Greedy algorithms on graphs, 1

1 Shortest paths in digraphs

A *weighted digraph* is a digraph where each edge e has a weight $w(e)$ associated with it. The weight is a numerical value. An example of a weighted graph is presented in Figure 1.

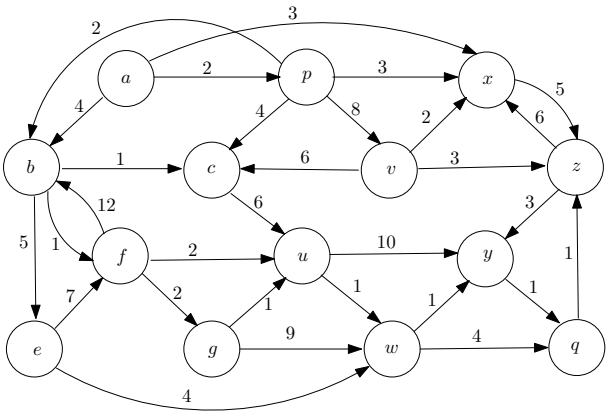

Fig. 1. An example of a weighted graph G

Let G be a weighted digraph. We always assume that the weights are non-negative.

Definition 1. *Let $v_0, v_1, \ldots, v_n$ be a path P.* The weight (or the cost) of this path P, *denoted by $w(P)$, is the sum of the weights of its edges, that is:*

$$w(P) = \sum_{i=0}^{n-1} w(e_i),$$

where e_i is the edge from v_i to v_{i+1}.

The shortest path-distance from a vertex u to a vertex v is the minimum weight among the weights of all paths from u to v. We denote this shortest path-distance by $\delta(u, v)$. Thus,

$$\delta(u, v) = \min \{w(P) \mid P \text{ is a path from } u \text{ to } v \text{ in } G\}.$$

A *shortest path from u to v* is then any path P such that the weight of P is equal to the shortest path-distance $\delta(u, v)$ from u to v.

Consider the weighted digraph in Figure 1. There are infinitely many paths from p to u. Consider, as an example, four of them: the first path is p, c, u, the second path is p, b, c, u, the third path is p, b, e, f, u, and the fourth path is p, v, c, u. The weights of these paths are 10, 9, 16, and 20 respectively. The path p, b, c, u is the shortest path among these four.

Exercise 1. Find shortest paths in the digraph in Figure 1 from e to x, and from a to y.

Exercise 2. Prove that there is always a shortest path from u to v in a digraph any time when there is a path from u to v.

Exercise 3. Assume that negative weights are allowed in digraph G. Give an example where the shortest path from u to v might not exist even if there is a path from u to v.

If there is no path from u to v then we say that the path-distance from u to v is infinite and write this by $\delta(u, v) = \infty$. If there is a path from u to v then the shortest path-distance $\delta(u, v)$ is a real number, and hence we write $\delta(u, v) < \infty$.

Let P be a path $v_0, \ldots, v_n$ from u to v in a weighted digraph G. A *subpath* of P is any continuous segment of P. Thus, all subpaths of P are of the form $v_i, v_{i+1}, \ldots, v_j$, where $i \leq j \leq n$. Note that subpaths of P might have different start and end vertices than those of P. It is easy to see that a path of length n has at most $n \cdot (n+1)/2$ subpaths.

Observation 1 *A subpath of a shortest path is again a shortest path.*

Observation 2 (Triangle inequality) *For all vertices $x, y, z \in V$ of the weighted digraph G, the following inequality is true: $\delta(x, z) \leq \delta(x, y) + \delta(y, z)$.*

Exercise 4. Explain why the two observations above are true.

2 Formulation of the problem

The shortest path problem is the following. Given a weighted digraph G and its source vertex s, find the shorest path-distances $\delta(s, v)$ for all vertices $v \in V$. Here are two examples of problems that arise in real life and that can be reduced to the shortest path problem.

A flight agenda example. A travel agency is designing software for making flight plans for its clients. The agency has access to a database of all airports, flights, their costs, and their durations. The agency wants to support both clients that are looking for the cheapest airfares, as well as those that are looking for the fastest way to get from one location to another.

