KØBENHAVNS UNIVERSITET

Diskret Matematik og Formelle Sprog: Problem Set 4

Due: Sunday March 21 at 23:59 CET.
Submission: Please submit your solutions via *Absalon* as PDF file. State your name and e-mail address close to the top of the first page. Solutions should be written in LaTeX or some other math-aware typesetting system with reasonable margins on all sides (at least 2.5 cm). Please try to be precise and to the point in your solutions and refrain from vague statements. *Write so that a fellow student of yours can read, understand, and verify your solutions.* In addition to what is stated below, the general rules in the course information always apply.
Collaboration: Discussions of ideas in groups of two to three people are allowed and indeed, encouraged—but you should always write up your solutions completely on your own, from start to finish, and you should understand all aspects of them fully. It is not allowed to compose draft solutions together and then continue editing individually, or to share any text, formulas, or pseudo-code. Also, no such material may be downloaded from the internet and/or used verbatim. Submitted solutions will be checked for plagiarism.
Grading: A score of 160 points is guaranteed to be enough to pass this problem set.
Questions: Please do not hesitate to ask the instructor or TAs if any problem statement is unclear, but please make sure to send private messages — sometimes specific enough questions could give away the solution to your fellow students, and we want all of you to benefit from working on and learning on the problems. Good luck!

1 (80 p) Consider the graph in Figure 1. We assume that this graph is represented in adjacency list format, with the neighbours in each adjacency list sorted in lexicographic order (i.e., in the order $a, b, c, d, \ldots$), so this is the order in which vertices are encountered when going through neighbour lists.

1a (30 p) Run the code for depth-first search by hand on this graph, starting in the vertex a. Describe in every recursive call which neighbours are being considered and how they are dealt with. Show the spanning tree produced by the search.

Solution: See Figure 2 for an illustration of the algorithm execution described below (with every node marked by the times it was visited and finished) and the resulting spanning tree (redrawn separately in red for clarity).

1. We start in vertex a, which we mark as visited. Since the neighbours in each adjacency list are sorted in lexicographic order, we first neighbour of a that we look at is b.

2. Vertex b has not been visited, so we make a recursive call for b, and add the edge (a, b) to the spanning tree. The first neighbour of b is a, which is visited.

3. The next neighbour c of b has not been visited, so we make a recursive call for c and add the edge (b, c) to the spanning tree. The only neighbour of c is b, which is already visited, so the recursive call returns.

4. We are now back in the call made for b. The next neighbour e of b is not visited, so we make a recursive call for e and add the edge (b, e) to the spanning tree. The first neighbours of e are a and b, which are both already visited.

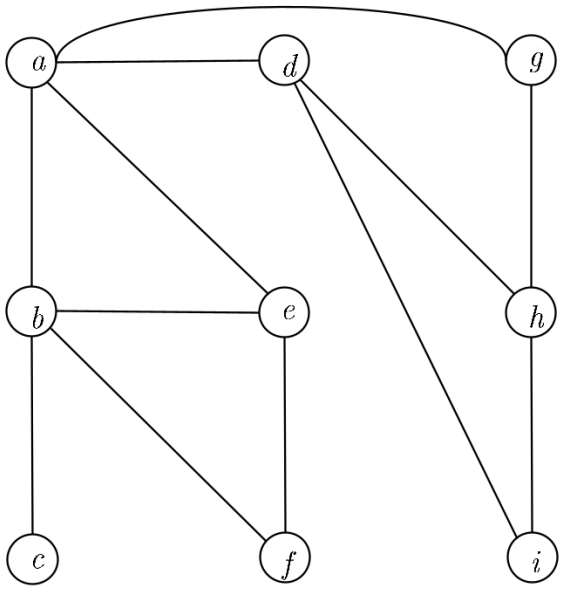

Figure 1: Undirected unweighted graph for graph traversals in Problem 1.

5. The next neighbour f of e has not been visited, so we make a recursive call for f and add the edge (e, f) to the spanning tree. The neighbours b and e of f are both visited already, so the recursive call returns.

6. We are now back in the call made for e, but since all neighbours have now been processed, this call returns.

7. This brings us back to the call made for b, and since all neighbours of b have also been processed, this call returns.

