KØBENHAVNS UNIVERSITET

Introduktion til diskret matematik og algoritmer: Problem Set 4

Due: Wednesday April 3 at 17:15 CET.
Submission: Please submit your solutions via *Absalon* as a 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. Never, ever just state the answer, but always make sure to explain your reasoning. *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 for problem sets stated on *Absalon* 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 pseudocode. Also, no such material may be downloaded from or generated via the internet to be used in draft or final solutions. Submitted solutions will be checked for plagiarism.
Grading: A score of 120 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 from, the problems. Good luck!

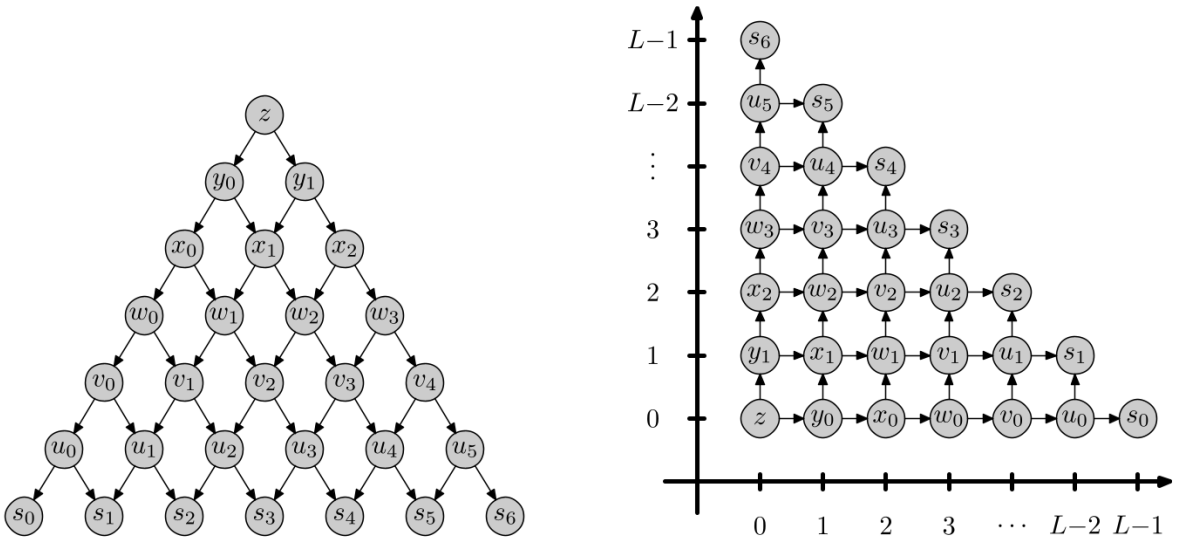
1 (100 p) Consider a directed graph that consists of L layers numbered from 0 up to $L-1$, with $i+1$ vertices in every layer i numbered from 0 to i, and with outgoing edges from vertex j in layer i to vertices j and $j+1$ in layer $i+1$. See Figure 1a for an illustration of this graph with 7 layers. We can agree to call this graph a *lattice graph* (because it can be obtained from a fragment of the integer lattice in 2 dimensions as shown in Figure 1b, but this is actually completely irrelevant to this problem).

Suppose we start in the unique source vertex on level 0 and walk along edges in the graph, flipping a fair coin at every vertex to decide whether to go left or right, until we reach one of the sinks in the last layer. For instance, in Figure 1a the walk "left–left–right–left–right–right" would visit vertices z, y_0, x_0, w_1, v_1, u_2, and end in s_3.

1a (40 p) For every vertex s_i in the lattice graph in Figure 1a, calculate the probability that such a walk ends in vertex s_i. Which vertex is the walk most likely to end up in?

1b (60 p) For a lattice graph with L layers, where $L \geq 2$ is a positive integer, calculate the probability that a walk as described above ends in vertex j in layer $L-1$ for $j = 0, 1, \ldots, L-1$.

Hint: Use the fact that all walks are equally likely to turn this into a counting problem.



