8

Sets

8.1 Representing a Set by a Bitmap

The topic of this chapter is different from that of the previous ones, in that a *set* is not a data structure, but rather a mathematical entity that appears frequently enough in our programs to raise the question of how to implement it efficiently. Generally, a set $\mathcal{S} = \{a_1, a_2, \ldots, a_n\}$ of n elements is defined, and one is interested in certain subsets of $\mathcal{S}$ and their interactions.

A straightforward representation of a subset is by means of a *bitmap*, also referred to as a bit-vector, which we have encountered already in the example of Section 3.4.1. A subset $C \subseteq \mathcal{S}$ will be represented by the bitmap $\mathcal{B}(C) = b_1 b_2 \cdots b_n$ of length n, in which $b_i = 1$ if and only if $a_i \in C$. For example, if $n = 10$ and $C = \{a_2, a_6, a_7\}$, then the corresponding bitmap is

$$\mathcal{B}(C) = 0100011000.$$

For the entire set, one gets $\mathcal{B}(\mathcal{S}) = 1111111111$ and for the empty set $\mathcal{B}(\emptyset) = 0000000000$. Intersection, union, and complementation are handled with the corresponding Boolean operators:

$$\mathcal{B}(X \cap Y) = \mathcal{B}(X) \wedge \mathcal{B}(Y),$$

$$\mathcal{B}(X \cup Y) = \mathcal{B}(X) \vee \mathcal{B}(Y),$$

$$\mathcal{B}(\overline{X}) = \overline{\mathcal{B}(X)},$$

where $\overline{X}$ denotes the complementing set $\mathcal{S} \setminus X$, $\overline{b} = 1 - b$ is the complementing bit of b, and the complement of a bit-vector is the vector of the bit-complements.

If the size of the set n is relatively small, it may fit into a computer word (32 bits) and all the operations can then be performed in a single step, but even for larger n, the necessary operations are simple loops and easily parallelized.

It will be convenient to prepare a set of *masks* $I_1, \ldots, I_n$, which are bitmaps representing the singletons $\{a_1\}, \ldots, \{a_n\}$, respectively, that is, $I_1 = 1000\cdots$, $I_2 = 0100\cdots$, $I_3 = 0010\cdots$, etc. Some of the basic operations can then be translated as follows:

add element a_r to the set X	$\mathcal{B}(X) \longleftarrow \mathcal{B}(X) \vee I_r$
delete a_r from X	$\mathcal{B}(X) \longleftarrow \mathcal{B}(X) \wedge \overline{I_r}$
check whether $a_r \in X$	if $(\mathcal{B}(X) \wedge I_r) = I_r$
check whether $Y \subseteq X$	if $(\mathcal{B}(X) \wedge \mathcal{B}(Y)) = \mathcal{B}(Y)$

This set representation is familiar to all users of the Unix operating system, in which the *access rights* to a given file are represented by a string of the form `-rwxrw-r-`, for example. This would mean that the owner may read, write and execute the file, other members of the group the owner belongs to may only read or write, and the rest of the users has only read access. In fact, the access rights may be written as a bitmap of length 10, since the interpretation of each bit position is fixed. For example, the third bit from the left indicates whether the owner has write permission. The preceding string would thus be 0111110100. Indeed, it is customary to set the access rights by the `chmod` command, using the octal representation of the 9 rightmost bits. For example the command setting the given string would be

```
chmod 764 filename.
```

Another instance of representing a set by a bitmap has been encountered in Section 3.2: the rows of the adjacency matrix of a graph give the sets of outgoing edges, whereas the columns correspond to the incoming edges for all the vertices.

8.1.1 Application: Full-Text Information Retrieval

Computers were initially just large machines performing calculations, and it took some time to realize that they can also be used to manipulate large texts. This might seem obvious for somebody growing up with internet access, but in the 1960s when large Information Retrieval Systems were built, this was a small revolution. The basic algorithms in these systems process various sets, which is why they are a good example for the topic of this chapter.

The general problem can be described as follows. In the background, a large corpus of natural language texts T is given and may be pre-processed. The system is then accessed by means of some *query* Q, and the task is to retrieve

all the passages that satisfy the query. For example, the query could be

$$Q = \text{information AND (processing OR retrieval)}, \tag{8.1}$$

and it could retrieve all the documents of T containing the first term and at least one of the other two. As extension, if the number of text passages satisfying Q is too large, one could be interested only in the most relevant results. To continue the example, one might restrict the retrieval to documents containing one of the *phrases* information processing or information retrieval, and not just happen to include the terms without any connection between them.