8. At this point, we have returned to the very first DFS call made, namely for a. The next neighbour of a after b is d, which has not been visited, so we visit d and add the edge (a, d) to the tree. The first neighbour a of d has been visited.

9. The next neighbour of d is h, which we see for the first time and so visit, adding (d, h) to the tree. The first neighbour of h is d, which is where we came from.

10. The second neighbour of h is g, which we see for the first time and so visit, adding (g, h) to the tree. The neighbours of g are a and h, which have both been visited, so the recursive call to g immediately returns.

11. The third and final neighbour of h is i, which we visit, adding the edge (h, i) to the spanning tree.

12. Now all vertices have been visited, so in what remains the algorithm will finish the recursive calls to i, h, d, and a (in this order) by discovering for each vertex that there are no non-visited vertices left.

1b (30 p) Run the code for breadth-first search by hand on this graph, also starting in the vertex a. Describe for every vertex which neighbours are being considered and how they are dealt with, and how the queue changes. Show the spanning tree produced by the search.

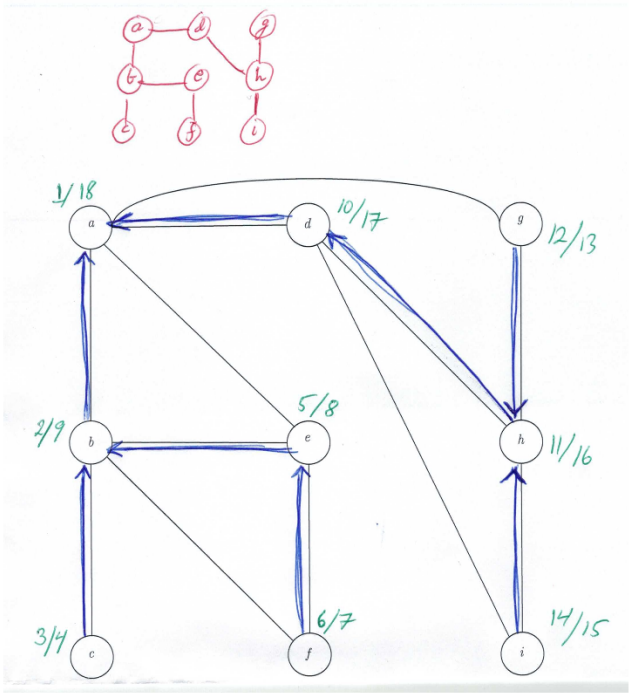

Figure 2: Depth-first search for graph in Figure 1.

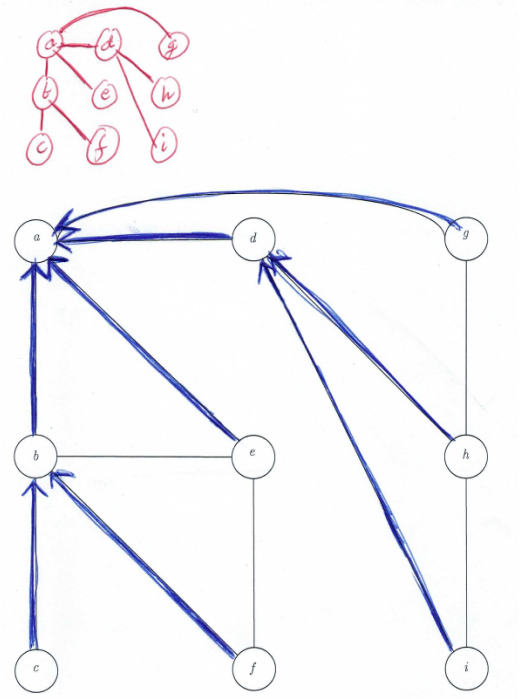

Figure 3: Breadth-first search for graph in Figure 1.

Solution: See Figure 3 for an illustration of the algorithm execution described below and the resulting spanning tree.

