Home Assignment 2

Jaan Tollander de Balsch

October 22, 2018

Contents

Question 1: Application to Graph Search

You are given an undirected graph $G = (V, E)$ where V is a list of cities and E is the road pattern between them. The road $e = \{u, v\}$ connects city u and city v, and you know its distance in kilometre $l(e)$. You want to drive from city s to city t. You own a small car which can only hold enough gas to cover Q km. There is a gas station in each city, but not between cities. Therefore, you can only take a route if every one of its edges has length at most Q.

Part 1

Describe an algorithm that given G, s, t and Q, checks whether there is a feasible route from s to t. Your algorithm should run in time $O(V + E)$.

Breadth-first search (BFS) can be modified to perform a search to find a feasible route. Instead of traversing all the edges, we only traverse those that satisfy the condition

$$l(e) \leq Q.$$

Assuming that this is a constant time operation, the worst case performance, $O(|E| + |V|)$, will not change. (Cormen et al., 2009, ch. 22.3)

Part 2

Modify Dijkstra's algorithm to, given G, s, t, compute a path from s to t that minimizes the length of the longest edge on that path.

The weight function is defined as the distance $l(e)$ if it's below the cars capacity Q and otherwise, it is infinite. Practically, this achieves the same outcome as removing the edges that have distance over Q from the graph.

$$w(e) = \begin{cases} l(e) & l(e) < Q \\ \infty & l(e) \geq Q \end{cases} \tag{1}$$

The pseudocode for Dijkstra's algorithm is given by (Cormen et al., 2009, ch.24.3). I have written this algorithm down in the appendix *Dijkstra's Algorithm*.

The modified Dijkstra's algorithm will *minimize the longest edge* in the path from s to t instead of minimizing the total length of the path. In order to minimize the longest edge, will need to modify the Relax(u, v, w) function into:

Relax'(u, v, w)

1) **if** $v.d > \max(u.d, w(u, v))$
2) _____ $v.d = \max(u.d, w(u, v))$
3) _____ $v.\pi = u$

This modification, instead of updating the of upper bound $v.d$ with total weight of the path, will update it with the maximum of the previous upper bound $u.d$ and the weight of the edge $e = \{u, v\}$ if the maximum is smaller than the upper bound $v.d$.

Question 2: Negative Weights

In this problem, assume that you are given an access to a sub-routine Dijkstra'(V', E', w', s') which returns an array D such that, for each vertex $v \in V'$, the length of a shortest path from s' to v is equal to $D[v]$.

Assume that the running time of this sub-routine is always at most $T(n, m)$ on n-vertex m-edge graph.

Part 1

You are given an input directed graph $G = (V, E)$, source $s \in V$, and a weight function w for which there is exactly one edge $e \in E$ with a negative weight. Assume that there is no negative cycle. Describe an algorithm that computes shortest paths from s to each vertex in V in time $O(T(|V|, |E|))$.

The algorithm computes the distance array D and then determines which edges $(u, v) \in E$ belong to the shortest path tree by checking if the distance to u added with the weight of the edge (u, v) equals the distance to v.

Input: A graph G, weight function $w : E \to \mathbb{R}$ and source vertex $s \in V$.

Output: A graph $G = (V, E')$ that consist of vertices V and edges E', where by following the edges all the shortest from s to each vertex $v \in V$ can be reconstructed.

1) $D =$ Dijkstra'$(G.V, G.E, w, s)$
2) $E' = \emptyset$
3) **for** each edge $(u, v) \in E$
4) _____ **if** $D[u] + w(u, v) = D[v]$
5) _____ _____ $E' = E' \cup \{(u, v)\}$

The algorithm runs in time

$$O(T(|V|, |E|)) + O(|E|) = O(T(|V|, |E|)).$$

Since $O(T(|V|, |E|))$ depends on the number of edges it cannot be faster than $O(|E|)$.

Part 2

You are given an input directed graph $G = (V, E)$, source $s \in V$, and a weight function w for which there are exactly k edges $e \in E$ with negative weight. Assume that there is no negative cycle. Describe an algorithm that computes shortest paths from s to each vertex in V in time $O((k + 1)!\, T(|V|, |E|))$.

