V

Miscellaneous Data Structures

28

Tries

Sartaj Sahni
University of Florida

28.1 What Is a Trie?

A *trie* (pronounced "try" and derived from the word re*trie*val) is a data structure that uses the digits in the keys to organize and search the dictionary. Although, in practice, we can use any radix to decompose the keys into digits, in our examples, we shall choose our radixes so that the digits are natural entities such as decimal digits $(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)$ and letters of the English alphabet $(a - z, A - Z)$.

Suppose that the elements in our dictionary are student records that contain fields such as student name, major, date of birth, and social security number (SS#). The key field is the social security number, which is a nine digit decimal number. To keep the example manageable, assume that the dictionary has only five elements. Table 28.1 shows the name and SS# fields for each of the five elements in our dictionary.

To obtain a trie representation for these five elements, we first select a radix that will be used to decompose each key into digits. If we use the radix 10, the decomposed digits are

Name	Social Security Number (SS#)
Jack	951-94-1654
Jill	562-44-2169
Bill	271-16-3624
Kathy	278-49-1515
April	951-23-7625

TABLE 28.1 Five elements (student records) in a dictionary

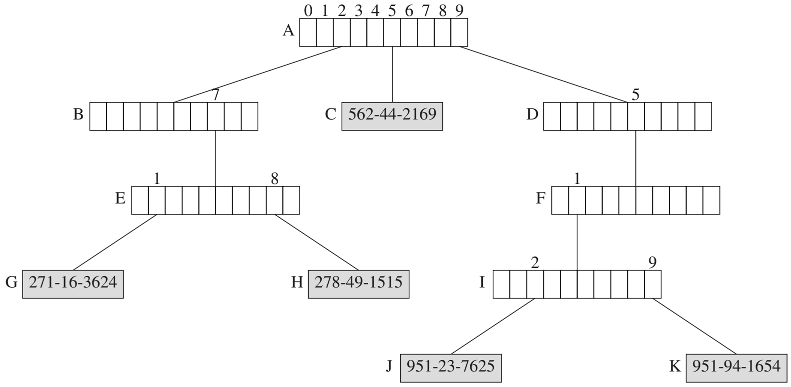


FIGURE 28.1: Trie for the elements of Table 28.1.

just the decimal digits shown in Table 28.1. We shall examine the digits of the key field (i.e., SS#) from left to right. Using the first digit of the SS#, we partition the elements into three groups–elements whose SS# begins with 2 (i.e., Bill and Kathy), those that begin with 5 (i.e., Jill), and those that begin with 9 (i.e., April and Jack). Groups with more than one element are partitioned using the next digit in the key. This partitioning process is continued until every group has exactly one element in it.

The partitioning process described above naturally results in a tree structure that has 10-way branching as is shown in Figure 28.1. The tree employs two types of nodes–*branch nodes* and *element nodes*. Each branch node has 10 children (or pointer/reference) fields. These fields, $child[0:9]$, have been labeled $0, 1, \cdots, 9$ for the root node of Figure 28.1. $root.child[i]$ points to the root of a subtrie that contains all elements whose first digit is i. In Figure 28.1, nodes A, B, D, E, F, and I are branch nodes. The remaining nodes, nodes C, G, H, J, and K are element nodes. Each element node contains exactly one element of the dictionary. In Figure 28.1, only the key field of each element is shown in the element nodes.

28.2 Searching a Trie

To search a trie for an element with a given key, we start at the root and follow a path down the trie until we either fall off the trie (i.e., we follow a *null* pointer in a branch node) or we reach an element node. The path we follow is determined by the digits of the search key. Consider the trie of Figure 28.1. Suppose we are to search for an element with key 951-23-7625. We use the first digit, 9, in the key to move from the root node A to the node $A.child[9] = D$. Since D is a branch node, we use the next digit, 5, of the key to move further down the trie. The node we reach is $D.child[5] = F$. To move to the next level of the trie, we use the next digit, 1, of the key. This move gets us to the node $F.child[1] = I$. Once again, we are at a branch node and must move further down the trie. For this move, we use the next digit, 2, of the key, and we reach the element node $I.child[2] = J$. When an element node is reached, we compare the search key and the key of the element in the reached element node. Performing this comparison at node J, we get a match. The element in node J, is to be returned as the result of the search.

When searching the trie of Figure 28.1 for an element with key 951-23-1669, we follow the same path as for the key 951-23-7625. The key comparison made at node J tells us that the trie has no element with key 951-23-1669, and the search returns the value *null*.

To search for the element with key 562-44-2169, we begin at the root A and use the first digit, 5, of the search key to reach the element node $A.\mathit{child}[5] = C$. The key of the element in node C is compared with the search key. Since the two keys agree, the element in node C is returned.

When searching for an element with key 273-11-1341, we follow the path $A, A.\mathit{child}[2] = B, B.\mathit{child}[7] = E, E.\mathit{child}[3] = \mathit{null}$. Since we fall off the trie, we know that the trie contains no element whose key is 273-11-1341.

When analyzing the complexity of trie operations, we make the assumption that we can obtain the next digit of a key in $O(1)$ time. Under this assumption, we can search a trie for an element with a d digit key in $O(d)$ time.

28.3 Keys with Different Length

In the example of Figure 28.1, all keys have the same number of digits (i.e., 9). In many applications, however, different keys have different length. This does not pose a problem unless one key is a prefix of another (for example, 27 is a prefix of 276). For applications in which one key may be a prefix of another, we normally add a special digit (say #) at the end of each key. Doing this ensures that no key is a prefix of another.

To see why we cannot permit a key that is a prefix of another key, consider the example of Figure 28.1. Suppose we are to search for an element with the key 27. Using the search strategy just described, we reach the branch node E. What do we do now? There is no next digit in the search key that can be used to reach the terminating condition (i.e., you either fall off the trie or reach an element node) for downward moves. To resolve this problem, we add the special digit # at the end of each key and also increase the number of children fields in an element node by one. The additional child field is used when the next digit equals #.

An alternative to adding a special digit at the end of each key is to give each node a *data* field that is used to store the element (if any) whose key exhausts at that node. So, for example, the element whose key is 27 can be stored in node E of Figure 28.1. When this alternative is used, the search strategy is modified so that when the digits of the search key are exhausted, we examine the *data* field of the reached node. If this *data* field is empty, we have no element whose key equals the search key. Otherwise, the desired element is in this *data* field.

It is important to note that in applications that have different length keys with the property that no key is a prefix of another, neither of just mentioned strategies is needed; the scheme described in Section 28.2 works as is.

28.4 Height of a Trie

In the worst case, a root-node to element-node path has a branch node for every digit in a key. Therefore, the height[*] of a trie is at most $\mathit{numberofdigits} + 1$.

A trie for social security numbers has a height that is at most 10. If we assume that it takes the same time to move down one level of a trie as it does to move down one level of a binary search tree, then with at most 10 moves we can search a social-security trie. With this

[*] The definition of height used in this chapter is: the height of a trie equals the number of levels in that trie.

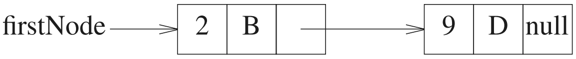


FIGURE 28.2: Chain for node A of Figure 28.1.

many moves, we can search a binary search tree that has at most $2^{10} - 1 = 1023$ elements. This means that, we expect searches in the social security trie to be faster than searches in a binary search tree (for student records) whenever the number of student records is more than 1023. The breakeven point will actually be less than 1023 because we will normally not be able to construct full or complete binary search trees for our element collection.

Since a SS# is nine digits, a social security trie can have up to 10^9 elements in it. An AVL tree with 10^9 elements can have a height that is as much as (approximately) $1.44 \log_2(10^9 + 2) = 44$. Therefore, it could take us four times as much time to search for elements when we organize our student-record dictionary as an AVL tree than when this dictionary is organized as a trie!

28.5 Space Required and Alternative Node Structures

The use of branch nodes that have as many child fields as the radix of the digits (or one more than this radix when different keys may have different length) results in a fast search algorithm. However, this node structure is often wasteful of space because many of the child fields are *null*. A radix r trie for d digit keys requires $O(rdn)$ child fields, where n is the number of elements in the trie. To see this, notice that in a d digit trie with n information nodes, each information node may have at most d ancestors, each of which is a branch node. Therefore, the number of branch nodes is at most dn. (Actually, we cannot have this many branch nodes, because the information nodes have common ancestors like the root node.)

We can reduce the space requirements, at the expense of increased search time, by changing the node structure. For example, each branch node of a trie could be replaced by any of the following:

1. A chain of nodes, each node having the three fields $digitValue, child, next$. Node A of Figure 28.1, for example, would be replaced by the chain shown in Figure 28.2.
   The space required by a branch node changes from that required for r children/pointer/reference fields to that required for $2p$ pointer fields and p digit value fields, where p is the number of children fields in the branch node that are not *null*. Under the assumption that pointer fields and digit value fields are of the same size, a reduction in space is realized when more than two-thirds of the children fields in branch nodes are *null*. In the worst case, almost all the branch nodes have only 1 field that is not *null* and the space savings become almost $(1 - 3/r) * 100\%$.
2. A (balanced) binary search tree in which each node has a digit value and a pointer to the subtrie for that digit value. Figure 28.3 shows the binary search tree for node A of Figure 28.1.
   Under the assumption that digit values and pointers take the same amount of space, the binary search tree representation requires space for $4p$ fields per branch node, because each search tree node has fields for a digit value, a subtrie pointer, a left child pointer, and a right child pointer. The binary search tree representation of a branch node saves us space when more than three-fourths of the children fields in branch nodes are *null*. Note that for large r, the binary search tree is

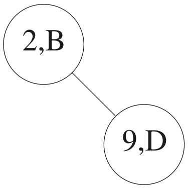


FIGURE 28.3: Binary search tree for node A of Figure 28.1.