(a) Lattice graph with 7 layers.

(b) Graph as connected points in 2D integer lattice.

Figure 1: Example graph for Problem 1.

Solution: Clearly, Problem 1a is a special case of Problem 1b, and here we will focus on the latter problem.

Using the hint, since a fair coin is being flipped we deduce that the probability of any particular walk is the same and is equal to $\left(\frac{1}{2}\right)^{L-1}$. In order to calculate the probability of reaching a certain leaf, it is sufficient to count the number of walks ending in that leaf. We can identify each walk with a binary sequence of length $L-1$, where 1 means going right and 0 means going left, say, so that the walk "left–left–right–left–right–right" in Figure 1a visiting vertices z, y_0, x_0, w_1, v_1, u_2, and s_3 is encoded as the sequence 001011.

Thinking a bit more, we realize that which leaf the walk ends up in is determined by the number of right turns in the walk, regardless of when these turns happen. Looking again at Figure 1a, any walk making exactly 3 right turns will end up in s_3. This means that the probability of reaching leaf j for $j = 0, 1, \ldots, L-1$ is proportional to the number of walks with j right turns, or the number of binary strings of length $L-1$ containing exactly j ones. But this number is nothing other than the binomial coefficient $\binom{L-1}{j}$. Thus, the probability of reaching leaf j can be written as

$$\Pr[\text{reaching leaf } j] = \binom{L-1}{j}\left(\frac{1}{2}\right)^{L-1} .$$

This number, in turn, is maximized when j is as close to $\frac{L-1}{2}$ as possible (so in Problem 1a the random walk is most likely to end up in s_3).

To see that $\binom{n}{k}$ is largest when k is as close to $n/2$ as possible, we can reason as follows. Firstly, since we have $\binom{n}{k} = \binom{n}{n-k}$, we can argue by symmetry and only consider $k \leq \lfloor n/2 \rfloor$. Suppose that $k < \lfloor n/2 \rfloor$. Then we will have $\binom{n}{k+1} > \binom{n}{k}$, since it holds that $\binom{n}{k+1} = \binom{n}{k} \cdot \frac{n-k}{k+1}$, which is straightforward to verify from the definition, and it is also easy to check that $\frac{n-k}{k+1} > 1$ if $k < \lfloor n/2 \rfloor$. This means that $\binom{n}{k}$ will increase with k until we reach $k = \lfloor n/2 \rfloor$.

2 (80 p) In this problem we wish to understand algorithms for minimum spanning trees.

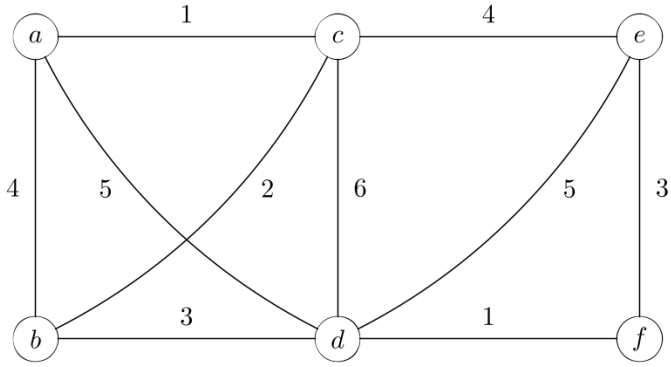


Figure 2: Graph G on which to run Prim's algorithm in Problem 2a.

2a (50 p) Consider the graph G in Figure 2, which is given to us in adjacency list representation, with the neighbour lists sorted so that vertices are encountered in lexicographic order. (For instance, the neighbour list of c is (a, b, d, e) sorted in that order.)