The algorithms to be used actually depend on the size of the underlying textual database. For small enough texts (and the definition of what *small* means clearly changes with time), a brute-force approach may be feasible, using a fast *string matching* algorithm, like the one of Boyer and Moore presented in the first chapter. For larger corpora, even a fast scan may take too long. The solution, called *inverted files,* is then to build the following auxiliary files.

- A *dictionary* D, containing the lexicographically sorted list of all the different terms in T; and
- A *concordance* C, containing, for each term $x \in D$, the sorted list $L(x)$ of the exact references of all occurrences of x in T. These references may be given as *coordinates*, which can be just document numbers, or when more fine grained queries are to be supported, the coordinates can be tuples, as (*document*, *paragraph*, *sentence*, *word*), or in any other hierarchical form.

The algorithm processing a query Q would then access the dictionary D with all the terms of Q and get there pointers to their coordinate lists in C. These lists have then to be merged or intersected, according to the Boolean structure induced by the query. For the example query, the sought list would be

$$L(\text{information}) \cap \big(L(\text{processing}) \cup L(\text{retrieval})\big).$$

In the case more precise metrical constraints are supported, the intersection is more involved. Suppose the query is $\mathsf{X}\ [\ell]\ \mathsf{X}'$, meaning that we are looking for occurrences of X and X' within the same sentence, but at most ℓ words apart, with $\ell = 1$ standing for adjacent terms. Then if $(d, p, s, w) \in L(X)$ and $(d', p', s', w') \in L(X')$, this pair of coordinates should be retrieved if it satisfies

$$d = d' \ \wedge \ p = p' \ \wedge \ s = s' \ \wedge \ |w - w'| \leq \ell.$$

For queries caring only about the document level, bitmaps acting as occurrence maps can be used, as explained in Section 3.4.1: the length of each bitmap will be the total number of documents in the system, and the bit in position i of

the bitmap $\mathcal{B}(A)$ of term A will be set to 1, if and only if the term A appears in document i. The query (8.1) would then be processed by evaluating

$$\mathcal{B}(\mathsf{information}) \wedge \big(\mathcal{B}(\mathsf{processing}) \vee \mathcal{B}(\mathsf{retrieval})\big),$$

and the list of document indices to be retrieved would just be the list of the indices of the 1-bits of the resulting bitmap.

8.2 Union-Find

In many applications involving sets, the scenario is the following. Given is a set $\mathcal{S} = \{x_1, \ldots, x_n\}$ of n elements and its partition into k disjoint subsets $S_1, \ldots, S_k$, that is,

$$\mathcal{S} = \bigcup_{i=1}^{k} S_i \qquad \text{and} \qquad S_i \cap S_j = \emptyset \quad \text{for} \quad i \neq j.$$

Two operations are to be supported:

(i) $\mathsf{Find}(x)$: given an element $x \in \mathcal{S}$, find the index of the (unique) subset $S_i \subseteq \mathcal{S}$ to which x belongs, i.e., $x \in S_i$;
(ii) $\mathsf{Union}(S_i, S_j)$: merge the subsets S_i and S_j into a single subset and return a pointer to the merged set.

If the partition is given and static throughout the process, one could use a simple array R of integers to implement the Find operation, by defining

$$R[i] = j \quad \text{if and only if} \quad x_i \in S_j \qquad \text{for } 1 \leq i \leq n,\ 1 \leq j \leq k.$$

This would enable a constant time evaluation of $\mathsf{Find}(x)$. A $\mathsf{Union}(i, j)$ request, however, would force a linear scan of the array, to change all i entries to j or vice versa, and would thus cost $\Omega(n)$.

To improve the time complexity of the Union command, one could think of representing each set S_i in the partition as a circular linked list with a sentinel element containing the index i of the subset. The lists can then be concatenated in time $O(1)$ as already mentioned in Section 2.4.1, but given an element x, one needs to follow pointers until getting to the sentinel element to retrieve the name of the set in a Find command, so the number of steps may be the length of the subset, which could be $\Omega(n)$.