1. At the start, the queue contains only the vertex a.
2. We dequeue a and add its neighbours b, d, e, and g to the queue in this order, as well as the edges (a, b), (a, d), (a, e), and (a, g), to the spanning tree, since none of these vertices had been seen before. The queue now looks like (b, d, e, g).
3. We dequeue b. Neighbour a is already marked, but c is not and so is enqueued. Neighbour e is also marked, but f is not and so is enqueued. The edges (b, c) and (b, f) are added to the spanning tree. The queue now looks like (d, e, g, c, f).
4. We dequeue d. Neighbour a is already marked, but h and i are new vertices. We therefore enqueue them and add the edges (d, h) and (d, i) to the spanning tree. The queue now looks like (e, g, c, f, h, i).
5. We dequeue e. All neighbours are already marked, so no new vertices are enqueued and the queue shrinks to (g, c, f, h, i).
6. We dequeue g. Again all neighbours are marked, so no new vertices are enqueued and the queue shrinks to (c, f, h, i).
7. The algorithm will continue like this until the queue is empty, because all vertices have already been discovered and so no new vertices can be added. (This was true already after processing d, and it would also have been in order to point this out immediately after this step.)

1c (20 p) Suppose that the graph in Figure 1 is modified to give every edge weight 1, and that we run Dijkstra's algorithm starting in vertex a. How does the tree computed by Dijkstra's algorithm compare to that produced by DFS? What about BFS?

Solution: If all edges have the same weight 1, then Dijkstra's algorithm produces the same result as breadth-first search. Just as the BFS trees and DFS trees can be very different, there is no particular relation between the DFS tree and what Dijkstra's algorithm would produce in this case.

2 (110 p) Consider the graph in Figure 4. We assume that this graph is represented in adjacency list format, with the neighbours in each adjacency list sorted in lexicographic order (so that this is the order in which vertices are encountered when going through neighbour lists).

2a (30 p) Generate a minimum spanning tree by running Prim's algorithm by hand on this graph, starting in the vertex a. Use a heap for the priority queue implementation. Describe for every dequeueing operation which neighbours are being considered and how they are dealt with, and show how the heap changes. Also show the spanning tree produced by the algorithm.

Solution: The execution of Prim's algorithm is illustrated on the graph in Figure 5. We do not describe in this solution sketch the details of how the heap changes. At the outset, the vertex a has key 0 and all other vertices have key ∞. In what follows, if vertex v has key value k, we will use the notation (v, k) for this for brevity.

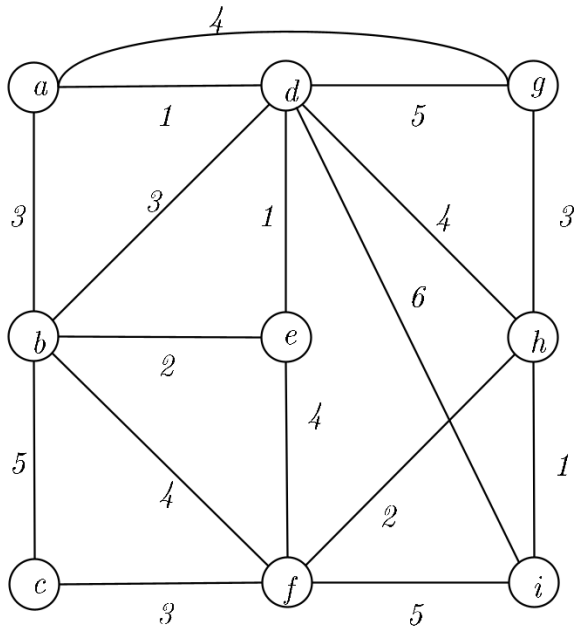

Figure 4: Undirected weighted graph for minimum spanning trees in Problem 2 and shortest paths in Problem 3.

1. Dequeue a. The keys of the neighbours of a are updated to $(b, 3)$, $(d, 1)$, $(g, 4)$, after which d is at the front of the queue.

2. Dequeue d and add edge (a, d) to the spanning tree. The keys of the neighbours of d are updated to $(e, 1)$, $(h, 4)$, $(i, 6)$. Neighbours b and g already have values that are not larger than the weights of the edges to d, and vertex a has already been processed. Now e is at the front of the queue.

3. Dequeue e and add edge (d, e) to the spanning tree. The keys of the neighbours are updated to $(b, 2)$, $(f, 4)$. Now b is at the front of the queue.