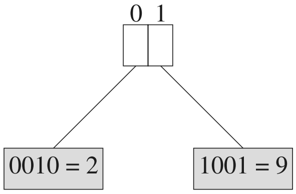


FIGURE 28.4: Binary trie for node A of Figure 28.1.

Node	A	B	C	D	E	F	G	H	I	J	K
Number	10	11	0	12	13	14	1	2	15	3	4

FIGURE 28.5: Number assignment to nodes of trie of Figure 28.1.

faster to search than the chain described above.

3. A binary trie (i.e., a trie with radix 2). Figure 28.4 shows the binary trie for node A of Figure 28.1. The space required by a branch node represented as a binary trie is at most $(2 * \lceil \log_2 r \rceil + 1)p$.
4. A hash table. When a hash table with a sufficiently small loading density is used, the expected time performance is about the same as when the node structure of Figure 1 is used. Since we expect the fraction of *null* child fields in a branch node to vary from node to node and also to increase as we go down the trie, maximum space efficiency is obtained by consolidating all of the branch nodes into a single hash table. To accomplish this, each node in the trie is assigned a number, and each parent to child pointer is replaced by a triple of the form $(currentNode, digitValue, childNode)$. The numbering scheme for nodes is chosen so as to easily distinguish between branch and information nodes. For example, if we expect to have at most 100 elements in the trie at any time, the numbers 0 through 99 are reserved for information nodes and the numbers 100 on up are used for branch nodes. The information nodes are themselves represented as an array $information[100]$. (An alternative scheme is to represent pointers as tuples of the form $(currentNode, digitValue, childNode, childNodeIsBranchNode)$, where $childNodeIsBranchNode = true$ iff the child node is a branch node.)

   Suppose that the nodes of the trie of Figure 28.1 are assigned numbers as given in Figure 28.5. This number assignment assumes that the trie will have no more than 10 elements.

   The pointers in node A are represented by the tuples $(10, 2, 11), (10, 5, 0)$, and $(10, 9, 12)$. The pointers in node E are represented by the tuples $(13, 1, 1)$ and $(13, 8, 2)$.

   The pointer triples are stored in a hash table using the first two fields (i.e., the

$currentNode$ and $digitValue$) as the key. For this purpose, we may transform the two field key into an integer using the formula $currentNode * r + digitValue$, where r is the trie radix, and use the division method to hash the transformed key into a home bucket. The data presently in information node i is stored in $information[i]$.

To see how all this works, suppose we have set up the trie of Figure 28.1 using the hash table scheme just described. Consider searching for an element with key 278-49-1515. We begin with the knowledge that the root node is assigned the number 10. Since the first digit of the search key is 2, we query our hash table for a pointer triple with key $(10, 2)$. The hash table search is successful and the triple $(10, 2, 11)$ is retrieved. The $childNode$ component of this triple is 11, and since all information nodes have a number 9 or less, the child node is determined to be a branch node. We make a move to the branch node 11. To move to the next level of the trie, we use the second digit 7 of the search key. For the move, we query the hash table for a pointer with key $(11, 7)$. Once again, the search is successful and the triple $(11, 7, 13)$ is retrieved. The next query to the hash table is for a triple with key $(13, 8)$. This time, we obtain the triple $(13, 8, 2)$. Since $childNode = 2 < 10$, we know that the pointer gets us to an information node. So, we compare the search key with the key of the element $information[2]$. The keys match, and we have found the element we were looking for.

When searching for an element with key 322-167-8976, the first query is for a triple with key $(10, 3)$. The hash table has no triple with this key, and we conclude that the trie has no element whose key equals the search key.

The space needed for each pointer triple is about the same as that needed for each node in the chain of nodes representation of a trie node. Therefore, if we use a linear open addressed hash table with a loading density of α, the hash table scheme will take approximately $(1/\alpha - 1) * 100\%$ more space than required by the chain of nodes scheme. However, when the hash table scheme is used, we can retrieve a pointer in $O(1)$ expected time, whereas the time to retrieve a pointer using the chain of nodes scheme is $O(r)$. When the (balanced) binary search tree or binary trie schemes are used, it takes $O(\log r)$ time to retrieve a pointer. For large radixes, the hash table scheme provides significant space saving over the scheme of Figure 28.1 and results in a small constant factor degradation in the expected time required to perform a search.

The hash table scheme actually reduces the expected time to insert elements into a trie, because when the node structure of Figure 28.1 is used, we must spend $O(r)$ time to initialize each new branch node (see the description of the insert operation below). However, when a hash table is used, the insertion time is independent of the trie radix.

To support the removal of elements from a trie represented as a hash table, we must be able to reuse information nodes. This reuse is accomplished by setting up an available space list of information nodes that are currently not in use.

Andersson and Nilsson [1] propose a trie representation in which nodes have a variable degree. Their data structure, called LC-tries (level-compressed tries), is obtained from a binary trie by replacing full subtries of the binary trie by single node whose degree is 2^i, where i is the number of levels in the replaced full subtrie. This replacement is done by examining the binary trie from top to bottom (i.e., from root to leaves).

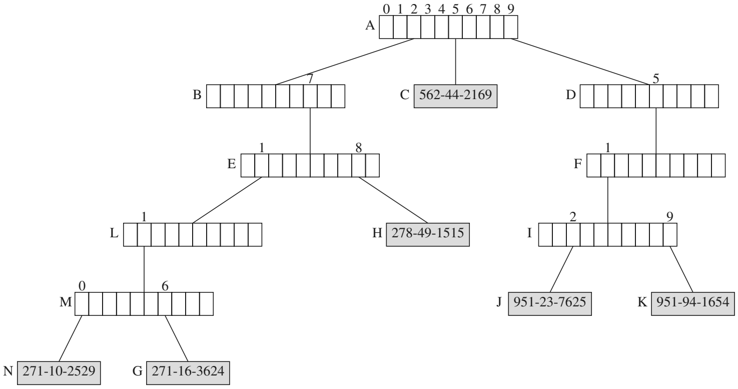


FIGURE 28.6: Trie of Figure 28.1 with 271-10-2529 inserted.

28.6 Inserting into a Trie

To insert an element *theElement* whose key is *theKey*, we first search the trie for an existing element with this key. If the trie contains such an element, then we replace the existing element with *theElement*. When the trie contains no element whose key equals *theKey*, *theElement* is inserted into the trie using the following procedure.

Case 1 For Insert Procedure

If the search for *theKey* ended at an element node X, then the key of the element in X and *theKey* are used to construct a subtrie to replace X.

Suppose we are to insert an element with key 271-10-2529 into the trie of Figure 28.1. The search for the key 271-10-2529 terminates at node G and we determine that the key, 271-16-3624, of the element in node G is not equal to the key of the element to be inserted. Since the first three digits of the keys are used to get as far as node E of the trie, we set up branch nodes for the fourth digit (from the left) onwards until we reach the first digit at which the two keys differ. This results in branch nodes for the fourth and fifth digits followed by element nodes for each of the two elements. Figure 28.6 shows the resulting trie.

Case 2 For Insert Procedure

If the search for *theKey* ends by falling off the trie from the branch node X, then we simply add a child (which is an element node) to the node X. The added element node contains *theElement*.

Suppose we are to insert an element with key 987-33-1122 to the trie of Figure 28.1. The search for an element with key equal to 987-33-1122 ends when we fall off the trie while following the pointer $D.child[8]$. We replace the *null* pointer $D.child[8]$ with a pointer to a new element node that contains *theElement*, as is shown in Figure 28.7.

The time required to insert an element with a d digit key into a radix r trie is $O(dr)$ because the insertion may require us to create $O(d)$ branch nodes and it takes $O(r)$ time to initialize the children pointers in a branch node.

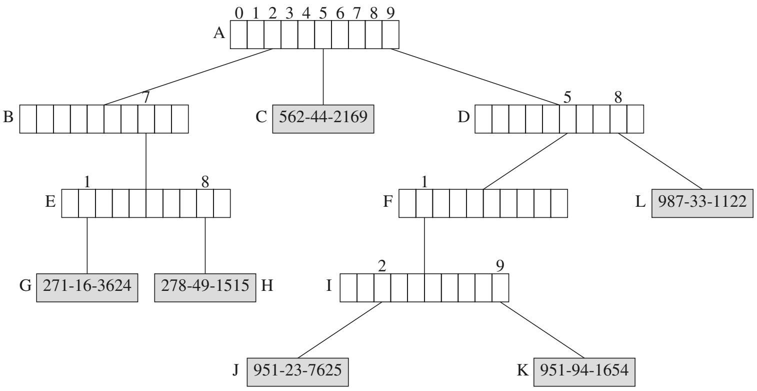


FIGURE 28.7: Trie of Figure 28.1 with 987-33-1122 inserted.

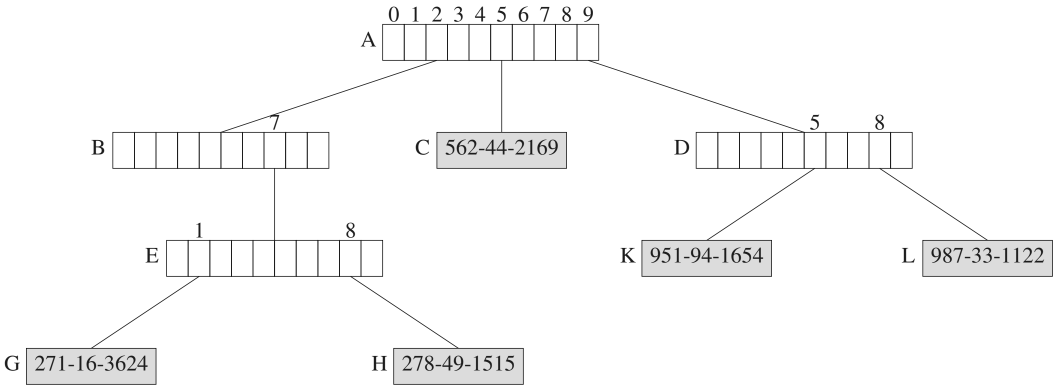


FIGURE 28.8: Trie of Figure 28.7 with 951-23-7635 removed.

