sum $\sum_{j \in N} c_{\sigma^*(j)j} = C^*$. However, the same is true for the second term, the double summation is merely the sum over all possible locations j, and so the second terms contribute a total of $-\sum_{j \in N} c_{\sigma(j)j} = -C$. Now consider the second summation in (9.11). We can upper bound this expression by

$$\sum_{j:\sigma(j)=i} 2c_{\sigma^*(j)j}.$$

What is the effect of adding this summation over all crucial swaps? Each facility $i \in S$ occurs in 0, 1, or 2 crucial swaps; let n_i be the number of swaps in which each $i \in S$ occurs. Thus, we can upper bound the double summation as follows:

$$\sum_{i \in S} \sum_{j:\sigma(j)=i} 2n_i c_{\sigma^*(j)j} \leq 4 \sum_{i \in S} \sum_{j:\sigma(j)=i} c_{\sigma^*(j)j}.$$

But now we can apply the same reasoning as above; each location j is served in the current solution by a unique facility location $\sigma(j) \in S$, and hence the effect of the double summation is merely to sum over each j in N. That is, we have now deduced that this term is at most $4\sum_{j \in N} c_{\sigma^*(j)j} = 4C^*$. Furthermore, we have concluded that $0 \leq 5C^* - C$, and hence $C \leq 5C^*$. □

Finally, we observe that the same idea used in the previous section to obtain a polynomial-time algorithm can be applied here. The central ingredients to that proof are that if we restrict attention to moves (in this case swap moves) in which we improve the total cost by a factor of $1 - \delta$, then provided the analysis is based on a polynomial number of moves (each of which generates an inequality that the change in cost from this move is non-negative), we can set δ so that we can obtain a polynomial-time bound, while degrading the performance guarantee by an arbitrarily small constant.

Theorem 9.7: *For any constant $\rho > 5$, the local search algorithm for the k-median problem that uses bigger improving swaps yields a ρ-approximation algorithm.*

9.3 Minimum-degree spanning trees

In this section we return to the minimum-degree spanning tree problem introduced in Section 2.6. Recall that the problem is to find a spanning tree T in a graph $G = (V, E)$ that minimizes the maximum degree. If T^* is an optimal tree that minimizes the maximum degree, let OPT be the maximum degree of T^*. In Section 2.6, we showed that a particular local search algorithm finds a locally optimal tree of maximum degree at most $2\,\mathrm{OPT} + \lceil \log_2 n \rceil$ in polynomial time. In this section, we will show that another variation on the local search algorithm finds a locally optimal tree of maximum degree at most $\mathrm{OPT} + 1$ in polynomial time. As we discussed in Section 2.6, since it is NP-hard to minimize the maximum degree of a spanning tree, this is the best result possible unless P = NP.

As in the algorithm of Section 2.6, we start with an arbitrary spanning tree T and we will make local changes to it in order to decrease the degree of nodes in the tree. Let $d_T(u)$ be the degree of u in T. We pick a node u and attempt to reduce its degree by adding an edge (v, w) to T that creates a cycle C containing u, then removing an edge of the cycle C incident on u. Let $\Delta(T) = \max_{v \in V} d_T(v)$ be the maximum degree of the current tree T. We will make local changes in a way that is driven by the following lemma, which gives us a condition under which the current tree T has $\Delta(T) \leq \mathrm{OPT} + 1$.

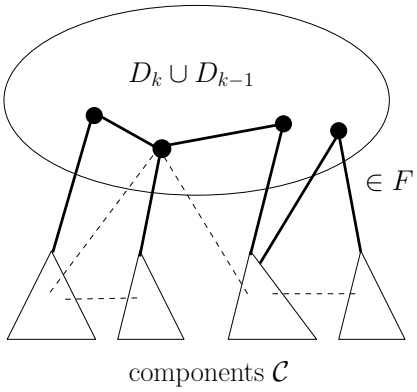

Figure 9.4: Illustration of the terms used in the statement of Lemma 9.8. The edges in F are shown in bold. Some edges in G that are not in the tree T are shown as dotted lines; note that they are not all incident on nodes in $D_k \cup D_{k-1}$. The components $\mathcal{C}$ are the components of the tree T remaining after the edges in F are removed.