4. Dequeue b and add edge (b, e) to the spanning tree. The key of c is updated to 5. All other neighbours of b are already processed or have at least as good keys already. Now either f or g is at the front of the queue—let us say it is f, just because f comes before g in alphabetical order.

5. Dequeue f and add edge (e, f) to the spanning tree. The neighbour key updates that are made in this step are $(c, 3)$, $(h, 2)$, $(i, 5)$, and g is no longer next in queue, since both h and c sneaked ahead.

6. Dequeue h and add edge (f, h) to the spanning tree. The neighbour key updates that are made in this step are $(g, 3)$, $(i, 1)$, and so now i is next in queue.

7. Dequeue i and add edge (h, i). There are no key updates, and either c or g is at the front of the queue let us say that g loses out again.

8. Dequeue c and add edge (c, f). There are no key updates, g is the only vertex left in the queue.

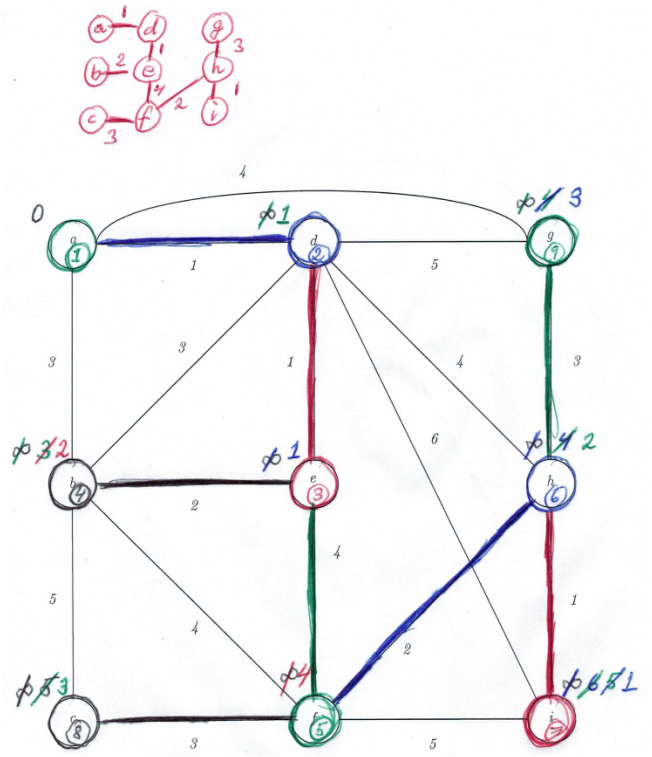

Figure 5: Minimum spanning tree for graph in Figure 4 using Prim's algorithm.

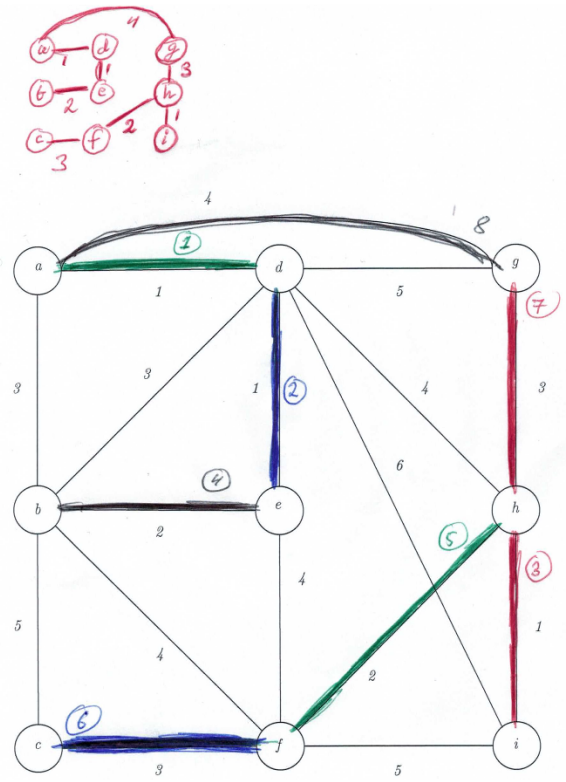

Figure 6: Minimum spanning tree for graph in Figure 4 using Kruskal's algorithm.