28.7 Removing an Element

To remove the element whose key is *theKey*, we first search for the element with this key. If there is no matching element in the trie, nothing is to be done. So, assume that the trie contains an element *theElement* whose key is *theKey*. The element node X that contains *theElement* is discarded, and we retrace the path from X to the root discarding branch nodes that are roots of subtries that have only 1 element in them. This path retracing stops when we either reach a branch node that is not discarded or we discard the root.

Consider the trie of Figure 28.7. When the element with key 951-23-7625 is removed, the element node J is discarded and we follow the path from node J to the root node A. The branch node I is discarded because the subtrie with root I contains the single element node K. We next reach the branch node F. This node is also discarded, and we proceed to the branch node D. Since the subtrie rooted at D has 2 element nodes (K and L), this branch node is not discarded. Instead, node K is made a child of this branch node, as is shown in Figure 28.8.

To remove the element with key 562-44-2169 from the trie of Figure 28.8, we discard the element node C. Since its parent node remains the root of a subtrie that has more than one element, the parent node is not discarded and the removal operation is complete. Figure 28.9 show the resulting trie.

The time required to remove an element with a d digit key from a radix r trie is $O(dr)$

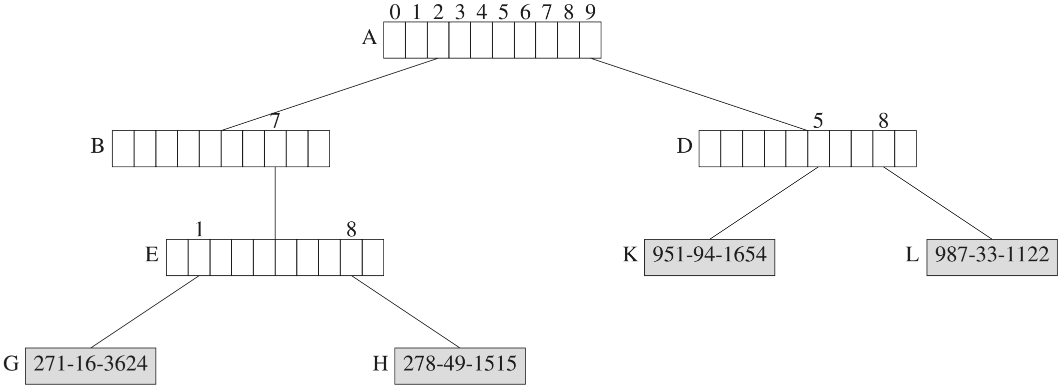


FIGURE 28.9: Trie of Figure 28.8 with 562-44-2169 removed.

ps2ascii	ps2pdf	psbook	psmandup	psselect
ps2epsi	ps2pk	pscal	psmerge	pstopnm
ps2frag	ps2ps	psidtopgm	psnup	pstops
ps2gif	psbb	pslatex	psresize	pstruct

TABLE 28.2 Commands that begin with "ps"

because the removal may require us to discard $O(d)$ branch nodes and it takes $O(r)$ time to determine whether a branch node is to be discarded. The complexity of the remove operation can be reduced to $O(r + d)$ by adding a $numberOfElementsInSubtrie$ field to each branch node.

28.8 Prefix Search and Applications

You have probably realized that to search a trie we do not need the entire key. Most of the time, only the first few digits (i.e., a prefix) of the key is needed. For example, our search of the trie of Figure 28.1 for an element with key 951-23-7625 used only the first four digits of the key. The ability to search a trie using only the prefix of a key enables us to use tries in applications where only the prefix might be known or where we might desire the user to provide only a prefix. Some of these applications are described below.

Criminology

Suppose that you are at the scene of a crime and observe the first few characters CRX on the registration plate of the getaway car. If we have a trie of registration numbers, we can use the characters CRX to reach a subtrie that contains all registration numbers that begin with CRX. The elements in this subtrie can then be examined to see which cars satisfy other properties that might have been observed.

Automatic Command Completion

When using an operating system such as Unix or DOS, we type in system commands to accomplish certain tasks. For example, the Unix and DOS command *cd* may be used to change the current directory. Table 28.2 gives a list of commands that have the prefix *ps* (this list was obtained by executing the command $ls/usr/local/bin/ps*$ on a Unix system).

We can simply the task of typing in commands by providing a command completion facility which automatically types in the command suffix once the user has typed in a long enough prefix to uniquely identify the command. For instance, once the letters *psi* have been entered, we know that the command must be *psidtopgm* because there is only one command that has the prefix *psi*. In this case, we replace the need to type in a 9 character command name by the need to type in just the first 3 characters of the command!

A command completion system is easily implemented when the commands are stored in a trie using ASCII characters as the digits. As the user types the command digits from left to right, we move down the trie. The command may be completed as soon as we reach an element node. If we fall off the trie in the process, the user can be informed that no command with the typed prefix exists.

Although we have described command completion in the context of operating system commands, the facilty is useful in other environments:

1. A network browser keeps a history of the URLs of sites that you have visited. By organizing this history as a trie, the user need only type the prefix of a previously used URL and the browser can complete the URL.
2. A word processor can maintain a dictionary of words and can complete words as you type the text. Words can be completed as soon as you have typed a long enough prefix to identify the word uniquely.
3. An automatic phone dialer can maintain a list of frequently called telephone numbers as a trie. Once you have punched in a long enough prefix to uniquely identify the phone number, the dialer can complete the call for you.

28.9 Compressed Tries

Take a close look at the trie of Figure 28.1. This trie has a few branch nodes (nodes B, D, and F) that do not partition the elements in their subtrie into two or more nonempty groups. We often can improve both the time and space performance metrics of a trie by eliminating all branch nodes that have only one child. The resulting trie is called a *compressed trie*.

When branch nodes with a single child are removed from a trie, we need to keep additional information so that dictionary operations may be performed correctly. The additional information stored in three compressed trie structures is described below.

28.9.1 Compressed Tries with Digit Numbers

In a *compressed trie with digit numbers*, each branch node has an additional field $digitNumber$ that tells us which digit of the key is used to branch at this node. Figure 11 shows the compressed trie with digit numbers that corresponds to the trie of Figure 28.1. The leftmost field of each branch node of Figure 28.10 is the $digitNumber$ field.

Searching a Compressed Trie with Digit Numbers

A compressed trie with digit numbers may be searched by following a path from the root. At each branch node, the digit, of the search key, given in the branch node's $digitNumber$ field is used to determine which subtrie to move to. For example, when searching the trie of Figure 28.10 for an element with key 951-23-7625, we start at the root of the trie. Since the root node is a branch node with $digitNumber = 1$, we use the first digit 9 of the search key to determine which subtrie to move to. A move to node $A.child[9] = I$ is made. Since $I.digitNumber = 4$, the fourth digit, 2, of the search key tells us which subtrie to move to. A move is now made to node $I.child[2] = J$. We are now

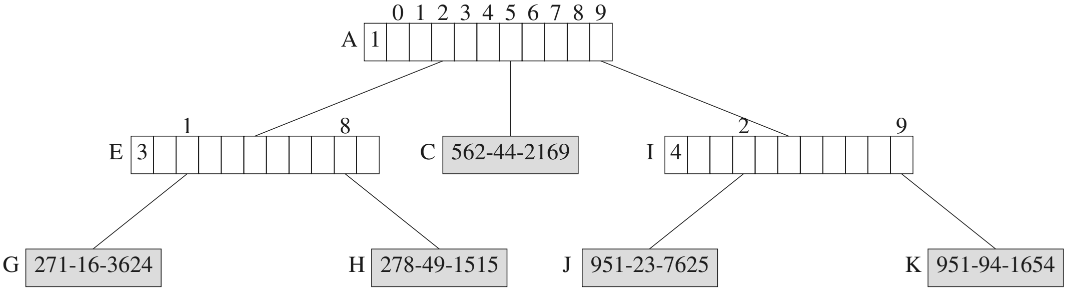


FIGURE 28.10: Compressed trie with digit numbers.

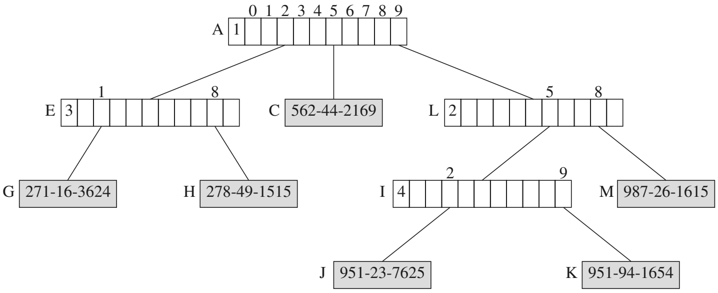


FIGURE 28.11: Compressed trie following the insertion of 987-26-1615 into the compressed trie of Figure 28.10.

at an element node, and the search key is compared with the key of the element in node J. Since the keys match, we have found the desired element.

Notice that a search for an element with key 913-23-7625 also terminates at node J. However, the search key and the element key at node J do not match and we conclude that the trie contains no element with key 913-23-7625.

Inserting into a Compressed Trie with Digit Numbers

To insert an element with key 987-26-1615 into the trie of Figure 28.10, we first search for an element with this key. The search ends at node J. Since the search key and the key, 951-23-7625, of the element in this node do not match, we conclude that the trie has no element whose key matches the search key. To insert the new element, we find the first digit where the search key differs from the key in node J and create a branch node for this digit. Since the first digit where the search key 987-26-1615 and the element key 951-23-7625 differ is the second digit, we create a branch node with $digitNumber = 2$. Since digit values increase as we go down the trie, the proper place to insert the new branch node can be determined by retracing the path from the root to node J and stopping as soon as either a node with digit value greater than 2 or the node J is reached. In the trie of Figure 28.10, this path retracing stops at node I. The new branch node is made the parent of node I, and we get the trie of Figure 28.11.

