Data Structures and Algorithms
FR 6.2 Informatik

Sanders, Telikepalli WS 03/04

http://www.mpi-sb.mpg.de/~sanders/courses/algdat03/index.html

---

Solutions to assignment 2

Exercise 1
Suppose you want to solve the SSSP on a graph with positive edge weights. Let

$$r = \frac{\max_{e\in E} c(e)}{\min_{e\in E} c(e)}$$

Develop an algorithm that runs in time $\mathcal{O}(m + nr)$ for such inputs. Hint: Use a bucket queue with buckets of width $\min_{e\in E} c(e)$. Show that *all* vertices in the smallest nonempty bucket have $d(v) = \mu(s, v)$. Describe your algorithm design and prove the claimed runtime bound.

Solution: We use Dijkstra's Algorithm (as presented in the lecture) with a bucket queue with $r + 1$ buckets of width $w = \min_{e\in E} c(e)$ each, where $r = \lceil \frac{\max_{e\in E} c(e)}{\min_{e\in E} c(e)} \rceil$.

The bucket $i, i \in \{0, \ldots, r\}$, contains nodes v with $d(v) \in [z\cdot(r+1)w+i\cdot w, z\cdot(r+1)w+(i+1)\cdot w)$, where $z \in \mathbb{N}$.

$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$[(r+1)w, (r+2)w)$	$\cdots$	$\cdots$	$\cdots$	$[(2r+1)w, (2r+2)w)$
$[0, w)$	$[w, 2w)$	$[2w, 3w)$	$\cdots$	$[rw, (r+1)w)$
B[0]	**B[1]**	**B[2]**	$\cdots$	**B[r]**

The *insert* operation can be done in constant time: $\mathrm{B}[\lfloor (x - \lfloor \frac{x}{(r+1)w} \rfloor \cdot (r+1)w)/w \rfloor]$.insert(x)

The *decreaseKey* operation can be done in constant time, too. We can store for each node a pointer to the entry in the bucket queue so that we can remove it in $O(1)$.

In order to perform the *deleteMin* operation, we have to find the next nonempty bucket. Eventually, after *all deleteMin* operations have been performed, this search terminates at the value $(n-1)\cdot \max_{e\in E} c(e)$ (or earlier) so that at most $((n-1)\cdot \max_{e\in E} c(e))/w \le (n-1)r$ buckets are searched altogether. As we will show that it is sufficient to delete any element from the smallest nonempty bucket (i.e., not necessarily the smallest one), we can perform a *popFront* operation in constant time after we have found the appropriate bucket so that all *deleteMin* operations take $O(nr)$ time. This leads to the claimed runtime bound $O(m + nr)$.

Now we still have to prove that *all* vertices in the smallest nonempty bucket have $d(v) = \mu(s, v)$. Assume that there is a vertex v in the smallest nonempty bucket so that $\mu(s, v) < d(v)$. Then there has to be a shortest path $p = \langle s = u_1, u_2, \ldots, u_k, v\rangle$, where not all nodes $u_i, i \in \{1, \ldots, k\}$, have been visited yet. Let be $j = \min\{i \mid u_i \text{ has not been visited yet}\}$. Since all nodes $u_1, u_2, \ldots, u_{j-1}$ have been visited, we have $d(u_j) = \mu(s, u_j)$ as $p' = \langle s = u_1, u_2, \ldots, u_{j-1}, u_j\rangle$ is a shortest path from s to u_j. When the outgoing edges of u_{j-1} were relaxed, the node u_j was added to the priority queue (if it had not been there before). Furthermore, our assumption $(\mu(s, v) < d(v))$ leads to $\mu(s, u_j) + \mu(u_j, v) < d(v)$ because u_j lies on p. As the shortest path from u_j to v cannot be shorter than w, we obtain $d(u_j) + w < d(v)$. Hence, u_j has to be in a smaller bucket than v. This leads to a contradiction as v is in the smallest nonempty bucket.