This dilemma should remind a similar one of the beginning of Section 4.1, concerning the efficient implementation of a data structure supporting both efficient searches and updates. The compromise there was to pass to a tree structure, and a similar idea will be useful also for the present case.

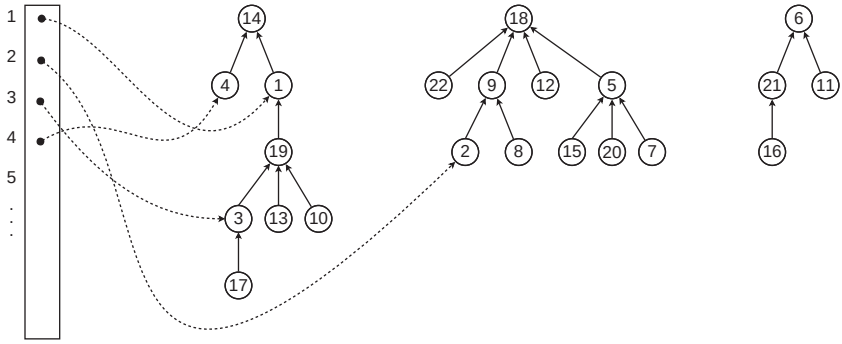

Figure 8.1. Representing disjoint sets by rooted trees.

8.2.1 Representing a Set by a Forest of Rooted Trees

Each element of the set $\mathcal{S}$ will be associated with a node in some tree. A set in the partition will be represented by a *rooted tree*, which is a tree in which every node has only one outgoing pointer, to its parent. Figure 8.1 displays three such rooted trees. Using again the notation v_x to design a node containing the value x, the root of the leftmost of the trees in Figure 8.1 is v_{14}, and it has two children, the leaf v_4 and v_1, which has a single child v_{19}. Note that the pointers, symbolized by the solid arrows, are pointing upward, so it is easy the reach the root from any node in the tree, but there is in fact no way of passing from a root to any of its offsprings.

We shall use the convention to name a set according to its root. The three trees in Figure 8.1 therefore represent the sets S_{14}, S_{18} and S_6. To implement the Find operation, an array of pointers is used, allowing direct access to each of the nodes. This array is shown on the left-hand side of the figure. For example, to process Find(x_3), the array is accessed at entry 3 that contains a pointer to the node v_3. From v_3, one follows then a series of parent pointers, passing through v_{19} and v_1, until reaching v_{14}, which is recognized as the root, because its outgoing pointer is NIL. The conclusion is that $x_3 \in S_{14}$. Similarly, one would get that $x_{20} \in S_{18}$ and that $x_6 \in S_6$, so there is no danger of confusing an element and the set it belongs to. The number of steps needed for Find(x) is thus related to the depth of x in its tree.

The merging of two sets for the Union(S_i, S_j) command is achieved in $O(1)$ time by letting one of the roots point to the other root. For example, for Union(S_{14}, S_{18}), one could set either

$$parent(v_{14}) \longleftarrow v_{18} \qquad \text{or} \qquad parent(v_{18}) \longleftarrow v_{14}. \tag{8.2}$$

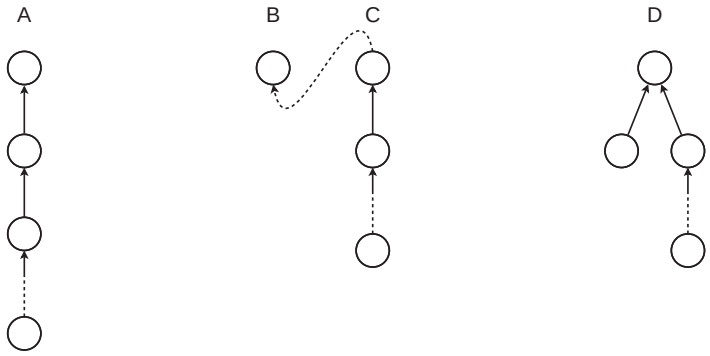

Figure 8.2. Worst case for Find.

Suppose the first possibility is chosen and that a Find(x_3) command is issued again. This time the pointer chain would not stop at v_{14}, but continue to the new root v_{18} of the merged set, resulting in the new conclusion that $x_3 \in S_{18}$.