Consider inserting an element with key 958-36-4194 into the compressed trie of Figure 28.10. The search for an element with this key terminates when we fall of the trie by following the pointer $I.child[3] = null$. To complete the insertion, we must first find an element in the subtrie rooted at node I. This element is found by following a downward path from node I using (say) the first non *null* link in each branch node encountered. Doing

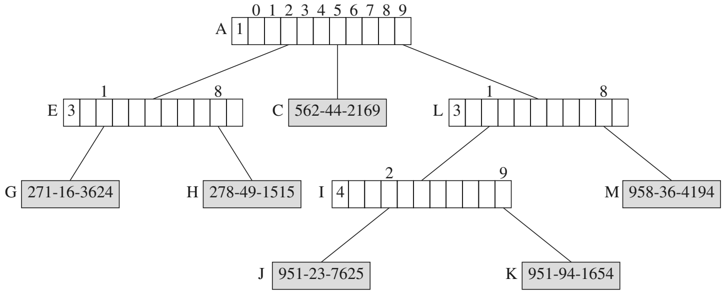


FIGURE 28.12: Compressed trie following the insertion of 958-36-4194 into the compressed trie of Figure 28.10.

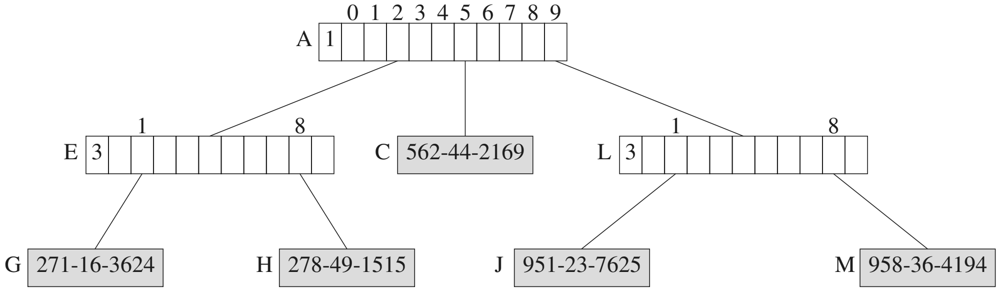


FIGURE 28.13: Compressed trie following the removal of 951-94-1654 from the compressed trie of Figure 28.12.

this on the compressed trie of Figure 28.10, leads us to node J. Having reached an element node, we find the first digit where the element key and the search key differ and complete the insertion as in the previous example. Figure 28.12 shows the resulting compressed trie.

Because of the possible need to search for the first non *null* child pointer in each branch node, the time required to insert an element into a compressed tries with digit numbers is $O(rd)$, where r is the trie radix and d is the maximum number of digits in any key.

Removing an Element from a Compressed Trie with Digit Numbers

To remove an element whose key is $theKey$, we do the following:

1. Find the element node X that contains the element whose key is $theKey$.
2. Discard node X.
3. If the parent of X is left with only one child, discard the parent node also. When the parent of X is discarded, the sole remaining child of the parent of X becomes a child of the grandparent (if any) of X.

To remove the element with key 951-94-1654 from the compressed trie of Figure 28.12, we first locate the node K that contains the element that is to be removed. When this node is discarded, the parent I of K is left with only one child. Consequently, node I is also discarded, and the only remaining child J of node I is the made a child of the grandparent of K. Figure 28.13 shows the resulting compressed trie.

Because of the need to determine whether a branch node is left with two or more children, removing a d digit element from a radix r trie takes $O(d + r)$ time.

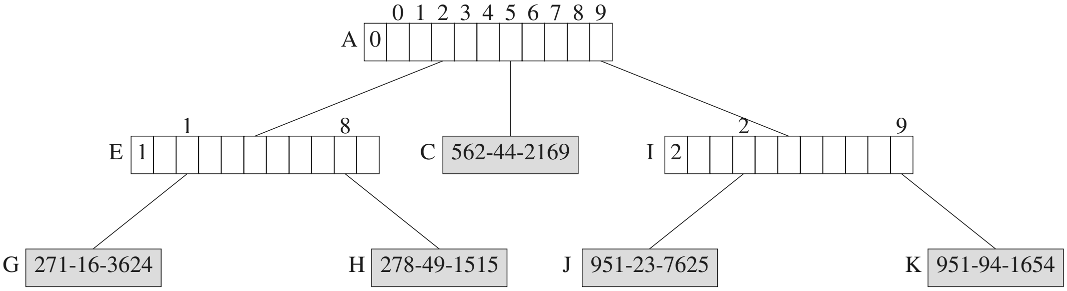


FIGURE 28.14: Compressed trie with skip fields.

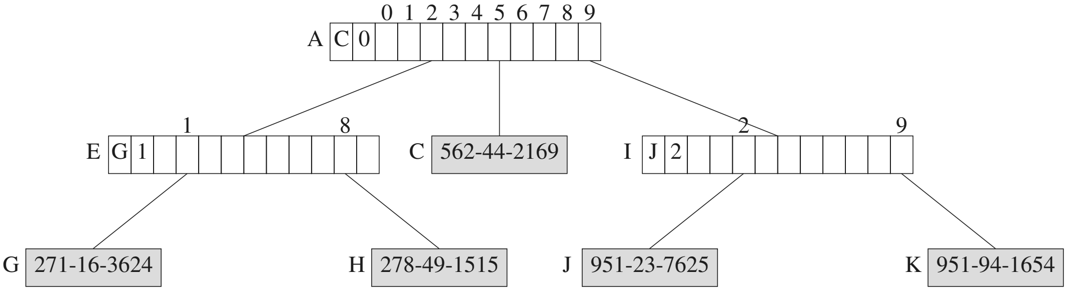


FIGURE 28.15: Compressed trie with edge information.

28.9.2 Compressed Tries with Skip Fields

In a *compressed trie with skip fields*, each branch node has an additional field *skip* which tells us the number of branch nodes that were originally between the current branch node and its parent. Figure 15 shows the compressed trie with skip fields that corresponds to the trie of Figure 28.1. The leftmost field of each branch node of Figure 28.14 is the skip field.

The algorithms to search, insert, and remove are very similar to those used for a compressed trie with digit numbers.

28.9.3 Compressed Tries with Edge Information

In a *compressed trie with edge information*, each branch node has the following additional information associated with it: a pointer/reference *element* to an element (or element node) in the subtrie, and an integer *skip* which equals the number of branch nodes eliminated between this branch node and its parent. Figure 28.15 shows the compressed trie with edge information that corresponds to the trie of Figure 28.1. The first field of each branch node is its *element* field, and the second field is the *skip* field.

Even though we store the "edge information" with branch nodes, it is convenient to think of this information as being associated with the edge that comes into the branch node from its parent (when the branch node is not the root). When moving down a trie, we follow edges, and when an edge is followed. we skip over the number of digits given by the *skip* field of the edge information. The value of the digits that are skipped over may be determined by using the *element* field.

When moving from node A to node I of the compressed trie of Figure 28.15, we use digit 1 of the key to determine which child field of A is to be used. Also, we skip over the next 2 digits, that is, digits 2 and 3, of the keys of the elements in the subtrie rooted at

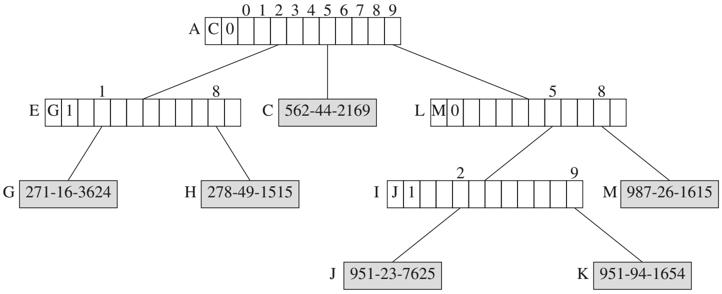


FIGURE 28.16: Compressed trie following the insertion of 987-26-1615 into the compressed trie of Figure 28.15.

I. Since all elements in the subtrie I have the same value for the digits that are skipped over, we can determine the value of these skipped over digits from any of the elements in the subtrie. Using the *element* field of the edge information, we access the element node J, and determine that the digits that are skipped over are 5 and 1.

Searching a Compressed Trie with Edge Information

When searching a compressed trie with edge information, we can use the edge information to terminate unsuccessful searches (possibly) before we reach an element node or fall off the trie. As in the other compressed trie variants, the search is done by following a path from the root. Suppose we are searching the compressed trie of Figure 28.15 for an element with key 921-23-1234. Since the *skip* value for the root node is 0, we use the first digit 9 of the search key to determine which subtrie to move to. A move to node $A.child[9] = I$ is made. By examining the edge information (stored in node I), we determine that, in making the move from node A to node I, the digits 5 and 1 are skipped. Since these digits do not agree with the next two digits of the search key, the search terminates with the conclusion that the trie contains no element whose key equals the search key.

Inserting into a Compressed Trie with Edge Information

To insert an element with key 987-26-1615 into the compressed trie of Figure 28.15, we first search for an element with this key. The search terminates unsuccessfully when we move from node A to node I because of a mismatch between the skipped over digits and the corresponding digits of the search key. The first mismatch is at the first skipped over digit. Therefore, we insert a branch node L between nodes A and I. The *skip* value for this branch node is 0, and its *element* field is set to reference the element node for the newly inserted element. We must also change the *skip* value of I to 1. Figure 28.16 shows the resulting compressed trie.