This proof works because at any time a bucket contains only nodes v with $a \le d(v) < a + w$ for one specific a. If one node v is deleted from bucket i, the new tentative distance $d(u)$ to another node u cannot be greater than $\mu(s, v) + \max_{e\in E} c(e)$ so that u is inserted in one of the buckets $i + 1, i + 2, \ldots, i + r \pmod{(r + 1)}$ as $\max_{e\in E} c(e)/w \le r$. Hence, there is no overlapping.

Exercise 2
Design a family of graphs and a non-negative cost function such that the relaxation of $m-(n-1)$ edges causes a *decreaseKey* operation in Dijkstra algorithm.

Solution: Description of the graph family: Node set $V = \{ 1, \ldots, n \}$, first node is the source. Edge set $E = \bigcup_{i=1}^{n} E_i$, where E_i is set of edges going out of node i, $E_i = \{ (i,j) \mid i+1 \leq j \leq n \}$. The weight of the edge from node i to node j is defined as $c((i,j)) = j - i - 1$. An example of a graph from this family for $n = 8$ is shown on the Figure 1.

The $n-1$ egdes which start from the source node are examined first by the Dijkstra algorithm. Since their destination nodes $\{2, \ldots, n\}$ were not yet in the priority queue all these nodes are *inserted* with their initial tentative distances $\{0, \ldots, n-2\}$. Processing the rest $m-(n-1)$ edges always causes the *decreaseKey* operation for each edge and decreases the tentative distance of the destination node by one.

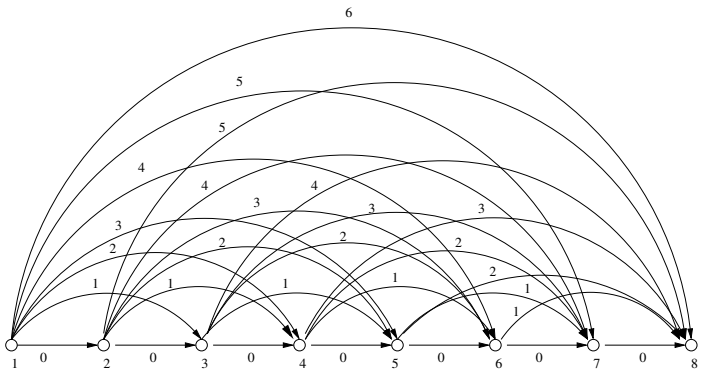

Figure 1: Example graph for $n = 8$

The presented family of graphs is very dense: $m = n(n+1)/2$. To construct sparser graphs we can remove edges from the set E_{n-1} and when it is empty, we continue to remove edges from the set E_{n-2}, then E_{n-3} and so fort, until we achieve desired m. This order of removal keeps the required number of *decreaseKey* operation equal to $m-(n-1)$. This way we obtain graphs with variable density: $n \leq m < n(n+1)/2$.

Exercise 3
Running in Saarbrücken

To get in shape, you have decided to start running to the university. You want a route that goes entirely uphill and then downhill so that you can work up a sweat going uphill and then get a nice breeze at the end of your run as you run faster downhill. Your run will start at home and end at the university and you have a map detailing the roads with m road segments (any existing road between two intersections) and n intersections. Each road segment has a positive length, and each intersection has a distinct elevation.

1. Assuming that every road segment is either uphill or downhill, give an efficient algorithm to find the shortest route that meets you specifications.
2. Give an efficient algorithm to solve the problem if some roads may be level (i.e., both intersections at the end of the road segments are at the same elevation) and therefore can be taken at any point.

Solution:

Part 1

Let $G = (V, E)$ be the graph that we construct given the road segments and their intersections. We have costs on the edges that represent the length of the corresponding road segment. We

also have a value on every node that represents the elevation of that node. Let s be the node representing the house, call it source, and let t be the node representing the university, call it sink.