9. Dequeue g and add edge (g, h) to the minimum spanning tree, which is now complete.

2b (30 p) Generate a minimum spanning tree by running Kruskal's algorithm by hand on this graph. Assume that edges of the same weight are sorted in lexicographic order (so that we would have (a, b) coming before (a, d), which would in turn come before (b, a) for hypothetical edges of equal weight). Describe how the forest changes at each step (but you do not need to describe in detail how the set operations are implemented). Also show the final spanning tree produced by the algorithm.

Solution: We illustrate the execution of Kruskal's algorithm in Figure 6. After sorting, the edges will be processed in the following order:

Weight 1: (a, d), (d, e), (h, i).

Weight 2: (b, e), (f, h).

Weight 3: (a, b), (b, d), (c, f), (g, h).

Weight 4: (a, g), (b, f), (d, h), (e, f).

Weight 5: (b, c), (d, g), (f, i).

Weight 6: (d, i).

The algorithm now does the following:

1. Looking first at edges of weight 1, the edge (a, d) creates a forest $\{a, d\}$ and is added.
2. The edge (d, e) enlarges this forest to $\{a, d, e\}$ and is added.
3. The edge (h, i) creates a forest $\{h, i\}$ and is added.
4. Moving on to edges of weight 2, the edge (b, e) grows the forest $\{a, d, e\}$ further to $\{a, b, d, e\}$ and is added.
5. The edge (f, h) grows the forest $\{h, i\}$ to $\{f, h, i\}$ and is added.
6. For the edges of weight 3, we see that (a, b) and (b, d) would both create cycles in the forest $\{a, d, e\}$, so they are discarded. The edge (c, f) grows the forest $\{f, h, i\}$ to $\{c, f, h, i\}$ and is added.
7. The edge (g, h) grows the forest $\{c, f, h, i\}$ to $\{c, f, g, h, i\}$ and is added.
8. For the weight-4 edges, according to our sorting order the edge (a, g) comes first, and is added since it creates a spanning tree for the full graph. Since we know we have now added the right number of edges, we can terminate without considering any further edges.

We note that in the last step we could also have added the edge (e, f) instead, if the edges of weight 4 would have been sorted differently, and this would have given us the minimum spanning tree in Figure 5. It would also have been possible to pick either of the two other weight-4 edges (b, f) or (d, h) to get other minimum spanning trees.

2c (10 p) Suppose that some vertex v with several neighbours in a graph G has a unique neighbour u such that the edge (u, v) has strictly smaller weight than any other edge incident to v. Is it true that the edge (u, v) must be included in any minimum spanning tree? Prove this or give a simple counter-example.

Solution: Yes, any MST has to include the edge (u, v), since this is the unique light edge in a cut with vertex v on one side and the rest of the graph on the other side. If there were an MST without the edge (u, v), then adding that edge to the MST would create a cycle, from which we could remove a heavier edge and get another spanning tree for the graph. This contradicts that the tree we started with was an MST.

2d (20 p) Suppose that some vertex v with several neighbours in a graph G has a unique neighbour u such that the edge (u, v) has strictly larger weight than any other edge incident to v. Is it true that the edge (u, v) can never be included in any minimum spanning tree? Prove this or give a simple counter-example.

Solution: No, this is not true. Consider the case when u has only one neighbour and this neighbour is v. Then the edge (u, v) has to be included in any MST.

2e (20 p) Suppose that T is a minimum spanning tree for a weighted, undirected graph G. Modify G by adding some constant $c \in \mathbb{R}^+$ to all edge weights. Is T still a minimum spanning tree? Prove this or give a simple counter-example.

Solution: If the spanning tree has M edges, then after the edge weight increase the new tree has a total weight that increased by an amount $M \cdot c$. But any spanning tree for a graph G will have the same number of edges, and hence the weight increase will be exactly the same for all spanning trees. This means, in particular, that all MSTs before the weight increase remain MSTs also after the weight increase.

3 (60 p) Consider again the graph in Figure 4 with the same assumptions.

3a (40 p) Run Dijkstra's algorithm by hand on this graph, starting in the vertex a. Use a heap for the priority queue implementation. Describe for every vertex which neighbours are being considered and how they are dealt with, and show how the heap changes. Also show the tree produced by the algorithm.