Suppose we are to insert an element with key 958-36-4194 into the compressed trie of Figure 16. The search for an element with this key terminates when we move to node I because of a mismatch between the digits that are skipped over and the corresponding digits of the search key. A new branch node is inserted between nodes A and I and we get the compressed trie that is shown in Figure 28.17.

The time required to insert a d digit element into a radix r compressed trie with edge information is $O(r + d)$.

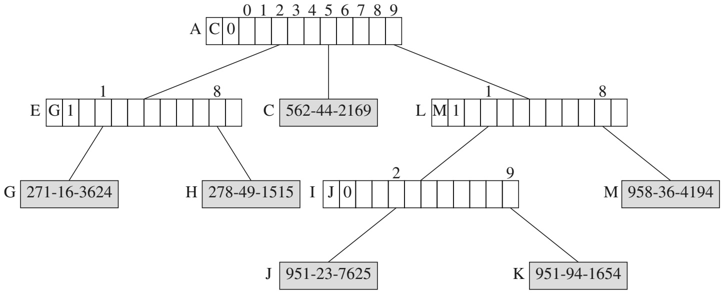


FIGURE 28.17: Compressed trie following the insertion of 958-36-4194 into the compressed trie of Figure 28.15.

Removing an Element from a Compressed Trie with Edge Information

This is similar to removal from a compressed trie with digit numbers except for the need to update the *element* fields of branch nodes whose *element* field references the removed element.

28.9.4 Space Required by a Compressed Trie

Since each branch node partitions the elements in its subtrie into two or more nonempty groups, an n element compressed trie has at most $n - 1$ branch nodes. Therefore, the space required by each of the compressed trie variants described by us is $O(nr)$, where r is the trie radix.

When compressed tries are represented as hash tables, we need an additional data structure to store the nonpointer fields of branch nodes. We may use an array (much like we use the array $information$) for this purpose.

28.10 Patricia

The data structure Patricia (**P**ractical **A**lgorithm **T**o **R**etrieve **I**nformation **C**oded **I**n **A**lphanumeric) is a compressed binary trie in which the branch and element nodes have been melded into a single node type. Consider the compressed binary trie of Figure 28.18. Circular nodes are branch nodes and rectangular nodes are element nodes. The number inside a branch node is its bit number field; the left child of a branch node corresponds to the case when the appropriate key bit is 0 and the right child to the case when this bit is 1. The melding of branch and element nodes is done by moving each element from its element node to an ancestor branch node. Since the number of branch nodes is one less than the number of element nodes, we introduce a header node and make the compressed binary trie the left subtree of the header. Pointers that originally went from a branch node to an element node now go from that branch node to the branch node into which the corresponding element has been melded. Figure 28.19 shows a possible result of melding the nodes of Figure 28.18. The number outside a node is its bit number values. The thick pointers are *backward* pointers that replace branch-node to element-node pointers in Figure 28.18. A backward pointer has the property that the bit number value at the start of the pointer is $\geq$ the bit number value at its end. For original branch-node to branch-node pointers (also called *downward* pointers), the bit number value at the pointer end is always greater than the bit number value at the pointer start.

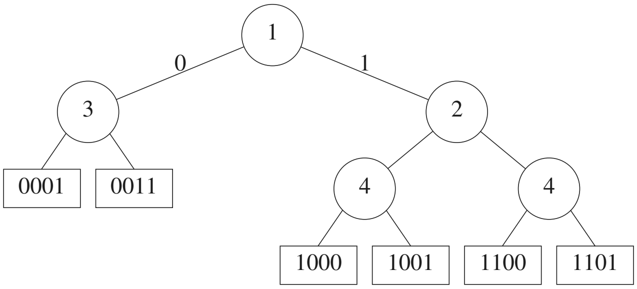


FIGURE 28.18: A compressed binary trie.

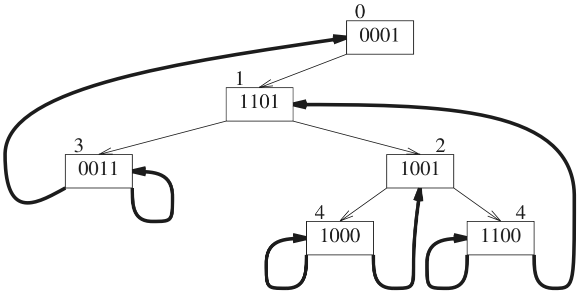


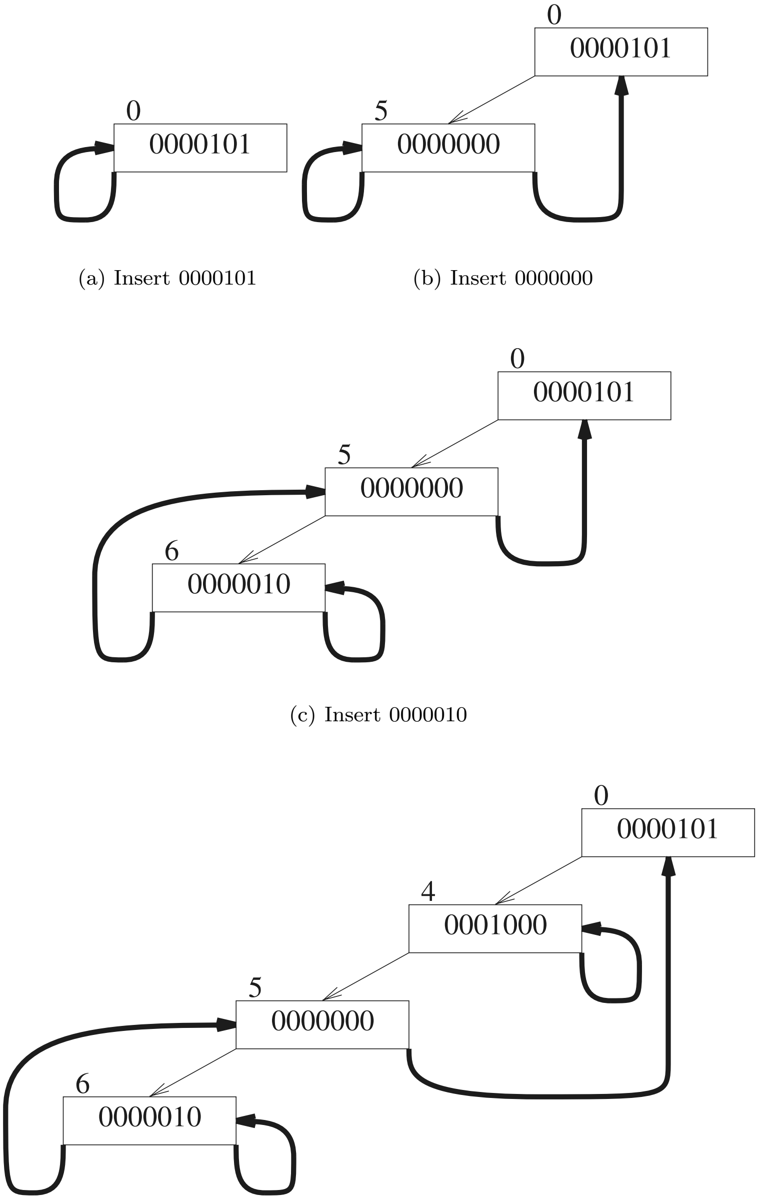
FIGURE 28.19: A Patricia instance that corresponds to Figure 28.18.

28.10.1 Searching

To search for an element with key *theKey*, we use the bits of *theKey* from left to right to move down the Patricia instance just as we would search a compressed binary trie. When a backward pointer is followed, we compare *theKey* with the key in the reached Patricia node. For example, to search for *theKey* = 1101, we start by moving into the left subtree of the header node. The pointer that is followed is a downward pointer (start bit number is 0 and end bit number is 1). We branch using bit 1 of *theKey*. Since this bit is 1, we follow the right child pointer. The start bit number for this pointer is 1 and the end bit number is 2. So, again, a downward pointer has been followed. The bit number for the reached node is 2. So, we use bit 2 of *theKey* to move further. Bit 2 is a 1. So, we use the right child pointer to get to the node that contains 1100. Again, a downward pointer was used. From this node, a move is made using bit 4 of *theKey*. This gets us to the node that contains 1101. This time, a backward pointer was followed (start bit of pointer is 4 and end bit is 1). When a backward pointer is followed, we compare *theKey* with the key in the reached node. In this case, the keys match and we have found the desired element. Notice that the same search path is taken when *theKey* = 1111. In this case, the final compare fails and we conclude that we have no element whose key is 1111.

28.10.2 Inserting an Element

We use an example to illustrate the insert algorithm. We start with an empty instance. Such an instance has no node; not even the header. For our example, we will consider keys that have 7 bits. For the first insert, we use the key 0000101. When inserting into an empty instance, we create a header node whose left child pointer points to the header node; the new element is inserted into the header; and the bit number field of the header set to 0.



(a) Insert 0000101

(b) Insert 0000000

(c) Insert 0000010

(d) Insert 0001000

FIGURE 28.20: Insertion example.

The configuration of Figure 28.20(a) results. Note that the right child field of the header node is not used.

The key for the next insert is 0000000. The search for this key terminates at the header. We compare the insert key and the header key and determine that the first bit at which they differ is bit 5. So, we create a new node with bit number field 5 and insert the new element into this node. Since bit 5 of the insert key is 0, the left child pointer of the new node points to the new node and the right child pointer to the header node (Figure 28.20(b)).

For the next insertion, assume that the key is 0000010. The search for this key terminates at the node with 0000000. Comparing the two keys, we determine that the first bit at which the two keys differ is bit 6. So, we create a new node with bit number field 6 and put the new element into this new node. The new node is inserted into the search path in such a way that bit number fields increase along this path. For our example, the new node is inserted as the left child of the node with 0000000. Since bit 6 of the insert key is 1, the right child pointer of the new node is a self pointer and the left child pointer points to the node with 0000000. Figure 28.20(c) shows the result.