I couldn't figure out how the algorith for the second part differs from the first part.

Question 3: Multiple Selection

We discussed the method of selection using $O(n)$ comparisons. Recall that selection(A, k) takes as input an unsorted array A and returns the k-th smallest elements in A.

Now, we are interested in doing multiple selection efficiently. Design an algorithm that takes array A together with integers $1 \leq k_1 \leq k_2 \leq ... \leq k_m \leq n$ and returns the k_i-th smallest elements for all i. For instance, mult_selection$(A, 1, n)$ is expected to return the minimum together with the maximum of elements in A.

Describe an algorithm for multiple selection that runs in time $O(n \log(m + 1))$.

???

Question 4: Dijkstra's Modification

Shortest paths are not always unique: sometimes there are two or more different paths with the minimum possible length. Show how to solve the following problem in $O((V + E) \log E)$ time.

Input: An undirected graph $G = (V, E)$; edge lengths $l_e > 0$; vertex $s \in V$.

Output: A boolean array $usp[\cdot]$ where, for each $v \in V$ the value [V] is true if and only if there is a unique shortest path from s to v.

Modify Dijkstra's algorithm to solve this problem.

The pseudocode for Dijkstra's algorithm is given by (Cormen et al., 2009, ch.24.3). I have written this algorithm down in the appendix *Dijkstra's Algorithm*.

The values the *usp* array should be initialized to $TRUE$.

The Relax function is modified such that when a vertex u is relaxed and if it finds a new lowest upper bound $v.d > u.d + w(u, v)$, it sets the the corresponding value in the array *ups* to $TRUE$. Since it is the first time it encounters this lowest value, it must be *unique* shortest path.

On the other hand, if we encounter that new lowest upper bound $v.d$ is equal to the sum of the upper bound of vertex u and weight of the edge (u, v), $u.d +$

$w(u, v)$, it means that the algorithm has found a possible *non-unique* shortest path to the vertex v.

Relax'(u, v, w)

1) **if** $v.d > u.d + w(u, v)$
2) _____ $v.d = u.d + w(u, v)$
3) _____ $v.\pi = u$
4) _____ $usp[v] = TRUE$
5) **else if** $v.d = u.d + w(u, v)$
6) _____ $usp[v] = FALSE$

The running time of the Dijkstra's algorithm is

$$O((|V| + |E|) \log |E|))$$

as given in (Cormen et al., 2009, ch. 24.3). The modifications made to the algorithm will not change this since the operations on array *usp* are simple constant time operations.

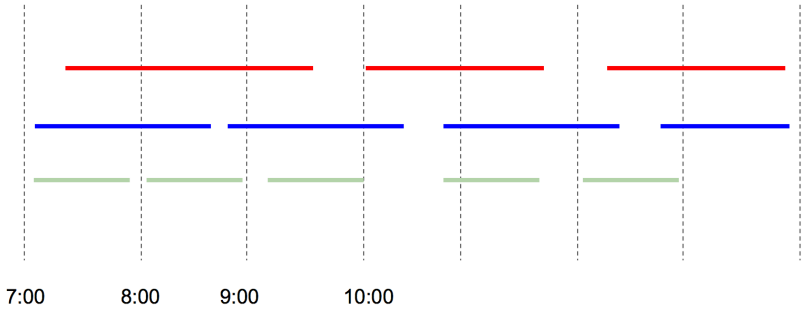
Question 5: Lecture Hall Allocation

Aalto University receives multiple requests from professors to book meeting rooms on the upcoming Christmas Day 2018; professors like to work hard, and working on Christmas is a part of their enjoyment (professors are masochistic in nature – and you could argue some may be sadistic).

There are k lecture halls in total and n Aalto professors. Each professor i makes a request of the form $[s_i, t_i]$ where s_i and t_i are the starting and ending tiem of their booking.