It should be noted that we restrict our attention here only to the Union and Find commands, and shall not deal with other operations like inserting or deleting nodes into one of the trees. We may therefore assume that the given form of a tree has been achieved by a series of Union operations, where initially the partition of the set $\mathcal{S}$ was into n singletons $\{x_1\}, \ldots, \{x_n\}$. A second observation is that our discussion in terms of trees is for convenience only, and that the forest may well be represented simply by an array P of parent indices. For example, the array for the forest of Figure 8.1 would be

1	2	3	4	5	6	7	8	9	⋯	17	18	19	20	21	22
14	9	19	14	18	0	5	9	18	⋯	3	0	1	5	6	18

with the NIL pointer represented by the value 0, for example for $P[6]$.

So far, the solution suggested by these rooted trees does not seem satisfactory. While Union can be performed in $O(1)$, Find might require $\Omega(n)$, so this does not improve on our previous attempts. There is, however, still a degree of freedom that has not been exploited: there are two possible choices in (8.2) for merging two sets, and we might take advantage of that.

To see how the two ways of merging two trees can influence the depth of the merged tree, consider the tree labeled A in Figure 8.2, representing a tree of depth $m-1$ with m nodes, which is the worst possible form in this context. There is only one way of getting such a tree by Union operations: the last step must have been the merging of a single node, like the tree labeled B, with a tree of depth $m-2$, like the tree labeled C. Moreover, the Union must have been

performed by letting the root of C point to that of B, as indicated by the broken line arrow in the figure. Had we chosen to let B point to C, the resulting tree would be like the one labeled D, and not A.

This leads to the intuition that to avoid degenerated trees and thereby restrict the depth of the trees, one should prefer, in the Union steps, to let the smaller tree point to the larger one.

The question is then, how should one define which of two trees is the smaller one? Two plausible definitions come to mind: referring to the depth or to the number of nodes. That these are not equivalent can be seen by inspecting again the trees of Figure 8.1. The left tree S_{14} has 8 nodes and depth 4, whereas the middle tree S_{18} has 10, thus more nodes, but is only of depth 2. So how should we decide to perform the merge? Since the purpose is to restrict the depth of the trees, it seems natural to take the depth as criterion for smallness. We shall, however, do exactly the opposite.

The total number of steps to be performed depends obviously on which elements will be searched for. Lacking any reasonable model for these elements, we shall assume that all the nodes will appear once in a Find command, and use the total number of steps in these n Find operations as comparative measure.

Let us compare the two possibilities for merging the leftmost trees of Figure 8.1. If v_{18} will point to v_{14}, the depth of the merged tree will remain 4 as for S_{14}, a Find on any of the 8 nodes of S_{14} will take the same number of steps as it did before the merge, but a Find for any of the 10 nodes of S_{18} will require one more step than before, as the search will have to traverse the additional pointer from v_{18} to v_{14}. The increase caused by the merge is thus of 10 steps. On the other hand, if v_{14} will point to v_{18}, the depth of the merged tree will become 5, a Find on any of the 10 nodes of S_{18} will take the same number of steps as it did before, but a Find for any of the 8 nodes of S_{14} will require an additional step, giving together a total increase of just 8.

The simple explanation for this fact is that the depth of a tree may be determined by a single leaf on the lowest level, while the complexity of applying Find should depend on the levels of all the nodes, according to our model. The conclusion is that when applying Union, the smaller tree in terms of number of nodes should point to the larger one.

8.2.2 Depth of Trees in Union-Find Forests

Even though the criterion for deciding how to merge the trees is based on the number of their nodes, we are still interested in a bound on the depths of the resulting trees, since this is also a bound on the complexity of the Find operation. Fortunately, a precise bound can be derived.

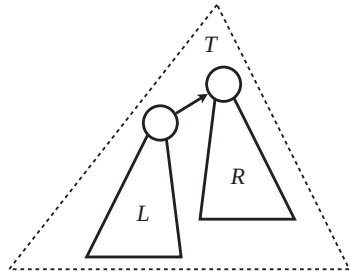

Figure 8.3. Schematic of inductive step.

Theorem 8.1. If the Union operations are implemented by letting the tree with the smaller number of nodes point to the tree with the larger number of nodes, then the depth of a tree with m nodes is at most $\lfloor \log_2 m \rfloor$.