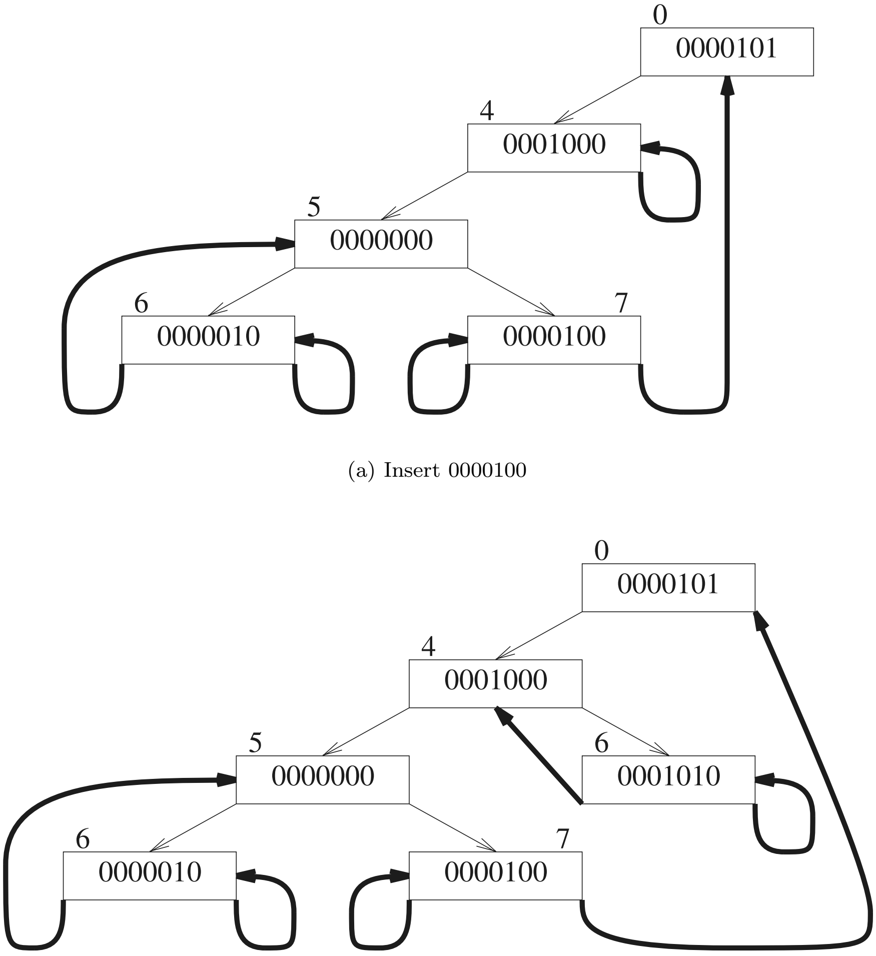
The general strategy to insert an element other than the first one is given below. The key of the element to be inserted is $theKey$.

1. Search for $theKey$. Let $reachedKey$ be the key in the node $endNode$ where the search terminates.
2. Determine the leftmost bit position $lBitPos$ at which $theKey$ and $reachedKey$ differ. $lBitPos$ is well defined so long as one of the two keys isn't a prefix of the other.
3. Create a new node with bit number field $lBitPos$. Insert this node into the search path from the header to $endNode$ so that bit numbers increase along this path. This insertion breaks a pointer from node p to node q. The pointer now goes from p to the new node.
4. If bit $lBitPos$ of $theKey$ is 1, the right child pointer of the new node becomes a self pointer (i.e., it points to the new node); otherwise, the left child pointer of the new node becomes a self pointer. The remaining child pointer of the new node points to q.

For our next insert, the insert key is 0001000. The search for this key terminates at the node with 0000000. We determine that the first bit at which the insert key and $reachedKey$ = 0000000 differ is bit 4. We create a new node with bit number 4 and put the new element into this node. The new node is inserted on the search path so as to ensure that bit number fields increase along this path. So, the new node is inserted as the left child of the header node (Figure 28.18(d)). This breaks the pointer from the header to the node q with 0000000. Since bit 4 of the insert key is 1, the right child pointer of the new node is a self pointer and the left child pointer goes to node q.

We consider two more inserts. Consider inserting an element whose key is 0000100. The reached key is 0000101 (in the header). We see that the first bit at which the insert and reached keys differ is bit 7. So, we create a new node with bit number 7; the new element is put into the new node; the new node is inserted into the search path so as to ensure that bit numbers increase along this path (this requires the new node to be made a right child of the node with 0000000, breaking the child pointer from 0000000 to the header); for the broken pointer, p is the node with 0000000 and q is the header; the left child pointer of the new node is a self pointer (because bit 7 of the insert key is 0); and remaining child pointer (in this case the right child) of the new node points to q (see Figure 28.21(a)).

For our final insert, the insert key is 0001010. A search for this key terminates at the node with 0000100. The first bit at which the insert and reached keys differ is bit 6. So, we create a new node with bit number 6; the new element is put into the new node; the new node is inserted into the search path so as to ensure that bit numbers increase along this path (this requires the new node to be made a right child of the node with 0001000; so, p = q = node with 001000); the right child pointer of the new node is a self pointer (because



(a) Insert 0000100

(b) Insert 0001010

FIGURE 28.21: Insertion example.

bit 6 of the insert key is 1); and the remaining child (in this case the left child) of the new node is q (see Figure 28.21(b)).

28.10.3 Removing an Element

Let p be the node that contains the element that is to be removed. We consider two cases for p—(a) p has a self pointer and (b) p has no self pointer. When p has a self pointer and p is the header, the Patricia instance becomes empty following the element removal (Figure 28.20(a)). In this case, we simply dispose of the header. When p has a self pointer and p is not the header, we set the pointer from the parent of p to the value of p's non-self pointer. Following this pointer change, the node p is disposed. For example, to remove the element with key 0000010 from Figure 28.21(a), we change the left child pointer in the node with 0000000 to point to the node pointed at by p's non-self pointer (i.e., the node with 0000000). This causes the left child pointer of 0000000 to become a self pointer. The node with 0000010 is then disposed.

For the second case, we first find the node q that has a backward pointer to node p. For example, when we are to remove 0001000 from Figure 28.21(b), the node q is the node with 0001010. The element *qElement* in q (in our example, 0001010) is moved to node p and we proceed to delete node q instead of node p. Notice that node q is the node from which we reached node p in the search for the element that is to be removed. To delete node q, we first find the node r that has a back pointer to q (for our example $r = q$). The node r is found by using the key of *qElement*. Once r is found, the back pointer to q that is in r is changed from q to p to properly account for the fact that we have moved *qElement* to p. Finally, the downward pointer from the parent of q to q is changed to q's child pointer that was used to locate r. In our example p is the parent of q and the right child pointer of p is changed from q to the right child of q, which itself was just changed to p.

We see that the time for each of the Patricia operations search, insert, and delete is $O(h)$, where h is the height of the Patricia instance. For more information on Patricia and general tries, see [2, 3].

Acknowledgment

This work was supported, in part, by the National Science Foundation under grant CCR-9912395.

References

[1] A. Andersson and S. Nillson, Improved behavior of tries by adaptive branching, *Information Processing Letters*, 46, 1993, 295-300.

[2] E. Horowitz, S. Sahni, and D. Mehta, *Fundamentals of Data Structures in C++*, Computer Science Press, 1995.

[3] D. Knuth, *The Art of Computer Programming: Sorting and Searching*, Second Edition, Addison-Wesley, 1998.

29

Suffix Trees and Suffix Arrays

Srinivas Aluru
Iowa State University

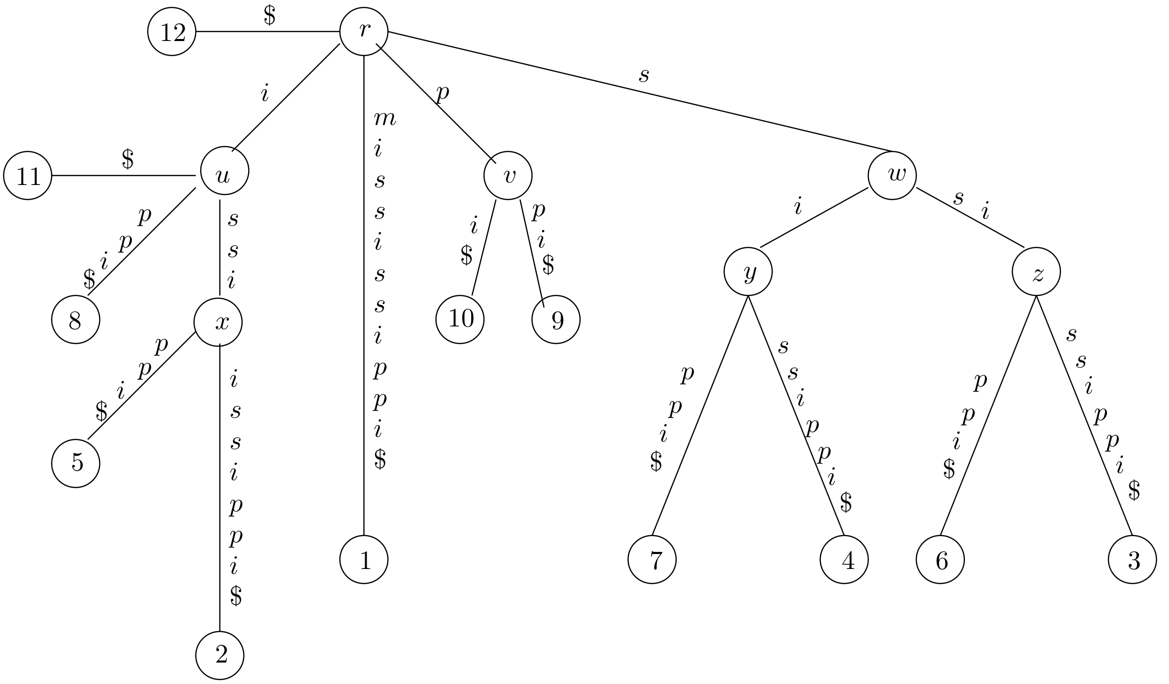
29.1 Basic Definitions and Properties

Suffix trees and suffix arrays are versatile data structures fundamental to string processing applications. Let s' denote a string over the alphabet Σ. Let $\$ \notin \Sigma$ be a unique termination character, and $s = s'\$$ be the string resulting from appending \$ to s'. We use the following notation: $|s|$ denotes the size of s, $s[i]$ denotes the i^{th} character of s, and $s[i..j]$ denotes the substring $s[i]s[i+1]\ldots s[j]$. Let $suff_i = s[i]s[i+1]\ldots s[|s|]$ be the suffix of s starting at i^{th} position.