Describe an efficient algorithm that determines whether the k lecture halls are enough to serve the requests by professors. If they are enough, your algorithm should produce the list of requests to be scheduled in each room. If now, the algorithm can simply output NO.

Please see below a sample input where $k = 3$ lecture halls are enough to serve the requests:



The lecture hall allocation problem can be solved using *interval scheduling*. Interval scheduling aims to partition intervals into sets of non-overlapping intervals. An optimal algorithm partitions them into the minimum number of sets. An algorithm that can solve the interval scheduling problem optimally is a greedy algorithm called *earliest deadline first*, which is greedy in the sense that it finds the optimal solution by always choosing candidates by the smallest ending time,i.e, deadline. (Kleinberg and Tardos, 2005, ch. 4.1)

A greedy algorithm for scheduling a single set of sorted intervals.

Input: A set of intervals I sorted by finishing time.

Output: A set of non-overlapping intervals S.

Find-Non-Overlapping-Intervals(x, I)

1) $(s, t) = x$
2) $S = \{x\}$
3) **for** each $(s_i, t_i) \in I$
4) _____ **if** $s_i \geq t$
5) _____ _____ $S = S \cup \{(s_i, t_i)\}$
6) _____ _____ $(s, t) = (s_i, t_i)$
7) **return** S

This algorithm runs in linear time $O(|I|)$ dependent on the size of the set of intervals I. It does not take into account the time of sorting the intervals I, which is done when scheduling all intervals.

The full algorithm for scheduling all intervals I into non-overlapping sets of intervals.

Input: A set of intervals $I = \{(s_i, t_i) \mid i \in \{1, ..., n\}\}$, where s_i denotes the starting time and t_i the ending time of an interval i.

Output: Minimum sized set S of sets of non-overlapping intervals.

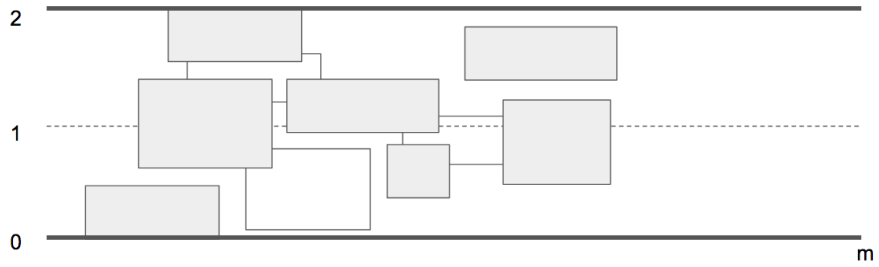
Earliest-Deadline-First-Scheduling(I)

1) J = Sort-By-Finishing-Time(I)
2) $S = \emptyset$
3) **while** $J \neq \emptyset$
4) _____ K = Find-Non-Overlapping-Intervals($J[1], J - J[1]$)
5) _____ $J = J - K$
6) _____ $S = S \cup \{K\}$
7) **return** S

This algorithm runs in quadratic time $O(n^2)$, because the while loop is bound to run $O(n)$ times, finding overlapping intervals is also bound by $O(n)$ and the sorting operation takes $O(n \log n)$ time.

Question 6: Label Placement

You are given a collection of n rectangles (represented by the coordinates of their four corners) that lie in the strip of height 2 and length m. Each rectangle has a height between $[1/3, 1]$ and can be arbitrarily long. See the image below.



Design an algorithm efficiently finds the maximum subset of rectangles that do not overlap. More formally, each rectangle is represented by the point set $[a, b] \times [c, d]$, and two rectangles overlap if and only if their interiors have non-empty intersection.

???

Question 7: Smooth Non-decreasing Subsequence

For a sequence of non-negative real numbers $a_1, a_2, ..., a_n$, we say that a subsequence $a_{i_1}, a_{i_2}, ..., a_{i_k}$ where $i_1 < i_2 < ... < i_k$ is non-decresing subsequence of length if $a_{i_j} \leq a_{i_{j+1}}$ for all $1 \leq j < k$. Further, we say that such a subsequence is B-smooth if we have $a_{i_{j+1}} - a_{i_j} \leq B$ for all $1 \leq j < k$.