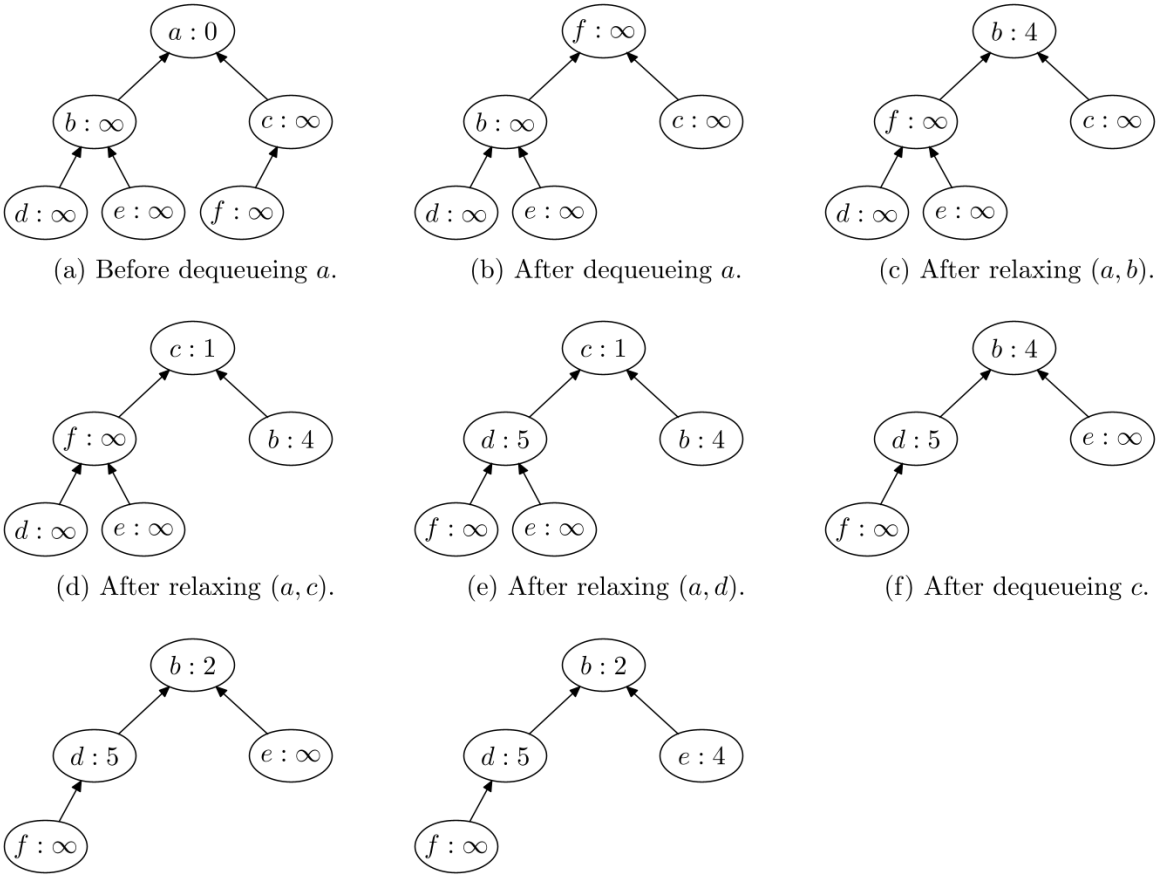
Run Prim's algorithm on the graph G, starting with the vertex a. Show the minimum spanning tree T produced at the end of the algorithm. Use a heap for the priority queue implementation. Assume that in the array representing the heap, the vertices are initially listed in lexicographic order. During the execution of the algorithm, describe for every vertex which neighbours are being considered and how they are dealt with. Show for the first two dequeued vertices how the heap changes after the dequeueing operations and all ensuing key value updates in the priority queue. For the rest of the vertices, you should describe how the algorithm considers all neighbours of the dequeued vertices and how key values are updated, but you do not need to draw any heaps.

Solution: We illustrate in Figure 3 how the heap used for the priority queue changes during the execution of Prim's algorithm. We use the notation $v : k$ in the heap when vertex v has key value k. At the outset, the vertex a has key 0 and all other vertices have key ∞ as in Figure 3a.