Routing example. Computer networks, such as the Internet, can also be represented as weighted digraphs. The vertices represent routers, and the weights of edges represent the bandwidth of the connection between routers. The bandwidth is the transmission capacity of a connection. A common problem in computer networks is to find a path, given two vertices, with the highest bandwidth. This problem can be transformed into the shortest path problem with a bit of manipulation on the weights.

3 Dijkstra's algorithm

In 1959, Edsger Dijkstra designed an efficient algorithm that solved the shortest path problem. In this section, we explain the algorithm. In the next section, we present an example. Before we present the algorithm, we informally describe several of its ingredients. The inputs to the algorithm are a weighted digraph G and a source vertex s. The algorithm finds the shortest path-distance between s and every vertex in G. The algorithm goes through iterations.

– At each iteration, the algorithm computes an *estimate* of the path-distances, denoted $d(v)$, from s to all vertices v. If the algorithm has not yet seen a vertex v, then the estimate is set to ∞. Thus, initially $d(s) = 0$, and $d(v) = \infty$ for all other vertices v.
– The algorithm maintains a set F of all vertices whose shortest path-distances from s have already been computed. This is the explored part of the digraph. Once a vertex v appears in F, its current path-distance estimate $d(v)$ will never be changed and stays equal to $\delta(s, v)$.
– At each iteration, the algorithm puts a new vertex v into F. When selecting v, the algorithm picks the one with the smallest path-distance estimate $d(v)$ among the vertices not in F. The intention is that $d(v)$ is now the shortest path-distance, and hence v must be put in F.
– Once a vertex v is added to F, the algorithm updates the current path-distance estimates $d(u)$ for all of v's adjacent vertices u. This updating process is called *relaxation*.

Here is the $Dijkstra(G, s)$-algorithm.

Algorithm 1 $Dijkstra(G, s)$ algorithm

1. Initialise $d(s) = 0$, $F = \{s\}$ and $R = V \setminus F$, and and $d(v) = \infty$ for all $v \neq s$.
2. For each v such that (s, v) is an edge set $d(v) = w(s, v)$.
3. *While* the set R contains a vertex z such that $d(z) < \infty$ do:
   (a) Select the first $v \in R$ with the smallest path-distance estimate $d(v)$.
   (b) Extend F to $F \cup \{v\}$.
   (c) *Relaxation process*: Relax each edge (v, u) outgoing from v as follows. If $d(u) > d(v) + w(v, u)$ then set $d(u) = d(v) + w(v, u)$.
   (d) Remove v from R.

The relaxation process in $Dijkstra$-algorithm, once v is put into F, updates the current path-distances $d(u)$ of all vertices u connected to v via outgoing edges (v, u). The *relaxation* concept is derived from the analogy between the current path-distance $d(u)$ from s to u and the length of a helical tension spring. Initially the current path-distance $d(u)$ is an overestimate. This is similar to the helical tension spring being stretched. As the current path-distance becomes shorter, the helical tension spring becomes more relaxed. Once a shortest path is found, the spring becomes relaxed and assumes its natural resting length.

It is easy to construct the path from s to u using the algorithm, lets denote the path by P_u, corresponding to the distance (that is, $d(u)$) found by Dijkstras Algorithm. As a node v is added to F, we *record* the edge (a, v), where a was already in F, that was responsible for v to appear in F. Thus, the path P_u is obtained by finding the path P_a and appending the edge (a, v) at the end of the path P_a. In other words, to construct P_u, we follow the edge (a, v) recorded for v in the reverse direction to a; then follow the edge recorded for a in the reverse direction from a to its predecessor; and so on until s is reached. This gives a sequence of vertices from v to s which when reversed gives as the desired path P_u.