Describe a dynamic programming algorithm that takes as input the sequence $a_1, ..., a_n$ together with integer B and produces the longest B-smooth non-decreasing subsequence (i.e. find the one with maximum k). To receive full credits, clearly, define the DP table as well as the recurrence. Briefly analyze the running time of your algorithm.

Finding B-smooth subsequences can be solved by adapting the algorithm of finding the longest increasing (non-decreasing) subsequences. (Dasgupta, Papadimitriou and Vazirani, 2008, ch. 6.2)

Construct the directed acyclic graph (DAG) $G = (V, E)$ such the vertices are the indices $V = \{i_1, ..., i_n\}$ and there exists an edge between two vertices $(i_{j+1}, i_j) \in E$ if

$$0 \leq a_{i_{j+1}} - a_{i_j} \leq B.$$

The DAG can be constructed as an adjacency list. Since there are n vertices, the time complexity of constructing the DAG is $O(n^2)$ because every element in the sequence a_i needs to be compared against every element in the sequence that comes after it. This also means that the number of edges $|E|$ is bound by $O(n^2)$. The problem now is to find the longest path in the DAG, record the predecessor vertices and then reconstruct the path.

Input: Sequence of positive real numbers $A = \langle a_1, a_2, ..., a_n \rangle$ and real number B.

Output: Longest B-smooth subsequence of sequence A.

Longest-Path-DAG(A, B)

1) Contruct the graph $G = (V, E)$
2) Initialize arrays L and $prev$ with NIL and to size n.
3) **for** $j \in \{1, ..., n\}$
4) _____ $i = \text{argmax}(L(i) : (i, j) \in E)$
5) _____ $prev(j) = i$
6) _____ $L(j) = 1 + L(i)$
7) $j = \text{argmax}\, L(j)$
8) $S = \langle j \rangle$
9) **while** $prev(j) \neq NIL$
10) _____ $j = prev(j)$
11) _____ Prepend(S, j)
12) **return** S

The algorithms calls argmax$(L(i) : (i, j) \in E)$, which takes $O(|E|) = O(n^2)$ time, n times making the total time complexity of

$$O(n^3).$$

Question 8: Catching Ameet's Mistake?

Ameet proposed the following algorithm and claimed that the algorithm solves Minimum Spanning Trees (MST) problem. He bets 100 EUR that his algorithm would be correct. Would you bet against him? Either prove that he is correct to give a counterexample showing that he is not.

Ameet-MST(G, w)

1) Sort the edges into a non-increasing order of edge weights w
2) $T = E$
3) **for** each edge $e \in E$, taken in non-increasing order of weights
4) _____ **if** $T - e$ is a connected graph, then remove e from T

The minimum spanning tree (MST) problem is defined as: Given a connected graph $G = (V, E)$ and weight function $w : E \to \mathbb{R}$, find an acyclic subset $T \subseteq E$ that connect all of the vertices and whose total weight $w(T) = \sum_{(u,v \in T)} w(u, v)$ is minimized. (Cormen et al., 2009, ch.23)

Here we'll use a slightly more clear representation of the algorithm's pseudocode.

Ameet-MST(G, w)

1) $E' =$ Sort-Non-Increasing(E, w)
2) $T = E$
3) **for** each edge $e \in E'$
4) _____ **if** $T - \{e\}$ is a connected graph
5) _____ _____ $T = T \cap \{e\}$

There are two conditions that are required for Ameet-MST algorithm to be correct.

The resulting graph $G' = (V, T)$ is a spanning tree, i.e, the graph is connected and has no cycles: The input graph G is a connected graph by definition and the algorithm only removes edges if the result is a connected graph, therefore final graph G' must also be connected. Furthermore, since we loop over all of the edges in the graph there will be only one edge left between each two vertices, i.e, the algorithm removes all cycles.