1. After a has been dequeued, vertex f is moved to the top of the heap and we have the configuration in Figure 3b. Relaxing the edge (a, b) gives value 4 to b (i.e., , the weight of the edge), and so shifts b to the top and pushes f down below b, yielding Figure 3c. Relaxing (a, c) decreases the key of c to 1 and so swaps b and c, yielding Figure 3d. Finally, relaxing (a, d) decreases the key of d to 5, which causes d to bubble up and f to bubble down, resulting in the heap in Figure 3e. There are no further outgoing edges from a to relax.

2. Since c is now at the top of the heap, it is the vertex dequeued next. This will add the edge (a, c) to the spanning tree, which we indicate in Figure 4. When c is removed, e is moved to the top. This violates the min-heap property, since the key of e is not smaller than or equal to those of its children. Since b has smaller key than d, we swap e and b. This restores the heap property (since the left subtree of the root was not changed, and e is now the root of a singleton subheap), and so the heap after removal of c looks as in Figure 3f.

   Relaxing the edge (c, b) decreases the key value of b to 2, which is the qeight of the edge. Since b is already the root, the heap does not change except for this key update and now looks like in Figure 3g. Relaxing (c, d) has no effect, since the key value is smaller than the



(a) Before dequeueing a. (b) After dequeueing a. (c) After relaxing (a, b).

(d) After relaxing (a, c). (e) After relaxing (a, d). (f) After dequeueing c.

(g) After relaxing (c, b). (h) After relaxing (c, e).

Figure 3: Heap configurations for priority queue in Prim's algorithm in Problem 2a.

edge weight. Relaxing (c, e) updates the key of e to 4 but does not change the structure of the heap, since the parent b of e has a smaller key (see Figure 3h). There are no further edges incident to c to relax. According to the instructions in the problem statement, we do not need to provide any further heap illustrations from this point on.

3. Since b is now the vertex with the smallest key value it is dequeued next, and the edge (c, b) is added to the spanning tree (since the latest update of the key value of b was when relaxing the edge (c, b)). The heap now has e at the root with children d and f. When we relax (b, d), the key of d becomes smaller than that of e, and so these two vertices swap places. Other than that, all neighbours of b have already been dequeued and there are no edges to relax.

4. Vertex d now has the smallest key 3 and is dequeued next. This adds the edge (b, d) to the spanning paths tree, since the key of d was last updated when (b, d) was relaxed. Relaxing (d, e) does not lead to any key decrease, but relaxing (d, f) updates the key of f to 1, meaning that e and f trade places in the heap.

5. Now vertex f has the smallest key 1 and so is dequeued, adding (d, f) to the spanning tree. When we relax (f, e) the key of e decreases to 3.

6. Finally, vertex e is dequeued, adding the just relaxed edge (f, e) to the spanning tree. The

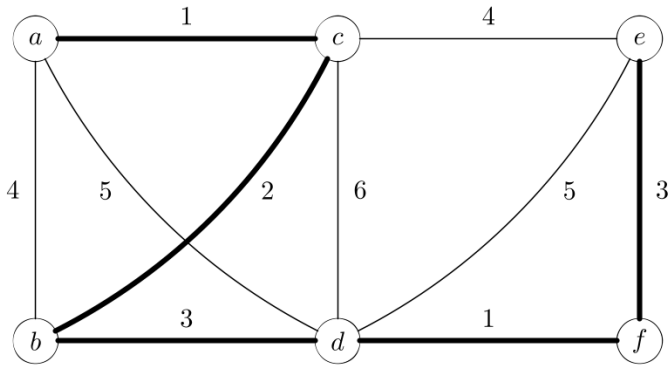


Figure 4: Graph G in Problem 2a with MST generated by Prim's algorithm.

queue is now empty, and there is nothing to relax.

The minimum spanning tree computed as described above is indicated by the bold edges in Figure 4.