The suffix tree of s, denoted $ST(s)$ or simply ST, is a compacted trie (See Chapter 28) of all suffixes of string s. Let $|s| = n$. It has the following properties:

1. The tree has n leaves, labeled $1 \ldots n$, one corresponding to each suffix of s.
2. Each internal node has at least 2 children.
3. Each edge in the tree is labeled with a substring of s.
4. The concatenation of edge labels from the root to the leaf labeled i is $suff_i$.
5. The labels of the edges connecting a node with its children start with different characters.

The paths from root to the suffixes labeled i and j coincide up to their longest common prefix, at which point they bifurcate. If a suffix of the string is a prefix of another longer suffix, the shorter suffix must end in an internal node instead of a leaf, as desired. It is to avoid this possibility that the unique termination character is added to the end of the string. Keeping this in mind, we use the notation $ST(s')$ to denote the suffix tree of the string obtained by appending \$ to s'.



SA	12	11	8	5	2	1	10	9	7	4	6	3
Lcp	0	1	1	4	0	0	1	0	2	1	3	

FIGURE 29.1: Suffix tree, suffix array and *Lcp* array of the string *mississippi*. The suffix links in the tree are given by $x \rightarrow z \rightarrow y \rightarrow u \rightarrow r$, $v \rightarrow r$, and $w \rightarrow r$.

As each internal node has at least 2 children, an n-leaf suffix tree has at most $n - 1$ internal nodes. Because of property (5), the maximum number of children per node is bounded by $|\Sigma| + 1$. Except for the edge labels, the size of the tree is $O(n)$. In order to allow a linear space representation of the tree, each edge label is represented by a pair of integers denoting the starting and ending positions, respectively, of the substring describing the edge label. If the edge label corresponds to a repeat substring, the indices corresponding to any occurrence of the substring may be used. The suffix tree of the string *mississippi* is shown in Figure 29.1. For convenience of understanding, we show the actual edge labels.

The suffix array of $s = s'\$$, denoted $SA(s)$ or simply SA, is a lexicographically sorted array of all suffixes of s. Each suffix is represented by its starting position in s. $SA[i] = j$ iff $Suff_j$ is the i^{th} lexicographically smallest suffix of s. The suffix array is often used in conjunction with an array termed *Lcp* array, containing the lengths of the longest common prefixes between every consecutive pair of suffixes in SA. We use $lcp(\alpha, \beta)$ to denote the longest common prefix between strings α and β. We also use the term *lcp* as an abbreviation for the term *longest common prefix*. $Lcp[i]$ contains the length of the *lcp* between $suff_{SA[i]}$ and $suff_{SA[i+1]}$, i.e., $Lcp[i] = lcp(suff_{SA[i]}, suff_{SA[i+1]})$. As with suffix trees, we use the notation $SA(s')$ to denote the suffix array of the string obtained by appending \$ to s'. The suffix and *Lcp* arrays of the string *mississippi* are shown in Figure 29.1.

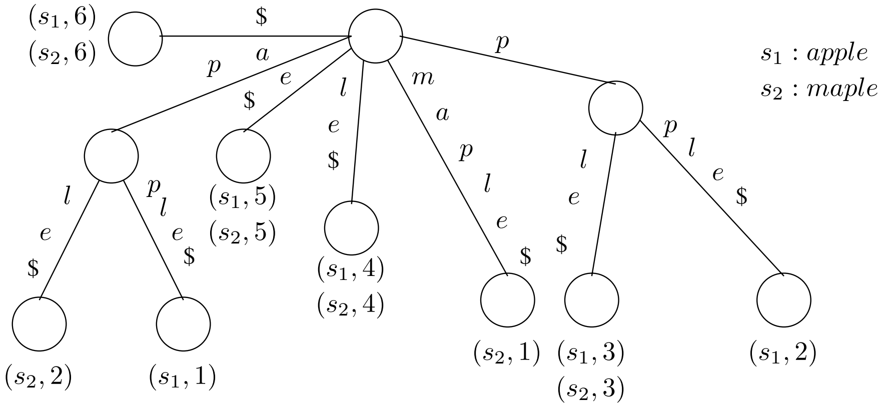
Let v be a node in the suffix tree. Let *path-label*(v) denote the concatenation of edge labels along the path from root to node v. Let *string-depth*(v) denote the length of *path-label*(v). To differentiate this with the usual notion of depth, we use the term *tree-depth* of a node to denote the number of edges on the path from root to the node. Note that the length of the longest common prefix between two suffixes is the string depth of the lowest common ancestor of the leaf nodes corresponding to the suffixes. A repeat substring of string S is *right-maximal* if there are two occurrences of the substring that are succeeded by different characters in the string. The path label of each internal node in the suffix tree corresponds to a right-maximal repeat substring and vice versa.

Let v be an internal node in the suffix tree with path-label $c\alpha$ where c is a character and α is a (possibly empty) string. Therefore, $c\alpha$ is a right-maximal repeat, which also implies that α is also a right maximal repeat. Let u be the internal node with path label α. A pointer from node v to node u is called a *suffix link*; we denote this by $SL(v) = u$. Each suffix $suff_i$ in the subtree of v shares the common prefix $c\alpha$. The corresponding suffix $suff_{i+1}$ with prefix α will be present in the subtree of u. The concatenation of edge labels along the path from v to leaf labeled i, and along the path from u to leaf labeled $i+1$ will be the same. Similarly, each internal node in the subtree of v will have a corresponding internal node in the subtree of u. In this sense, the entire subtree under v is contained in the subtree under u.

Every internal node in the suffix tree other than the root has a suffix link from it. Let v be an internal node with $SL(v) = u$. Let v' be an ancestor of v other than the root and let $u' = SL(v')$. As *path-label*(v') is a prefix of *path-label*(v), *path-label*(u') is also a prefix of *path-label*(u). Thus, u' is an ancestor of u. Each proper ancestor of v except the root will have a suffix link to a distinct proper ancestor of u. It follows that *tree-depth*$(u) \geq$ *tree-depth*$(v) - 1$.

Suffix trees and suffix arrays can be generalized to multiple strings. The generalized suffix tree of a set of strings $\mathcal{S} = \{s_1, s_2, \ldots, s_k\}$, denoted $GST(\mathcal{S})$ or simply GST, is a compacted trie of all suffixes of each string in $\mathcal{S}$. We assume that the unique termination character \$ is appended to the end of each string. A leaf label now consists of a pair of integers (i, j), where i denotes the suffix is from string s_i and j denotes the starting position of the suffix in s_i. Similarly, an edge label in a GST is a substring of one of the strings. An edge label is represented by a triplet of integers (i, j, l), where i denotes the string number, and j and l denote the starting and ending positions of the substring in s_i. For convenience of understanding, we will continue to show the actual edge labels. Note that two strings may have identical suffixes. This is compensated by allowing leaves in the tree to have multiple labels. If a leaf is multiply labeled, each suffix should come from a different string. If N is the total number of characters (including the \$ in each string) of all strings in $\mathcal{S}$, the GST has at most N leaf nodes and takes up $O(N)$ space. The generalized suffix array of $\mathcal{S}$, denoted $GSA(\mathcal{S})$ or simply GSA, is a lexicographically sorted array of all suffixes of each string in $\mathcal{S}$. Each suffix is represented by an integer pair (i, j) denoting suffix starting from position j in s_i. If suffixes from different strings are identical, they occupy consecutive positions in the GSA. For convenience, we make an exception for the suffix \$ by listing it only once, though it occurs in each string. The GST and GSA of strings *apple* and *maple* are shown in Figure 29.2.

Suffix trees and suffix arrays can be constructed in time linear to the size of the input. Suffix trees are very useful in solving a plethora of string problems in optimal run-time bounds. Moreover, in many cases, the algorithms are very simple to design and understand. For example, consider the classic pattern matching problem of determining if a pattern P occurs in text T over a constant sized alphabet. Note that P occurs starting from position i in T iff P is a prefix of $suff_i$ in T. Thus, whether P occurs in T or not can be determined by checking if P matches an initial part of a path from root to a leaf in $ST(T)$. Traversing



GSA	$(s_1, 6)$	$(s_2, 2)$	$(s_1, 1)$	$(s_1, 5)$	$(s_2, 5)$	$(s_1, 4)$	$(s_2, 4)$	$(s_2, 1)$	$(s_1, 3)$	$(s_2, 3)$	$(s_1, 2)$

FIGURE 29.2: Generalized suffix tree and generalized suffix array of strings *apple* and *maple*.

from the root matching characters in P, this can be determined in $O(|P|)$ time, independent of the size of T. As another application, consider the problem of finding a longest common substring of a pair of strings. Once the *GST* of the two strings is constructed, all that is needed is to identify an internal node with the largest string depth that contains at least one leaf from each string. These and many other applications are explored in great detail in subsequent sections. Suffix arrays are of interest because they require much less space than suffix trees, and can be used to solve many of the same problems. We first concentrate on linear time construction algorithms for suffix trees and suffix arrays. The reader interested in applications can safely skip to Section 29.3.