(Lemma 1)

Lemma 1. *Given a cycle $v_1 \to v_2 \to ... \to v_n \to v_1$ of edges v_i in a weighted, undirected graph, removing the edge v_i with the highest weight from the cycle, will result a path where all vertices are still reachable from any other vertex and then the total weight of the cycle is minimized.*

The total weight of the result $w(T)$ is minimized: Since the algorithm is only removing edges e when the resulting graph remains connected, i.e, the edge e is a part of a cycle, and the edges E were sorted in non-increasing (decreasing) order, i.e., the edges e are looped from largest to smallest by their weight, due to *Lemma*1 removing the edge e from the graph G will minimize the total weight of the resulting spanning tree G'.

I'll bet that the algorithm works (but slowly). Ameets algorithm can be regarded as a crude brute force solution to the MST problem.

Question 9: Application of Data Structures

Given k sorted arrays $A_1, A_2, ..., A_k$ as input, devise an algorithm that merges arrays into one sorted array A that combines elements from all such A_i. Your algorithm should run in time $O(n \log(k+1))$ where n is the total number of elements in the combined array A.

During mergesort – a divide-and-conquer approach for solving the sorting problem – a subproblem that arises is the sorting of sorted arrays. A slight modification to the algorithm can solve the sorting of sorted arrays in $O(n \log(k+1))$ compared to the mergesort that run in $O(n \log n)$ for an array with n elements. The following pseudocode algorithm is adapted from (Dasgupta, Papadimitriou and Vazirani, 2008, ch. 2.3).

Input: Two sorted arrays X and Y.

Output: Sorted array Z that contains all the elements in arrays X and Y.

Merge(X, Y)

1) **if** $|X| = 0$
2) ____ **return** Y
3) **if** $|Y| = 0$
4) ____ **return** X
5) **if** $X[1] < Y[1]$
6) ____ **return** $X[1] \circ \text{Merge}(X - X[1], Y)$
7) **else**
8) ____ **return** $Y[1] \circ \text{Merge}(X, Y - Y[1])$

Where $\circ$ denotes concatenation. Runtime of this algorithm is linear $O(|X| + |Y|)$ because in each recursion the algorithm removes one element from either array.

Input: An array of k sorted arrays $\langle A_1, A_2, ..., A_k \rangle$.

Output: Sorted array A that combines elements from all arrays A_i.

Merge-Sort-Arrays($\langle A_1, A_2, ..., A_k \rangle$)

1) **if** $k > 1$
2) ____ $B' =$ Merge-Sort-Arrays($\langle A_1, A_2, ..., A_{\lfloor n/2 \rfloor} \rangle$))
3) ____ $B'' =$ Merge-Sort-Arrays($\langle A_{\lfloor n/2 \rfloor + 1}, A_2, ..., A_k \rangle$))
4) ____ **return** Merge(B', B'')
5) **else**
6) ____ **return** A_1

Figure 2.3 in (Dasgupta, Papadimitriou and Vazirani, 2008, ch. 2.2, pg. 59) visualizes recurrence relations in the divide-and-conquer algorithm. The depth of the recurrence relation in Merge-Sort-Arrays is $\log_2 k$, i.e., there are k sorted arrays and the operation splits the problem into half each recursive call until it reaches the base case and starts merging the arrays.

The total computational complexity of the all the Merge operations in each recursive level of depth has complexity $O(n)$. For example, at bottom level we merge arrays

$$\text{Merge}(A_1, A_2), ... \text{Merge}(A_{k-1}, A_k)$$

which adds up to complexity of

$$O(n_1 + n_2) + ...O(n_{k-1} + n_k) = O(n).$$

At next level we would merge the arrays from the previous level, adding to computational complexity of

$$O((n_1 + n_2) + (n_3 + n_4)) + ...O((n_{k-3} + n_{k-2}) + (n_{k-1} + n_k)) \in O(n).$$

Since there are $\log k$ levels, the computational complexity of the algorithm is

$$O(n)O(\log k) = O(n \log k).$$

Question 10: Minimum Spanning Trees with Updates

You are given graph $G = (V, E)$, weight function w on edges, together with a subtree T of G that is guaranteed to be a minimum spanning tree.