2b (30 p) Jakob has designed a fairly nifty MST algorithm that works as follows for $G = (V, E)$:

1. Initialize $T = \emptyset$ and $S = \{v\}$, where v is a randomly chosen vertex of the graph.
2. Iterate $|V| - 1$ times:
   (a) Let w be the vertex most recently added to S.
   (b) Among all edges from w to vertices x not yet added to S, pick the edge (w, x) with lowest cost and add to T, and add x to S with predecessor w.
   (c) If w does not have any neighbours not already in S, go back to the predecessor of w, and then to the predecessor of the predecessor, et cetera, until you find a vertex z that has at least one neighbor not already in S. Use that vertex z instead of w in this iteration.

Jakob claims that T as computed by this algorithm is a minimum spanning tree, and that the algorithm is also faster than Prim's algorithm since it runs in time $O(|V| + |E|)$. Determine whether Jakob's algorithm is valid by giving a proof of correctness or a counter-example. Regardless of whether the algorithm is correct or not, did Jakob get the time complexity analysis right? Motivate your answer clearly.

Solution: No, unfortunately, for this problem Jakob is wrong about both correctness and time complexity.

To see that the algorithm is incorrect, consider a graph on vertices $\{1, 2, 3\}$ with edges $(1, 2)$ of weight 12, $(1, 3)$ of weight 13, and $(2, 3)$ of weight 23. Suppose the algorithm starts in vertex 1, i.e., it initializes $S = \{1\}$ (which is one valid random choice for which the algorithm would have to work). Then the algorithm will pick the edge $(1, 2)$, add vertex 2 to S, and move to this new vertex. From there, it will pick edge $(2, 3)$ and add it to the set, resulting in a spanning tree. But clearly this spanning tree does not have minimal weight, since the edge $(1, 3)$ should instead be chosen together with the edge $(1, 2)$ to obtain a minimum spanning tree.

Regarding the time complexity, in order to achieve time $O(|V| + |E|)$ we would need to argue that the processing cost of each edge is constant. But this is not clear at all—in the backtracking step (c), an edge that has already been considered before could be considered again, and it is not

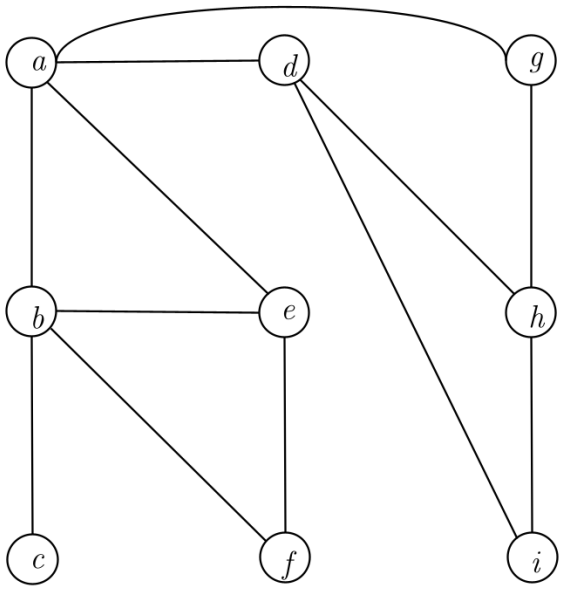


Figure 5: Undirected unweighted graph for graph traversals in Problem 3.

at all clear that edges would be reconsidered only a constant number of times. On the contrary, it is not too hard to cook up examples where at least some edges are revisited a linear number of times. Therefore, without carefully chosen data structures, the time complexity looks more like $O(|V| \cdot |E|)$. With the right data structures the running time could probably be improved quite a bit, but the complexity of the algorithm as described in the problem statement is definitely not $O(|V| + |E|)$.

3 (80 p) Consider the graph in Figure 5. 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.

3a (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 6 for an illustration of the algorithm execution described below with the resulting spanning tree edges drawn with (directed) edges in red.

1. We start in vertex a, which we mark as discovered at time 1. Since the neighbours in each adjacency list are sorted in lexicographic order, the 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 mark it as discovered at time 2, and also 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 mark it as discovered at time 3, and also 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 and c is marked as finished at time 4.