Lemma 9.8: *Let $k = \Delta(T)$, let D_k be any nonempty subset of nodes of tree T with degree k and let D_{k-1} be any subset of nodes of tree T with degree $k-1$. Let F be the edges of T incident on nodes in $D_k \cup D_{k-1}$, and let $\mathcal{C}$ be the collection of $|F|+1$ connected components formed by removing the edges of F from T. If each edge of graph G that connects two different components in $\mathcal{C}$ has at least one endpoint in $D_k \cup D_{k-1}$, then $\Delta(T) \leq \mathrm{OPT} + 1$.*

Proof. See Figure 9.4 for an illustration of the terms. We use the same idea as in the proof of Theorem 2.19 to obtain a lower bound on OPT. Since any spanning tree in G will need $|F|$ edges of G to connect the components in $\mathcal{C}$, the average degree of the nodes in $D_k \cup D_{k-1}$ in any spanning tree is at least $|F|/|D_k \cup D_{k-1}|$. Thus $\mathrm{OPT} \geq \lceil |F|/|D_k \cup D_{k-1}| \rceil$.

We now bound $|F|$ in order to prove the lemma. The sum of the degrees of the nodes in D_k and D_{k-1} must be $|D_k|k + |D_{k-1}|(k-1)$. However, this sum of degrees may double count some edges of F which have both endpoints in $D_k \cup D_{k-1}$. Because T is acyclic, there can be at most $|D_k| + |D_{k-1}| - 1$ such edges. Hence, $|F| \geq |D_k|k + |D_{k-1}|(k-1) - (|D_k| + |D_{k-1}| - 1)$. Thus

$$\begin{aligned} \mathrm{OPT} &\geq \left\lceil \frac{|D_k|k + |D_{k-1}|(k-1) - (|D_k| + |D_{k-1}| - 1)}{|D_k| + |D_{k-1}|} \right\rceil \\ &\geq \left\lceil k - 1 - \frac{|D_{k-1}| - 1}{|D_k| + |D_{k-1}|} \right\rceil \\ &\geq k - 1, \end{aligned}$$

implying that $k = \Delta(T) \leq \mathrm{OPT} + 1$. □

The goal of the local search algorithm is to continue to reduce the degree of the nodes of degree $\Delta(T)$ while trying to attain the conditions of Lemma 9.8. The algorithm works in a sequence of phases, with each phase divided into subphases. At the beginning of the phase, for the current tree T, let $k = \Delta(T)$. At the beginning of a subphase, we let D_k be all the degree k vertices in T, let D_{k-1} be all the degree $k-1$ vertices in T, let F be the edges of T incident to $D_k \cup D_{k-1}$, and let $\mathcal{C}$ be the components of T formed if F is removed from T. The goal of each phase is to make local moves to remove all nodes of degree k from the tree;

the goal of each subphase is to make local moves to remove a single vertex of degree k. In the process of executing a subphase, we discover nodes of degree $k-1$ in D_{k-1} for which we can make a local move to reduce their degree. We do not yet execute these moves, but we mark the nodes as *reducible* via the particular local move, remove them from D_{k-1}, and update F and $\mathcal{C}$ accordingly by removing edges from F and merging components in $\mathcal{C}$. The algorithm is summarized in Algorithm 9.1, and we now discuss its details.

In a subphase, we consider all edges of G that connect any two components of $\mathcal{C}$. If all such edges have an endpoint in $D_k \cup D_{k-1}$, we meet the condition of Lemma 9.8, and the algorithm terminates with a tree T such that $\Delta(T) \leq \text{OPT} + 1$. If there is some such edge (v, w) without an endpoint in $D_k \cup D_{k-1}$, then consider the cycle formed by adding (v, w) to T. Because (v, w) connects two different components of $\mathcal{C}$, it must be the case that the cycle includes some node $u \in D_k \cup D_{k-1}$. Note that if we desire, we can make a local move to reduce the degree of u by adding (v, w) to the tree T and removing a tree edge incident to u. If the cycle contains nodes of D_{k-1} only, then we do not yet make the local move, but we make a note that for any of the nodes in D_{k-1} on the cycle, we could do so if necessary. We label these nodes in D_{k-1} on the cycle as reducible via the edge (v, w), then remove them from D_{k-1}, then update F to be the tree edges incident on the current set of $D_k \cup D_{k-1}$, and update $\mathcal{C}$ accordingly. Note that since we removed all nodes on the cycle from D_{k-1}, this only removes edges from F and merges components in $\mathcal{C}$; in particular, the two components connected by (v, w) will be merged in the updated $\mathcal{C}$. If, on the other hand, the cycle includes a node of $u \in D_k$, we go ahead and reduce its degree by adding (v, w) to the tree and removing an edge incident on u. Decreasing the degree of u decreases the number of nodes of degree k in the tree, but we want to ensure that we do not increase the degree of nodes v and w to k. Note that this could only happen if the degree of v or w in the tree is $k-1$ and at some previous point in the subphase the node was removed from D_{k-1} and labelled reducible. In this case, we carry out the local move that allows us to reduce the degree of reducible node to $k-2$, then add (v, w) to the tree and remove an edge incident to u from the tree. It is possible that carrying out the local move to reduce the degree of the reducible node to $k-2$ might cause a cascade of local moves; for instance, if v has degree $k-1$, and we can reduce its degree by adding edge (x, y), potentially x also has degree $k-1$ and is reducible, and so on; we will show that it is possible to carry out all these moves, and reduce the degree of u to $k-1$ without creating any new nodes of degree k. We say that we are able to *propagate* the local move for u. Once we reduce the degree of u from k to $k-1$, we start a new subphase. If there are no further nodes of degree k, we start a new phase.