The idea is that we perform two separate shortest path computations. One from the source to the rest of the nodes following only uphill edges, i.e, edges whose difference of elevations of the target node minus the source node is positive. One more, from the sink to the rest of the nodes following uphill edges. On every node we store these two shortest path distances and we choose the path that corresponds to the node that has the minimum sum of this values.

The next 4 steps describe more precisely the above procedure. Let for each node, the shortest path distance from the source using only uphill edges be denoted as sp_from_s(v). Moreover, let sp_to_t(v) denote the shortest path distance from node v to the sink t, for every $v \in V$.

1. For all nodes $v \in V$ set sp_from_s(v) $= \infty$ and sp_to_t(v) $= \infty$.
2. Let $G_1 = (V, E_1)$ be the graph induced by G if we remove all downhill edges. Run SSSP algorithm on G_1 using as source the node s. For every node in V set sp_from_s(v) accordingly.
3. Let $G_2 = (V, E_2)$ be the graph induced by G if we remove all uphill edges. Run SSSP algorithm on G_2 using as source the node t. For every node in V set sp_to_t(v) accordingly.
4. Find $\min_{v \in V}\{\text{sp\_from\_s}(v) + \text{sp\_to\_t}(v)\}$. Choose the corresponding uphill and downhill paths in order to get the desired s-t path.

The above procedure clearly produces the correct answer. Regarding the running time we get:

- Step 1 takes time $\mathcal{O}(n)$
- Step 2 takes time equal to the running time of an SSSP algorithm plus $\mathcal{O}(n)$ for updating the sp_from_s values.
- Step 3 same as Step 2.
- Step 4 takes time $\mathcal{O}(n)$

The running time of an SSSP algorithm dominates all other steps. Let $SP(m, n)$ denote the running time of an SSSP algorithm on a graph with m edges and n nodes. Then the above procedure takes time $\mathcal{O}(\text{SP}(m, n))$.

Part 2

For the second part we also have level roads that we are allowed to use either we go uphill or downhill. Thus the only modification of the above procedure is that in Step 2 and Step 3 we include these edges as edges of both G_1 and G_2.

Exercise 4

Give an n-element set of K-bit integers such that the veb search tree takes $\Omega(n \log K)$ space.

Solution: Each non-trivial VEB-tree of bit-breadth $2k$ stores

- a minimum value (constant space)
- a maximum value (constant space)
- a top-tree (space for one VEB-tree of bit-breadth k)
- a sequence of bottom-trees (space for 2^k VEB-trees of bit-breadth k).

In order to maximize the space consumption, it is most promising to concentrate on the bottom trees. If we are given n numbers, then each of them shall create a bottom-tree structure. Thus, the high parts of these n numbers have all to be different. We assume that their low parts are simply set to 0. Each of the bottom tree structures shall have the deepest possible nesting. Up to now, each of the bottom trees contains just one element. To avoid a trivial representation, we give a "partner number" to each of the n numbers (thus doubling the number of numbers). This partner number has the very same high part, but its low part is set to 1 instead of 0. Thus, each bottom tree becomes non trivial. Each of them stores only the low parts of a partner pair. As the high parts of these low parts are equal, both partners fall again into the same bottom tree. This bottom tree is again non-trivial and has to create another bottom tree. Each time, the bit-breadth of the bottom trees is divided by two. If $2k$ is the initial bit-breadth, we get $\log_2(k)$ bottom trees for one partner pair. Since we have n of such partner pairs, we have $n \cdot \log_2(k)$ bottom trees. Thus, space consumption is in $\Omega(n \cdot \log_2(k))$. The set is thus:

$$\bigcup_{i=1}^{n/2} \{2^k \cdot i, 2^k \cdot i + 1\}$$

This presupposes that a sequence of b bottom trees uses a space of $\Omega(b)$ – like this is the case with a linked list. If a hash-table (or, even worse: an array) is used, the space consumption increases.