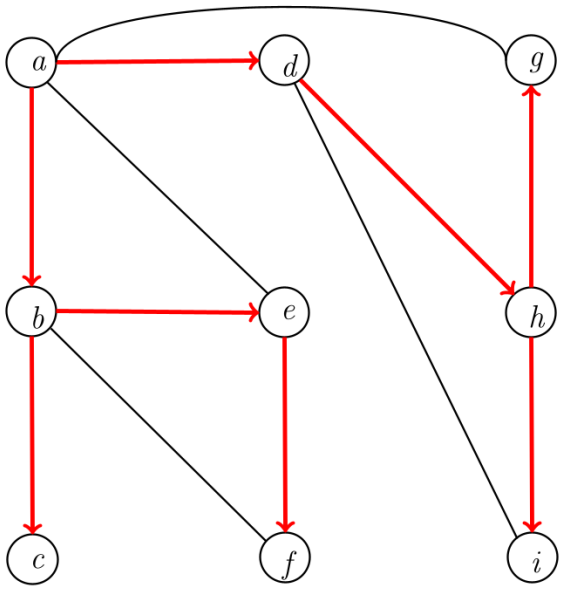


Figure 6: Depth-first-search tree for graph traversal in Problem 3 with (directed) edges in spanning tree in red.

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, mark e as discovered at time 5, and add the edge (b, e) to the spanning tree. The first neighbours of e are a and b, which are both already visited.

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 discovery time for f is 6. The neighbours b and e of f are both visited already, so the recursive call returns. and f is marked as finished at time 7.

6. We are now back in the call made for e, but since all neighbours have now been processed, this call returns and e is marked as finished at time 8.

7. This brings us back to the call made for b, and since all neighbours of b have also been processed, this call returns, marking b as finished at time 9.

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 discovery time for d is 10. 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 and marking h as discovered at time 11. 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 and marking g as discovered at time 12. The neighbours of g are a and h, which have both been visited, so the recursive call to g immediately returns and g is marked as finished at time 13.

11. The third and final neighbour of h is i, which we thus discover and visit at time 14, adding the edge (h, i) to the spanning tree.

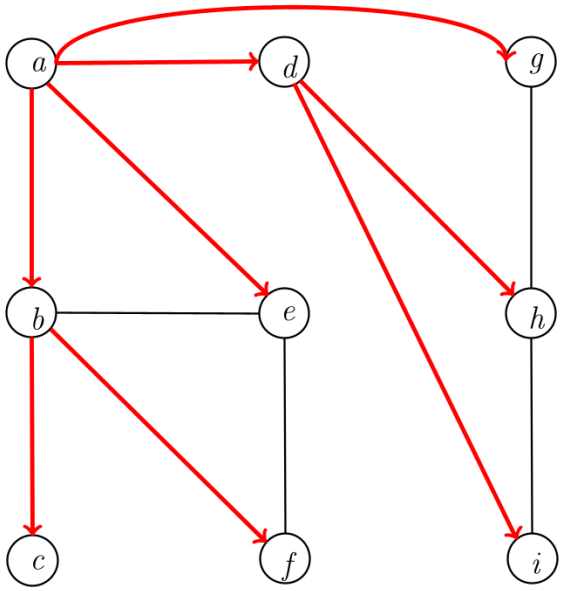


Figure 7: Breadth-first-search tree for graph traversal in Problem 3 with (directed) edges in spanning tree in red.

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. The finishing times are 15 for i, 16 for h, 17 for d, and 18 for a.

3b (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.

Solution: See Figure 7 for an illustration of the algorithm execution described below and the resulting spanning tree with edges again drawn in red.

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 (marking them as being at distance 1 from a), 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, and since b is at distance 1 from a, the newly discovered vertices c and f are marked as being at distance $1 + 1 = 2$ from a. 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, and h and i are marked as being at distance $1 + 1 = 2$ from a. 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.)

3c (20 p) Suppose that the graph in Figure 5 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.