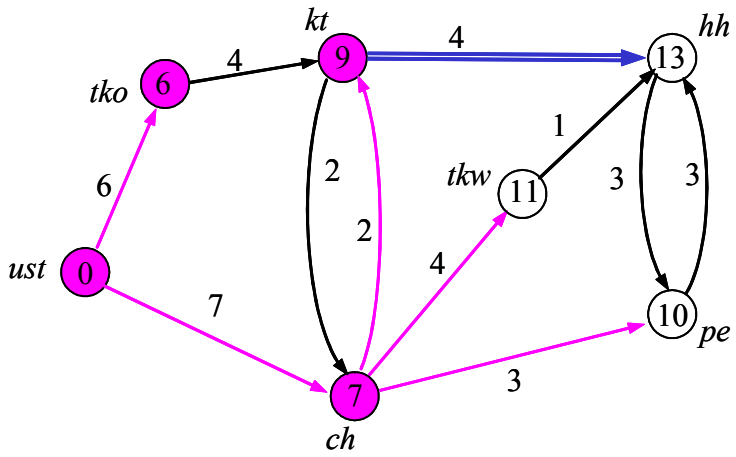


Figure 8. Notice that immediate predecessor of *kt* is changed when we relax *ch*
Select the next node to enter S, which is 'kt', and relax it

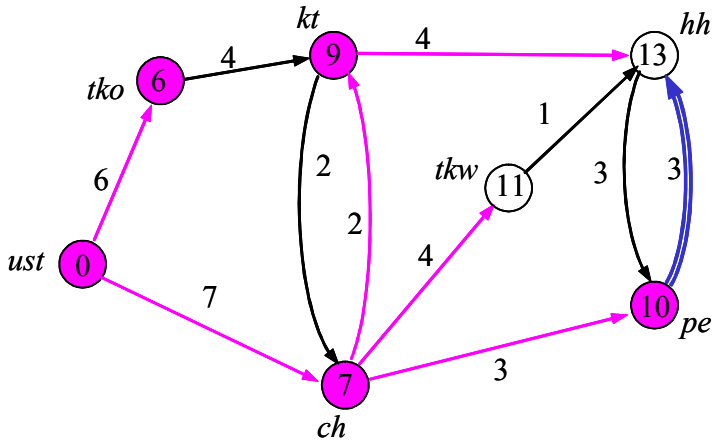


Figure 9. Next node to enter S is *pe*; relaxing *pe* does not change *d[hh]*, so immediate predecessor of *hh* remains unchanged

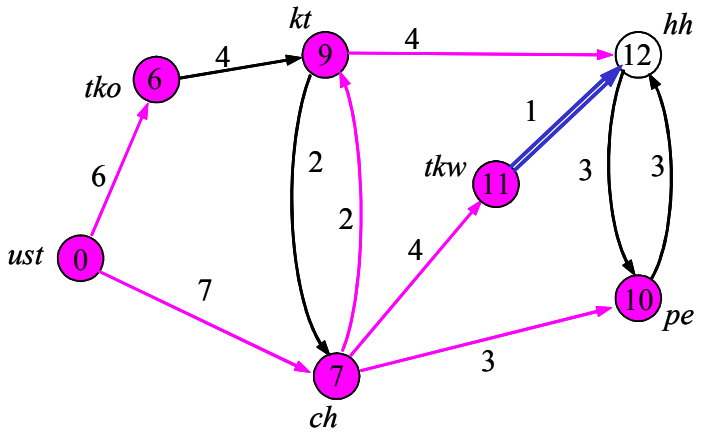


Figure 10. Next node to enter S is *tkw*; relax *tkw*.

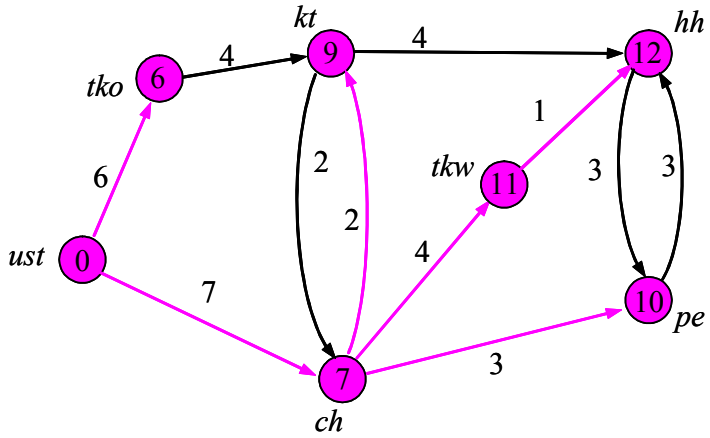


Figure 11. Set immediate predecessor of node *hh*; *hh* enters S; relax *hh*.