Solution: The execution of Dijkstra's algorithm is illustrated on the graph in Figure 7. We do not describe in this solution sketch the details of how the heap changes. At the outset, the vertex a has key 0 and all other vertices have key ∞. We will again use the notation (v, k) when vertex v has key value k.

1. Dequeue a and relax the neighbours to get $(b, 3)$ $(d, 1)$, $(g, 4)$; leaving d at the front of the priority queue.
2. Dequeue d, add the edge (a, d), and relax the neighbours to get $(e, 2)$, $(h, 5)$, $(i, 7)$. Note that a, b, and g are not affected by the relaxation since their key values do not decrease. Now e is first in line.
3. Dequeue e, add the edge (d, e), and relax the neighbours to get $(f, 6)$. Vertices b and d are not affected, and now b is at the front of the queue.
4. Dequeue b, add the edge (a, b), and relax the neighbours to get $(c, 8)$. No other neighbours of b are affected by the relax operations. Now g is first in line.
5. Dequeue g and add the edge (a, g). No neighbours of g are affected by the relax operations. Now h is first in line.

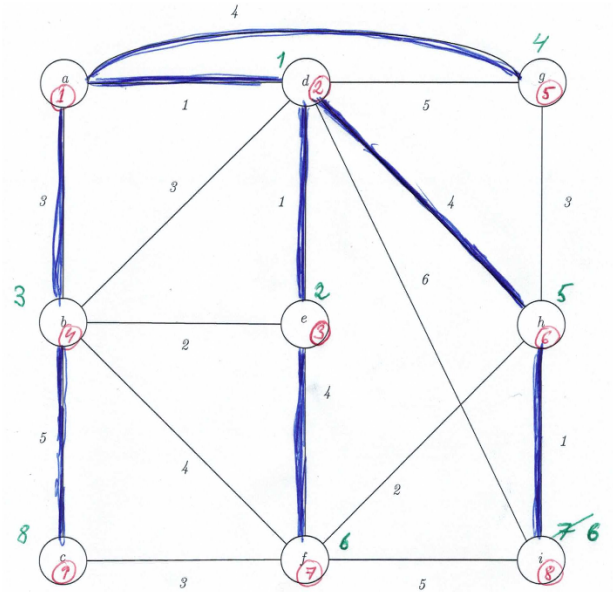

Figure 7: Shortest paths computed by Dijkstra's algorithm on the graph in Figure 4.

6. Dequeue h, add the edge (d, h), and relax the neighbours to get $(i, 6)$. No other neighbours of h are affected by the relax operations. Now either f or i is first in the queue.

7. In the last two steps, the edge (e, f) is added for f and the edge (h, i) is added to reach i. This concludes the execution of the algorithm.

3b (20 p) If Dijkstra's algorithm is run on an undirected graph, is it true that the spanning tree produced is a minimum spanning tree? Prove this or give a simple counter-example.

Solution: No, this is not true, and the spanning tree produced in our solution of Problem 3a shows this. (For instance, we can decrease the weight of this tree by replacing edge (a, b) by edge (b, e), or by replacing edge (d, h) by edge (g, h).)

4 (50 p) Let us return to our array $A = [5, 6, 4, 7, 3, 8, 2, 9, 1, 10]$ from problem set 2, which we still want to sort in increasing order.

4a (40 p) Run heap sort by hand on this array, and show in detail in every step of the algorithm what calls to the heap are made and how the array and the heap change.

4b (10 p) Suppose that we build a max-heap from any given array A. Is it true that once the array A has been converted to a correct max-heap, then considering the elements in reverse order (i.e., $A[n], A[n-1], \ldots, A[2], A[1]$) yields a correct min-heap? Prove this or give a simple counter-example.

Solution: No, this is not true. If we look at what the array from our max-heap in Problem 4a, it looks like $10, 9, 8, 7, 6, 4, 2, 5, 1, 3$. If we now take the array in the reverse order to get $3, 1, 5, 2, 4, 6, 7, 8, 9, 10$, then we have that 1 is the left child of the root 3, which violates the min-heap property.