To get a logarithmic depth might have been expected, but why should the base of the logarithm be 2? The $\log_2 n$ depth we encountered earlier was due to the fact that the trees were binary; m-ary trees, like the B-trees of Chapter 6, had a depth of $O(\log_m n)$. Our rooted trees are not binary, and in fact, there is no limit on the number of children a node can have, since no explicit pointer to the children is kept. So what is the explanation of the base 2 of the logarithm? We shall return to this question after the proof.

We saw already in Section 5.2 that a theorem with such a formulation might be hard to prove, which is why it will be convenient to formulate another, equivalent, theorem, by changing our point of view. Instead of assuming the number of nodes is given and deriving an upper bound on the depth, we assume that the depth is given and derive a lower bound on the number of nodes.

Theorem 8.1′. If the Union operations are implemented as described earlier, then the number of nodes in a tree of depth h is at least 2^h.

Proof By induction on the depth h. For each depth, let us consider the *minimal* tree, defined as a tree with smallest number of nodes among all trees attaining this depth. This is no real restriction, because if one shows a lower bound on the number of nodes for a minimal tree, this bound holds even more so for nonminimal trees of the same depth.

For $h = 0$, the only possible tree consists of the root alone, like the tree labeled B in Figure 8.2. Indeed, it has $2^0 = 1$ node. This is enough to prove the basis of the induction. Just for the exercise, let us also check the case $h = 1$. A tree of depth 1 must have at least one node on level 1, and one (the root) on level 0, like the leftmost tree in Figure 4.3. The number of its nodes is $2^1 = 2$.

Assume now the truth of the statement up to depth $h-1$ and let us show it for h. Let T be a minimal tree of depth h. Since any tree is the result of a Union of two other trees, let us call L and R the two trees merged to yield T. More precisely, we assume that as a result of the Union, the root of L points to that of R. as depicted in Figure 8.3.

The depth of T being h, this can be obtained either if R itself is already of depth h, or if L has depth $h-1$. The first possibility has to be ruled out: R being a subtree of T, it has obviously less nodes, contradicting the assumption that T is minimal. It follows that the depth of L must be $h-1$, so the inductive assumption can be applied to yield $|L| \geq 2^{h-1}$, where $|X|$ denotes the number of nodes in a tree X.

What is the depth of R? If it were $h-1$, we could reach the desired conclusion, but R can be much more shallow. Nevertheless, we can bound its size, not by the inductive hypothesis, but by the rules of the Union algorithm. Since L points to R, the latter must have at least as many nodes as the former.

Summarizing, given any tree G of depth h, we get

$$|G| \geq |T| = |L| + |R| \geq |L| + |L| \geq 2^{h-1} + 2^{h-1} = 2^h, \qquad (8.3)$$

where the first inequality is due to the minimality of T, the second to the rules of Union, and the third to the inductive assumption. ∎

The proof also solves the riddle posed earlier about the base 2 of the logarithm. Although the unified trees are not necessarily binary, the Union operation itself is a binary operation in the sense that it is always applied on a pair of trees. This implied the doubling of the number of nodes with each new level, as shown in eq. (8.3).

Note that a consequence of the theorem is that not all the shapes of trees are possible. In particular, the leftmost tree of Figure 8.1 cannot be obtained by the given Union commands, because with depth 4, it would need at least 16 nodes. The smallest tree of depth 2 cannot consist just of a chain of 3 nodes, as tree A of Figure 8.2, but needs at least 4 nodes, as the rightmost tree in Figure 8.1.

There is still an open question: how do we know the size of each tree? A tree is identified by its root, but there are no outgoing pointers from there. Even if there were, it would not be reasonable to evaluate the size on the fly. The solution is to keep track of the sizes of the trees from the beginning. A field $size(x)$ will be added to each node, indicating the number of nodes (including itself) in the (sub)tree rooted by x. Initially, all the sets are singletons and all sizes are 1. In a Union, when x will be set to point to y, the only node for which the size field needs an update is the root y of the unified tree, for which

$$size(y) = size(y) + size(x).$$

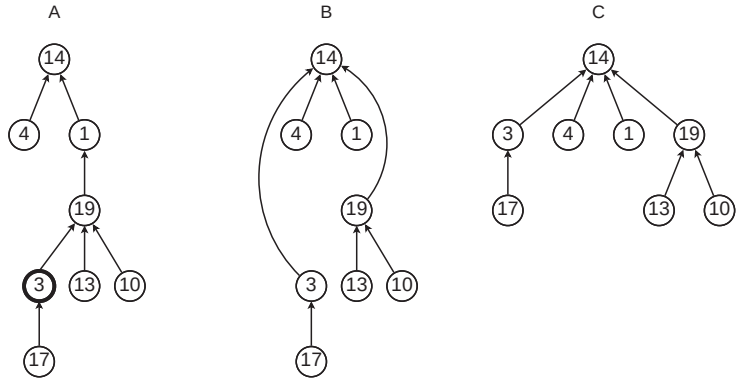

Figure 8.4. Path compression.