After we reach the state shown in Figure 11, Q is empty, so the method terminates. At this stage, d[u] value of any node is its final shortest paths from *ust*. We can trace the immediate predecessors of *hh*: *hh* ← *tkw* ← *ch* ← *ust*; this is the shortest path from *ust* to *hh* in reverse.

Proof of correctness

The algorithm works mainly because to two ideas. We have already seen the proof of correctness of the *relaxation* step. Now we prove that correctness of *greedy selection*.

A few properties of relaxation will be used:

(i) ***d[u] is non-increasing:*** We denote the upper bound on the distance of a node *u* from *s* as d[u]. The value of d[u] can never increase due to relaxation (obvious from the definition of relaxation).

(ii) ***d[u] cannot go below the shortest distance from s to u.*** If the graph is strongly connected, then for each node *u*, there is a shortest distance from *s* to *u*. This is obvious, since there are only a finite number of simple paths from *s* to *u*, and therefore one of them must be the shortest. Denote the length of this shortest distance as *d*[u]*. Certainly, d[u] can never be less than *d*[u]*.

From properties (i) and (ii), we can conclude that once we have found the shortest path to a node, the value, d[u], associated with it will never change.

Notice that in Dijkstra's algorithm, the set *S* consists of exactly such nodes (whose shortest path has been found). Thus all we need to show is that (a) when we select the next node to enter the set S, it's shortest path has indeed been found, and (b) after it enters S, it's shortest path cannot change.

At some intermediate stage, we have a set *S* such that for each node *u* in *S*, *d[u]* is the shortest distance of the node from *s*. Assume that the next selected node, say node v, does not have this property. Since our graph is connected, there must be a shortest path from *s* to *v* (though we have not found it yet). There are two possibilities for this path: it may go through some node(s) that are still not in S, or it may only go through nodes in S before it reaches v (see figure 12)

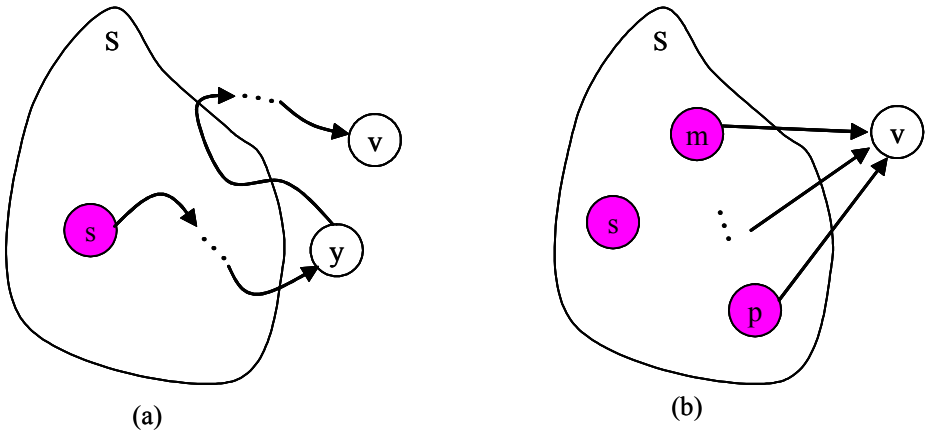


Figure 12. Proof of correctness of Dijkstra's method

In this first case (Figure 12a): since this is the shortest path from s to v, thus it must have been the shortest path from s to y (some node not yet in S). But in that case, *d[y]* < *d[u]* because the edges from *y* to *v* have positive weight, and greedy selection should have selected *y* instead of *v*. So the shortest path from s to v must contain only nodes inside S. But all of these nodes have already been relaxed, and therefore if any of them lead directly to v, the corresponding relaxation must have already set d[v] to the value it can get on the shortest path from s to v. Thus our assumption that v has not yet reached its minimum path length must have been wrong.

We now complete the proof by induction: At the beginning, we add node s, with d[s]=0 to the set S. After that, each selection adds another node whose shortest path has been determined (and will remain unchanged, due to property (ii)). Since each step adds one node to S, and the graph only has finite number of nodes, so the method terminates after a finite number of steps (since there are only a finite number of nodes in V). This completes the proof.

Concluding remarks on Dijkstra's algorithm

1. Notice that Dijkstra's algorithm will work equally well for undirected graphs as for directed graphs. If the original problem consists of undirected graphs, a simple way to

transform it to an equivalent, directed graph is to split each edge (u, v) into two directed edges, (u, v) and (v, u), each with weight equal to the weight of the original edge (u, v).

2. Shortest route problems are very important for many transportation and logistics companies -- for example FedEx and DHL (who deliver packages by trucks and spend millions of dollars on petrol), logistics companies who deliver large shipments of goods from factories to shipping ports, etc.

3. Since I have a little space left in the page, here is a picture of the Prof Dijkstra (he died in 2002), that I got from the web.