Describe a deterministic algorithm that, given an additional edge e (not in G) with cost $w(e)$, find a minimum spanning tree T' for $G' = (V, E \cup \{e\})$ in $O(|V| + |E|)$.

Notice that your algorithm should not try to recompute MST from scratch but should instead use information about T.

Input:

- Subtree T of graph G that is guaranteed to be a minimum spanning tree.
- Weight function $w : E \to \mathbb{R}$
- The vertices $u, v \in T$ which form an edge $e = (u, v)$ that the algorithm adds to the graph.

Output: New subtree T' of $G' = (V, E \cup \{e\})$ that is guaranteed to be a minimum spanning tree.

MST-Update(T, w, u, v)

1) $P =$ Find-Path(T, u, v)
2) $e' =$ Find-Maximum-Weighted-Edge(P, w)
3) $e = (u, v)$
4) **if** $w(e') \leq w(e)$
5) _____ $T' = T$
6) **else**
7) _____ $T' = (T \cap e') \cup e$

MST-Update algorithm is based on the observation that adding a new edge to a tree will create a cycle in it. If anyone edge is removed from this cycle the graph will become a tree again. Since the algorithm has to maintain minimum total weight it must remove an edge with the highest weight. See *Lemma*1.

The Find-Path operation outputs a set of edges P that form a path from u to v in the tree T. It can be implemented using for example Breadth-first search (BFS) which has the worst-case performance of $O(|V| + |E|)$.

The Find-Maximum-Weighted-Edge operation outputs an edge e' with the highest weight from a set of edges P measured by the weight function w. It is a linear time operation and depends on the size of the input set $O(|P|)$.

In lines 3-7 the algorithm compares the weight of the edge e' with the highest weight on the path P to the weight of the edge $e = (u, v)$. If the weight $w(e)$ is higher or equal than $w(e')$ the tree remains unchanged, otherwise, the algorithm removes the edge e' from the graph and add the edge e to the graph.

Appendices

Dijkstra's Algorithm

A good pseudocode about the implementation of the Dijkstra's algorithm and related analysis of its correctness and computational complexity is given by (Cormen et al., 2009, ch. 24, ch. 24.3).

Input:

- A graph $G = (V, E)$
- Weigh function $w : E \to \mathbb{R}^+$
- Souce vertex $s \in V$

Output: The shortest paths from the source vertex s to vertices $v \in V$, which can be constructed by following the predecessors of each vertex after the algorithm has executed.

Implementation details:

- $v.d$ denotes the upper bound on the weight of a shortest path from s to v (i.e, a shortest path estimate).
- $v.\pi$ denotes the predesessor node.
- Q denotes a min-priority queue that is queried by the value $v.d$.

The pseudocode:

Initialize-Single-Source(G, s)

1) **for** each $v \in G.V$
2) _____ $v.d = \infty$
3) _____ $v.\pi = NIL$
4) $s.d = 0$

Relax(u, v, w)

1) **if** $v.d > u.d + w(u, v)$
2) _____ $v.d = u.d + w(u, v)$
3) _____ $v.\pi = u$

Dijkstra(G, w, s)

1) Initialize-Single-Source(G, s)
2) $S = \emptyset$
3) $Q = G.V$
4) **while** $Q \neq \emptyset$
5) _____ $u =$ Extract-Min(Q)
6) _____ $S = S \cup \{u\}$
7) _____ **for** each vertex $v \in G.Adj[u]$
8) _____ _____ Relax(u, v, w)

Dijkstra's algorithm runs in time $O(|E| \log |V|)$ if min-priority queue is implemented using binary heap, which improves to $O(|V| \log |V| + |E|)$ if it is implemented using Fibonacci heap.

References

Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C., 2009. *Introduction to algorithms*. MIT press.

Dasgupta, S., Papadimitriou, C.H. and Vazirani, U., 2008. *Algorithms*. 1st ed. New York, NY, USA: McGraw-Hill, Inc.

Kleinberg, J. and Tardos, E., 2005. *Algorithm design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.