As an example, the way the algorithm runs, consider the graph G presented in Figure 1. The source vertex is a. The Table 18.1 shows the steps of the algorithm. The first column of the table represents the number of the iteration. The second enumerates the elements of F. The third lists the d-values of all vertices in F. The last column presents the d-values of vertices not in F. For the d-values of vertices v that are not in the table, we have $d(v) = \infty$

#	F	*d-values for vertices in* F	*d-values for vertices not in* F
0	a	$d(a) = 0$	$d(p) = 2, d(x) = 3, d(b) = 4$
1	a, p	$d(a) = 0, d(p) = 2$	$d(x) = 3, d(b) = 4, d(c) = 6,$ $d(v) = 10$
2	a, p, x	$d(a) = 0, d(p) = 2, d(x) = 3$	$d(b) = 4, d(c) = 6, d(z) = 8,$ $d(v) = 10$
3	a, p, x, b	$d(a) = 0, d(p) = 2, d(x) = 3,$ $d(b) = 4$	$d(c) = 5, d(f) = 5, d(z) = 8,$ $d(e) = 9, d(v) = 10$
4	a, p, x, b, c	$d(a) = 0, d(p) = 2, d(x) = 3$ $d(b) = 4, d(c) = 5,$	$d(f) = 5, d(z) = 8,$ $d(e) = 9, d(v) = 10, d(u) = 11$
5	$a, p, x, b, c,$ f	$d(a) = 0, d(p) = 2, d(x) = 3$ $d(b) = 4, d(c) = 5,\ d(f) = 5$	$d(u) = 7, d(g) = 7, d(z) = 8,$ $d(e) = 9, d(v) = 10$
6	$a, p, x, b, c,$ f, u	$d(a) = 0, d(p) = 2, d(x) = 3$ $d(b) = 4, d(c) = 5,\ d(f) = 5$ $d(u) = 7$	$d(g) = 7, d(z) = 8, d(w) = 8,$ $d(e) = 9, d(v) = 10, d(y) = 17$
7	$a, p, x, b, c,$ f, u, g	$d(a) = 0, d(p) = 2, d(x) = 3$ $d(b) = 4, d(c) = 5,\ d(f) = 5$ $d(u) = 7, d(g) = 7$	$d(z) = 8, d(w) = 8,$ $d(e) = 9, d(v) = 10, d(y) = 17$
8	$a, p, x, b, c,$ f, u, g, z	$d(a) = 0, d(p) = 2, d(x) = 3$ $d(b) = 4, d(c) = 5,\ d(f) = 5$ $d(u) = 7, d(g) = 7, d(z) = 8$	$d(w) = 8, d(e) = 9,$ $d(v) = 10, d(y) = 11$
9	$a, p, x, b, c,$ f, u, g, z, w	$d(a) = 0, d(p) = 2, d(x) = 3$ $d(b) = 4, d(c) = 5,\ d(f) = 5$ $d(u) = 7, d(g) = 7, d(z) = 8,$ $d(w) = 8$	$d(y) = 9, d(e) = 9,$ $d(v) = 10, d(q) = 12$
10	$a, p, x, b, c,$ $f, u, g, z, w,$ y	$d(a) = 0, d(p) = 2, d(x) = 3$ $d(b) = 4, d(c) = 5,\ d(f) = 5$ $d(u) = 7, d(g) = 7, d(z) = 8,$ $d(w) = 8, d(y) = 9$	$d(e) = 9,$ $d(v) = 10, d(q) = 10$
11	$a, p, x, b, c,$ $f, u, g, z, w,$ y, e	$d(a) = 0, d(p) = 2, d(x) = 3$ $d(b) = 4, d(c) = 5,\ d(f) = 5$ $d(u) = 7, d(g) = 7, d(z) = 8,$ $d(w) = 8, d(y) = 9, d(e) = 9$	$d(v) = 10, d(q) = 10$
12	$a, p, x, b, c,$ $f, u, g, z, w,$ y, e, v	$d(a) = 0, d(p) = 2, d(x) = 3$ $d(b) = 4, d(c) = 5,\ d(f) = 5$ $d(u) = 7, d(g) = 7, d(z) = 8,$ $d(w) = 8, d(y) = 9, d(e) = 9,$ $d(v) = 10$	$d(q) = 10$
13	$a, p, x, b, c,$ $f, u, g, z, w,$ y, e, v, q	$d(a) = 0, d(p) = 2, d(x) = 3$ $d(b) = 4, d(c) = 5,\ d(f) = 5$ $d(u) = 7, d(g) = 7, d(z) = 8,$ $d(w) = 8, d(y) = 9, d(e) = 9,$ $d(v) = 10, d(q) = 10$	

Table 1. Iteration process of *Dijkstra*-algorithm

.