We can now prove that the algorithm is correct and runs in polynomial time.

Theorem 9.9: *Algorithm 9.1 returns a spanning tree T with $\Delta(T) \leq \text{OPT} + 1$ in polynomial time.*

Proof. Because the algorithm terminates only when it meets the conditions of Lemma 9.8, it returns a tree T with $\Delta(T) \leq \text{OPT} + 1$ if it does indeed terminate. We claim that in each subphase, we can propagate local moves to reduce the degree of a reducible node in a component in $\mathcal{C}$ without creating any new nodes of degree k. Then either the algorithm terminates or in each subphase, we find some node u of degree k whose degree can be reduced to $k-1$ by making a local move with an edge (v, w). Since v and w must be in separate components of $\mathcal{C}$, either their degree is less than $k-1$ or they have degree $k-1$ and are reducible, and by the claim we can propagate local moves to reduce their degree. Thus we can reduce the degree of u from k to $k-1$ without creating any new nodes of degree k, and so each phase eliminates all nodes of degree k. Since we cannot have a feasible spanning tree with $\Delta(T) = 1$, the algorithm must

```
Let T be an arbitrary spanning tree of G = (V, E)
while true do
    k ← Δ(T)                                                    // Start a new phase
    while there are nodes of degree k in T do                   // Start a new subphase
        D_k ← all nodes of degree k in T
        D_{k-1} ← all nodes of degree k − 1 in T
        F ← all edges of T incident on nodes in D_k ∪ D_{k-1}
        C ← all components formed by removing F from T
        All nodes u ∈ D_{k-1} are unlabelled
        if for all (v, w) ∈ E connecting two components in C: either v or w in D_k ∪ D_{k-1}
        then
            return T
        for all (v, w) ∈ E connecting two components in C: v, w ∉ D_k ∪ D_{k-1} do
            Let C be cycle created by adding (v, w) to T
            if C ∩ D_k = ∅ then
                Mark all u ∈ C ∩ D_{k-1} reducible via (v, w)
                Remove C ∩ D_{k-1} from D_{k-1}
                Update F and C
            else
                if u ∈ C ∩ D_k then
                    if v or w marked reducible then
                        Reduce degree of v and/or w via local move and propagate local
                            moves if necessary
                    Reduce degree of u via local move with (v, w)
                    Break for loop                              // Start new subphase
```

Algorithm 9.1: Local search algorithm for the minimum-degree spanning tree problem.

eventually terminate. Clearly the algorithm runs in polynomial time.

We now prove the claim by showing that at any iteration of the subphase, we can propagate local moves to reduce the degree of a reducible node in a component in $\mathcal{C}$. We prove this by induction on the number of iterations in the subphase. In the first iteration, no nodes are marked reducible and the claim is trivially true. Now suppose we are at some iteration $i > 1$ of the subphase, and let u be labelled reducible in this iteration. The node u is currently reducible because we have a local move with a non-tree edge (v, w) that will reduce the degree of u from $k-1$ to $k-2$; furthermore, the components in $\mathcal{C}$ containing v and w are disjoint in the current iteration. If v is reducible, it was labelled such in an iteration $j < i$, and by induction we can carry out local moves to ensure that its degree is at most $k-2$. The same is true for w, and by induction we can carry out the local moves for both v and w because they are in separate components in $\mathcal{C}$. In the next iteration the components containing v and w are merged into a single component that also contains u. Since the only changes that can happen to components in $\mathcal{C}$ during a subphase is that components are merged, u, v, and w remain in the same component of $\mathcal{C}$ through the rest of the subphase, and the local moves of adding (v, w) and reducing the degree of u remain available. □

In Section 11.2, we will consider a version of the problem in which there are costs on the edges and specified bounds on the degrees of the nodes. If a tree with the given degree bounds