8.2.3 Path Compression

The performance of the Find can be improved at almost no cost with the following simple amendment, known as *path compression*. Consider the tree labeled A of Figure 8.4 and suppose the command to be executed is Find(x_3) as before, which leads us to the emphasized node in the tree. The algorithm inspects then all the nodes on the path from v_3 to the root v_{14}. If there is subsequently another Find(x_3) command, does it make sense to traverse the same path again, including v_{19} and v_1?

In binary search trees we would like to have as many nodes as possible close to the root, but the space there is limited; hence, most of the nodes have to be at logarithmic depth, which implies a logarithmic search time. However for the special trees of this chapter, the shape of the tree is not useful in itself, but rather a byproduct of the way a Union is performed. From the point of view of optimizing the Find, it would be preferable if all the elements in a tree would directly point to the root, which is possible since there are no limitations on incoming edges.

The idea is thus to take advantage of the fact that the Find command visits a chain of nodes, by letting all these nodes ultimately point to the root of the tree. The tree labeled B of Figure 8.4 shows the modified tree, in which v_3 and v_{19} point directly to the root v_{14} (v_1 is also on the path, but its parent pointer was already to the root). The tree labeled C is the same tree, but redrawn in the more standard way to emphasize how path compression will result in a tendency of the trees to get flatter.

For any Find(v) command, there is hardly any loss caused by the additional pointer updates, but there is a large potential gain, not only for the nodes on the

path from v to the root, but also for all the nodes in the subtrees rooted at any of the nodes on this path. For the preceding example, if one searches for v_{17} after the search for v_3, the root will be reached by 2 parent-pointers, instead of 4 without path compression.

Evaluating the depth of the trees for this case is not a simple task. It clearly depends on which elements have participated in a Find command, so the worst case did not change. On the other hand, in the long run, all the trees might end up with depth 1, if there are enough Find queries. We give here the complexity of the procedure without proof or details: using an appropriate model, the *average* cost of a Find on a set of n elements will be $O(\log^* n)$. The function $\log^*$ (also called iterative logarithm or log star) is the number of times the function $\log_2$ can be applied iteratively until the result is ≤ 1. For example, reading right to left,

$$0.54 = \log_2(1.45 = \log_2(2.73 = \log_2(6.64 = \log_2(100)))),$$

thus $\log^*(100) = 4$. This function grows extremely slowly: the lowest argument for which $\log^*$ will be 5 is $2^{16} = 65536$, and it will stay at value 5 for all reasonable numbers (up to 2^{65536}). For any practical use, we can thus consider $\log^* n$ as being a constant, even though theoretically, it tends to ∞.

The following table summarizes the complexities of applying a single Union-Find operation on a set of n elements. The values correspond to worst cases, except the Find in the last line:

	Union	Find	
Simple array	$O(n)$	$O(1)$	
Linear linked list	$O(1)$	$O(n)$	
Rooted trees	$O(1)$	$O(n)$	
Smaller tree pointing to larger	$O(1)$	$O(\log n)$	
With path compression	$O(1)$	$O(\log^* n)$	average

8.2.4 Formal Algorithms

The formal Union-Find algorithms, including all the variants mentioned earlier, are given in Figure 8.5.

It uses the array implementation of the trees, so the parameters are indices in $\{1, \ldots, n\}$ rather than nodes or sets. The field $size(x)$ of a node v_x is implemented here as an additional array $size[]$. The first while loop in Find reaches the root of the tree. At the end of this loop, the index of the root is stored in y. The second loop implements path compression and traverses again the path from v_x to v_y, letting each node now point directly to the root.

```
Union(v, w)
    if size[v] < size[w] then
        P[v] ⟵ w
        size[w] ⟵ size[w] + size[v]
        return w
    else
        P[w] ⟵ v
        size[v] ⟵ size[v] + size[w]
        return v
```

```
Find(x)
    y ⟵ x
    while P[y] ≠ 0 do
        y ⟵ P[y]
    while P[x] ≠ y do
        z ⟵ x
        x ⟵ P[x]
        P[z] ⟵ y
    return y
```

Figure 8.5. Formal Union-Find algorithms.