Exercise 5. Run the $Dijkstra(G, s)$–algorithm on graph G in Figure 1 starting with the following values for s: v, p, b and q.

4 Correctness of Dijkstra's algorithm

We will be using the notation P_u for the shortest path from s to u produced by the algorithm described at the end of Section 3. The main ingredient of showing the correctness consist of proving the following fact:

Fact 1 *Consider the set F at any point in the algorithms execution. For each $u \in F$, the path P_u is a shortest path from s to u.*

Proof. Before the algorithm starts executing its while loop, we have $F = \{s\}$. For this value of F, the fact is true. Hence, it suffices to prove the following. If the fact is true for F before any iteration of the while loop, then the fact remains true for F after the iteration.

Thus, assume that the set F_1 is the value of F before an iteration of the while loop. Let the set F_2 be the value of F after the iteration. We assume that for all $v \in F_1$, the path P_v is a shortest path from s to v. We need to prove that for all $u \in F_2$, the path P_u is a shortest path from s to u. This will prove the fact.

Note that $F_2 = F_1 \cup \{u\}$, where u is the vertex added to F_1 during the execution of the while loop. It suffices to show that P_u is the shortest path (as for all other v in F_2 belong to F_1, and we know that for these v, the paths P_v are shortest paths). Now consider any path P that goes from s to u. Our goal is to compare the weight of P with the weight weight of P_u. The path P, since it reaches u, must go out of F somewhere. Let y be the first vertex on P that is not in F, and let x be the vertex on P just before y. So x is in F. Now the gist of our argument is the following:

The path P cannot be shorter than P_u because it is already at least as long as P_v by the time it has left the set F.

Here is more formal argument. During the iteration that produces F_2, the algorithm considers adding y to F_1 through the edge (x, y) but rejects this option in favour of adding u to F_1 via the edge (v, u). This implies that

$$w(P_u) = w(P_v) + w(v, u) = d_v + w(v, u) \leq w(P_x) + w(x, y) \leq w(P).$$

Hence, the full path P is at least as long as P_u as well.

Exercise 6. In the $Dijkstra(G, s)$–algorithm, explain why the values $d(v)$ never increase.

Exercise 7. Extend the $Dijkstra(G, s)$–algorithm so that it outputs the distance $\delta(s, v)$ together with a shortest path from s to v.

5 Running time analysis: an easy upper bound

Here we give a high level analysis of the running time Dijkstra's algorithm. This is quite easy. Assume that the input weighted digraph has n vertices. The while loop of the algorithm makes n iterations since each iteration adds a vertex to F, and F can grow at most n times. In the iteration process, we need to select $v \in R$ with the minimal value $d(v)$; this takes $O(n)$ time. Then we need to perform the relaxation operation; this takes $O(m)$ time as we need to go through the edges of the graph. Hence, overall running time can be bounded by $O(n \cdot (n+m))$. This is not a bad running time. But with a right data structures we can do much better. The next two sections explain how we do that.

6 Priority queue data structure

A priority queue data structure maintains a set S of elements where each element v in S has a value $key(v)$. No two keys represent the same element but two elements might have the same key. We say element u *has higher priority than* v if $key(u) < key(v)$. So, elements with smaller keys have higher priorities. Priority queue data structures support three base operations: *insert*, *delete*, and *select*. The *insert* operation adds elements to S, the *delete* operation removes the element from S that has the highest priority, the *select* operation finds an element in S with the highest priority.