Exercises

8.1 The game of *Hex* is played on a honeycomb like board, representing a 2-dimensional matrix of $n \times n$ hexagonal cells. Two cells are considered adjacent if the hexagons share a common side. The players B and W alternate to pick at each turn one of the free (gray) cells, which is then colored black, respectively white. The winner is the first to build a sequence of adjacent cells of the same color that connects opposing edges of the matrix, for example NW to SE for B, or NE to SW for W. A possible situation after 9 turns for each of B and W is shown in Figure 8.6.

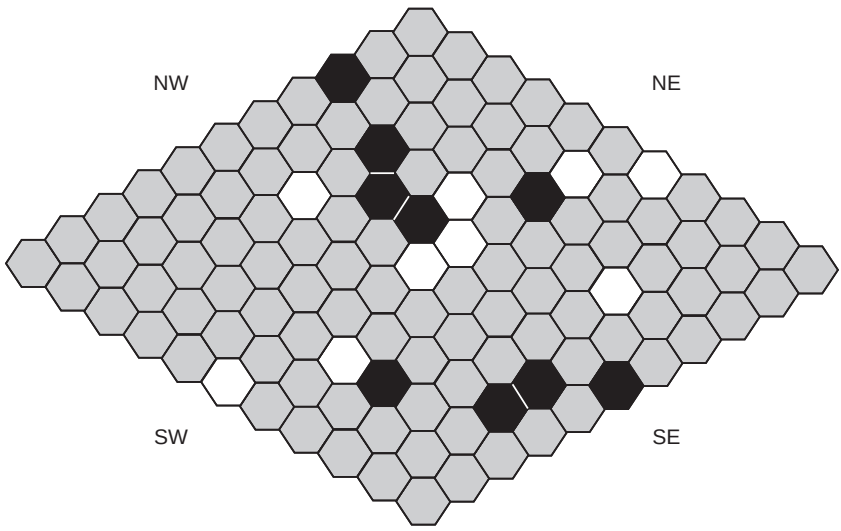

Figure 8.6. The game of Hex, with $n = 11$.

Show how to implement a program that checks after each turn whether the last move was a winning one.

8.2 The Union algorithm sets the smaller tree as a subtree of the larger one, where *small* and *large* are defined in terms of the number of nodes. If instead *small* and *large* are defined in terms of the depths of the trees, how will this affect the bound on the depths of all the trees in the forest?

8.3 The Minimum Spanning Tree problem for a graph $G = (V, E)$ with weights $w(x, y)$ on its edges $(x, y) \in E$ has been presented in Section 3.1.1. One of the algorithms solving it is due to J. B. Kruskal. The edges are first sorted into nonincreasing order according to their weights, and the tree T is initially empty. The edges are then considered in the given order, and an edge is adjoined to T unless it closes a cycle. The algorithm stops when there are $|V| - 1$ edges in T.

(a) What is the complexity of Kruskal's algorithm when BFS or DFS is used to check whether an edge closes a cycle?
(b) What is the complexity when Union-Find is used instead?
(c) What is the connection between the trees in the Union-Find forest and those forming the connected components of T at each stage of Kruskal's algorithm?

8.4 The expected depth of the trees when path compression is used has been stated previously to be bounded by $\log^* n$. Another analysis shows that the bound is $\alpha(n) = A^{-1}(n, n)$, which is known as the *Inverse-Ackermann* function, which also grows extremely slowly. One of the definitions of Ackermann's function is

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0, \\ A(m - 1, 1) & \text{if } m > 0 \ \wedge \ n = 0, \\ A(m - 1, A(m, n - 1)) & \text{otherwise.} \end{cases}$$

Show that the elements $A(n, n)$ for $n = 0, 1, 2, 3,$ are 1, 3, 7, 61, respectively. The next element in the sequence would already be $A(4, 4) = 2^{2^{65636}} - 3$.