What happens if we want to implement these base operations on data structures that we already know? Consider, for instance, the list data structures. Say, we have linked list data structure together with a pointer pointing to an element with the least key in the list. This data structure makes insertion operation easy. This data structure also makes it easy to delete the element that has the highest priority. However, finding the element with the highest priority becomes $O(n)$ operation. Another possible natural data structure that we can use is to store the keys in a sorted array. Then, certainly, we can find in $O(log(n))$ time the position where the key of an element v can be inserted. However, after that we need to move all the array positions greater than $key(v)$. This might take $O(n)$ time. The same type of issues arise when we maintain the elements as sorted doubly linked list. These examples show that in each of these data structures some base queue operations are quite efficient but others take $O(n)$ time. A natural question is if one can invent an implementation of the queue data structure in which *all* base operations can run in time $O(log(n))$.

7 Defining Heap

The heap data structure implements priority queue data structure such that all base operations run in time $O(log(n))$. The instances of the data structure are heaps. A *heap* is a complete binary tree. The tree has a root, and each node of the tree has up to two children nodes: the left child and the right child. The defining property of the heap is that the key of any element is at least as large as the key of the element at its parent node. This is called the *heap order*. More precisely, the heap order tells us the following. For all elements u at node i of the tree we have $key(u) \leq key(v)$ for the element v at i's parent.

There are several ways to implement heap trees. For instance, they can be implemented by using pointers. Namely, every node of the heap tree keeps the element from S that it stores, keeps the key, and has three pointers pointing to the parent, and the two children of the node.

Another way to represent heap trees, that we focus on, is through the arrays. For this we assume that S will always contain not more than B elements, where B is a big number (which will be the size of the array). Using arrays, we can forget about the pointers. This is done as follows. We create an array H of length B. The first element of the array $H(1)$ represents the element at the root of the tree. The elements $H(2)$ and $H(3)$ represent the left and the right children of the root. At position i of the array H, the children of (the element in) the position are located at positions $2i$ and $2i+1$ in the array H. In other words, we have

$$LefChild(i) = 2i,\ RighChild(i) = 2i+1,\ Parent(2i) = i,\ \text{ and }\ Parent(2i+1) = i.$$

In H we might identify positions i with the element $H(i)$ if no confusion arises. For instance, $Parent(i)$ formally refers to the position in H but it might also refer to the element stored at the parents position. So, it makes sense to write $Key(Parent(i))$ depending on the context.

Assume now that S has n elements with $n < B$. The array H is implemented in such a way that these n elements of S are all in the first n positions of H. We use $Length(H)$ to represent the number of elements in S. So, $Length(H) = n < B$.

Note that the tree represented through the array H is complete. In particular: (1) The tree has the root, (2) every internal node, apart from at most one, has the left and the right child, (3) if a node has a right child, then it has the left child, (4) the height of the tree is $\log(n)$, where n is the number of elements in S.

Now we would like to discuss the implementation of the base heap operations: select, inset, and delete. The element of the highest priority is located at the root of the heap tree. Selecting the minimal elements is a constant time operation.

How do we insert an element? Say, the new element is v. We put v at position $n+1$ of H. This corresponds to creating a new leaf. If $Key(Parent(n+1)) \leq key(v)$ then we have a heap tree. Otherwise, the new array has a small "damaged part" that prevents it from being a heap tree. To repair the damage we swap the elements in these two positions. Now the local damage is repaired. Let i be the new position of v. Consider the parent j of i. Say the node j has element w. If $key(w) \leq key(v)$ then we have a heap tree; otherwise, we repeat the process by moving v upwards until the array represents a heap tree. Call the operation of moving new element upwards, $HeapifyUp$ process:

Algorithm 2 $HeapifyUp(H, i)$

If $i > 1$ then let $j = Parent(i)$
− If $key(H(i)) < key(H(j))$ then swap the array entries $H(i)$ and $H(j)$
−− $HeapifyUp\ (H, j\ )$
− Endif
Endif

The swap is a constant time operation. Since the height of the tree is $\log(n)$, the running time of the insert operation, via $HeapifyUp$ process, is $O(\log(n)$. It is also not so hard to see that the result of the insert operation, via $HeapifyUp$ process, produces a heap tree.

Now we discuss the delete operation. Suppose that we remove v from H at position i. So, S now has n–1 elements and there is a "hole" in H at position i. We take the element w at position n and put w in position i of H. Now, just like we had in *insert* operation our heap H has a small "damage" that needs to be repaired. If the $key(w)$ is smaller than the key of $Parent(i)$ node then we use $HeapifyUp$ operation, and move w up the heap tree; This repairs the damage and produces a heap tree again in $O(\log(n)$ time as we saw above. On the other hand, if $key(w)$ is too big then we need to move w down by swapping it with one of the children of position i (if need be). Below is a description of this process that we call $HeapifyDown$:

Algorithm 3 $HeapifyDown(H, i)$

If $2i > n$ then stop with H unchanged
− Else if $2i < n$ then set $L = 2i$, $R = 2i + 1$, and $j = \min\{key(H(L), key(H(R))\}$
− Else if $2i = n$ then let $j = key(H(2i))$
Endif
If $key(H(j)) < key(H(i))$ then swap the array entries $H(i)$ and $H(j)$
− $HeapifyDown\ (H, j)$
Endif.

Note that the swapping is a constant time operation. Since the height of the tree is $\log(n)$, the running time of the delete operation, via $HeapifyDown$ process, is $O(\log(n))$. The result of the delete operation, via $HeapifyDown$ process, produces a heap tree.

8 Analysis of running time for Dijkstra's algorithm

The weighted digraph is given just like digraphs, in adjacency list representation. Given a vertex u, we have two linked lists associated with u, the list of vertices for outgoing edges and the list of vertices for in-going edges. If v appears in the list of out-going vertices, then together with v we also store the weight $w(e)$ of the edge $e = (u, v)$. The same is for ingoing edges.

The use of heap data structure lies in maintaining the set R (in Dijkstra's algorithm) of vertices v such that v is not in F and $d(v) < \infty$. So, the key of each element v from R is the value $d(v)$. The root of the heap tree representing R is the vertex v with the smallest value $d(v)$ (so, v has the highest priority to enter F). The select operation of the heap data structure takes care of the vertex with the highest priority. Once, v is put into F, the algorithm needs perform the relaxation step. For each outgoing edge (v, u), the algorithm needs to update the value $d(u)$ if u is not in F and if $d(u) > d(v) + w(v, u)$. If this happens, then the key $d(u)$ decreases. This means that the heap representing R changes. These changes can be taken care by the $HeapifyUp$ process described in the heap data structure. We call this $ChangeKey$ operation.

Since $ChangeKey$ operation uses the $HeapifyUp$ process, the operation runs in $O(\log(n))$ time. It is important to note that $ChangeKey$ operation is applied at most once per edge. So, $ChangeKey$ operation is used at most m times.

Thus, we need to extract the elements from R with the smallest key value $d(v)$ at most n times. Each such extraction results in $O(\log(n)))$ constant-time operations. All of these imply that using the heap data structure Dijkstra's algorithm can implemented so that the running time of the implementation is $O(m \cdot log(n))$. This is much better than $O(n \cdot (n+m))$ time derived through a high level analysis of the algorithm. A pseudocode of the algorithm is below:

Algorithm 4 Implementation of $Dijkstra(G, s)$

```
INPUT  A weighted graph G = (V, E, w), vertex s
OUTPUT  d(v) of all vertices v
  d(s) ← 0
  Initialize a set F ← {s}
  Initialize a priority queue R containing s with key(s) = 0          ▷ Use a heap for R
  for each vertex u that is not s do
      Set d(u) ← ∞.
      R.insert(u) with key(u) = ∞
  end for
  while R is not empty do
      Set u ← R.delete()
      Add u to F
      for (u, v) ∈ E where v ∉ F do                                   ▷ Relaxation
          if d(u) + w(u, v) < d(v) then
              d(v) ← d(u) + w(u, v)
              Call ChangeKey operation to set key(v) to d(v) in R
          end if
      end for
  end while
```

Exercise 8. Write a program implementing Dijkstra's algorithm using the heap data structure.