29.2 Linear Time Construction Algorithms

In this section, we explore linear time construction algorithms for suffix trees and suffix arrays. We also show how suffix trees and suffix arrays can be derived from each other in linear time. In suffix tree and suffix array construction algorithms, three different types of alphabets are considered: a constant or fixed size alphabet ($|\Sigma| = O(1)$), integer alphabet ($\Sigma = \{1, 2, \ldots, n\}$), and arbitrary alphabet. Suffix trees and suffix arrays can be constructed in linear time for both constant size and integer alphabets. The constant alphabet size case covers many interesting application areas, such as English text, or DNA or protein sequences in molecular biology. The integer alphabet case is interesting because a string of length n can have at most n distinct characters. Furthermore, some algorithms use a recursive technique that would generate and require operating on strings over integer alphabet, even when applied to strings over a fixed alphabet.

29.2.1 Suffix Trees vs. Suffix Arrays

We first show that the suffix array and *Lcp* array of a string can be obtained from its suffix tree in linear time. Define lexicographic ordering of the children of a node to be the order based on the first character of the edge labels connecting the node to its children. Define lexicographic depth first search to be a depth first search of the tree where the children of each node are visited in lexicographic order. The order in which the leaves of a suffix tree are visited in a lexicographic depth first search gives the suffix array of the corresponding string. In order to obtain *lcp* information, the string-depth of the current node during the search is remembered. This can be easily updated in $O(1)$ time per edge as the search progresses. The length of the *lcp* between two consecutive suffixes is given by the smallest string-depth of a node visited between the two suffixes.

Given the suffix array and the *Lcp* array of a string s ($|s\$| = n$), its suffix tree can be constructed in $O(n)$ time. This is done by starting with a partial suffix tree for the lexicographically smallest suffix, and repeatedly inserting subsequent suffixes in the suffix array into the tree until the suffix tree is complete. Let T_i denote the compacted trie of the first i suffixes in lexicographic order. The first tree T_1 consists of a single leaf labeled $SA[1]$ connected to the root with an edge labeled $suff_{SA[1]} = \$$.

To insert $SA[i+1]$ into T_i, start with the most recently inserted leaf $SA[i]$ and walk up $(|suff_{SA[i]}| - |lcp(suff_{SA[i]}, suff_{SA[i+1]})|) = ((n - SA[i] + 1) - Lcp[i])$ characters along the path to the root. This walk can be done in $O(1)$ time per edge by calculating the lengths of the respective edge labels. If the walk does not end at an internal node, create an internal node. Create a new leaf labeled $SA[i+1]$ and connect it to this internal node with an edge. Set the label on this edge to $s[SA[i+1] + Lcp[i]..n]$. This creates the tree T_{i+1}. The procedure works because $suff_{SA[i+1]}$ shares a longer prefix with $suff_{SA[i]}$ than any other suffix inserted so far. To see that the entire algorithm runs in $O(n)$ time, note that inserting a new suffix into T_i requires walking up the rightmost path in T_i. Each edge that is traversed ceases to be on the rightmost path in T_{i+1}, and thus is never traversed again. An edge in an intermediate tree T_i corresponds to a path in the suffix tree ST. When a new internal node is created along an edge in an intermediate tree, the edge is split into two edges, and the edge below the newly created internal node corresponds to an edge in the suffix tree. Once again, this edge ceases to be on the rightmost path and is never traversed again. The cost of creating an edge in an intermediate tree can be charged to the lowest edge on the corresponding path in the suffix tree. As each edge is charged once for creating and once for traversing, the total run-time of this procedure is $O(n)$.

Finally, the *Lcp* array itself can be constructed from the suffix array and the string in linear time [14]. Let R be an array of size n such that $R[i]$ contains the position in SA of $suff_i$. R can be constructed by a linear scan of SA in $O(n)$ time. The *Lcp* array is computed in n iterations. In iteration i of the algorithm, the longest common prefix between $suff_i$ and its respective right neighbor in the suffix array is computed. The array R facilitates locating an arbitrary suffix $suff_i$ and finding its right neighbor in the suffix array in constant time. Initially, the length of the longest common prefix between $suff_1$ and its suffix array neighbor is computed directly and recorded. Let $suff_j$ be the right neighbor of $suff_i$ in SA. Let l be the length of the longest common prefix between them. Suppose $l \geq 1$. As $suff_j$ is lexicographically greater than $suff_i$ and $s[i] = s[j]$, $suff_{j+1}$ is lexicographically greater than $suff_{i+1}$. The length of the longest common prefix between them is $l - 1$. It follows that the length of the longest common prefix between $suff_{i+1}$ and its right neighbor in the suffix array is $\geq l - 1$. To determine its correct length, the comparisons need only start from the l^{th} characters of the suffixes.

To prove that the run time of the above algorithm is linear, charge a comparison between the r^{th} character in suffix $suff_i$ and the corresponding character in its right neighbor suffix in SA to the position in the string of the r^{th} character of $suff_i$, i.e., $i + r - 1$. A comparison made in an iteration is termed successful if the characters compared are identical, contributing to the longest common prefix being computed. Because there is one failed comparison in each iteration, the total number of failed comparisons is $O(n)$. As for successful comparisons, each position in the string is charged only once for a successful comparison. Thus, the total number of comparisons over all iterations is linear in n.

In light of the above discussion, a suffix tree and a suffix array can be constructed from each other in linear time. Thus, a linear time construction algorithm for one can be used to construct the other in linear time. In the following subsections, we explore such algorithms. Each algorithm is interesting in its own right, and exploits interesting properties that could be useful in designing algorithms using suffix trees and suffix arrays.

29.2.2 Linear Time Construction of Suffix Trees

Let s be a string of length n including the termination character \$. Suffix tree construction algorithms start with an empty tree and iteratively insert suffixes while maintaining the property that each intermediate tree represents a compacted trie of the suffixes inserted so far. When all the suffixes are inserted, the resulting tree will be the suffix tree. Suffix links are typically used to speedup the insertion of suffixes. While the algorithms are identified by the names of their respective inventors, the exposition presented does not necessarily follow the original algorithms and we take the liberty to comprehensively present the material in a way we feel contributes to ease of understanding.

McCreight's Algorithm

McCreight's algorithm inserts suffixes in the order $suff_1, suff_2, \ldots, suff_n$. Let T_i denote the compacted trie after $suff_i$ is inserted. T_1 is the tree consisting of a single leaf labeled 1 that is connected to the root by an edge with label $s[1..n]$. In iteration i of the algorithm, $suff_i$ is inserted into tree T_{i-1} to form tree T_i. An easy way to do this is by starting from the root and following the unique path matching characters in $suff_i$ one by one until no more matches are possible. If the traversal does not end at an internal node, create an internal node there. Then, attach a leaf labeled i to this internal node and use the unmatched portion of $suff_i$ for the edge label. The run-time for inserting $suff_i$ is proportional to $|suff_i| = n - i + 1$. The total run-time of the algorithm is $\Sigma_{i=1}^{n}(n - i + 1) = O(n^2)$.

In order to achieve an $O(n)$ run-time, suffix links are used to significantly speedup the insertion of a new suffix. Suffix links are useful in the following way $-$ Suppose we are inserting $suff_i$ in T_{i-1} and let v be an internal node in T_{i-1} on the path from root to leaf labeled $(i-1)$. Then, *path-label*$(v) = c\alpha$ is a prefix of $suff_{i-1}$. Since v is an internal node, there must be another suffix $suff_j$ $(j < i-1)$ that also has $c\alpha$ as prefix. Because $suff_{j+1}$ is previously inserted, there is already a path from the root in T_{i-1} labeled α. To insert $suff_i$ faster, if the end of path labeled α is quickly found, comparison of characters in $suff_i$ can start beyond the prefix α. This is where suffix links will be useful. The algorithm must also construct suffix links prior to using them.

LEMMA 29.1 Let v be an internal node in $ST(s)$ that is created in iteration $i-1$. Let *path-label*$(v) = c\alpha$, where c is a character and α is a (possibly empty) string. Then, either there exists an internal node u with *path-label*$(u) = \alpha$ or it will be created in iteration i.

Proof As v is created when inserting $suff_{i-1}$ in T_{i-2}, there exists another suffix $suff_j$ $(j < i-1)$ such that *lcp*$(suff_{i-1}, suff_j) = c\alpha$. It follows that *lcp*$(suff_i, suff_{j+1}) = \alpha$. The tree T_i already contains $suff_{j+1}$. When $suff_i$ is inserted during iteration i, internal node u with path-label α is created if it does not already exist.

The above lemma establishes that the suffix link of a newly created internal node can be established in the next iteration.

The following procedure is used when inserting $suff_i$ in T_{i-1}. Let v be the internal node to which $suff_{i-1}$ is attached as a leaf. If v is the root, insert $suff_i$ using character comparisons starting with the first character of $suff_i$. Otherwise, let $\textit{path-label}(v) = c\alpha$. If v has a suffix link from it, follow it to internal node u with path-label α. This allows skipping the comparison of the first $|\alpha|$ characters of $suff_i$. If v is newly created in iteration $i-1$, it would not have a suffix link yet. In that case, walk up to parent v' of v. Let β denote the label of the edge connecting v' and v. Let $u' = SL(v')$ unless v' is the root, in which case let u' be the root itself. It follows that $\textit{path-label}(u')$ is a prefix of $suff_i$. Furthermore, it is guaranteed that there is a path below u' that matches the next $|\beta|$ characters of $suff_i$. Traverse $|\beta|$ characters along this path and either find an internal node u or insert an internal node u if one does not already exist. In either case, set $SL(v) = u$. Continue by starting character comparisons skipping the first $|\alpha|$ characters of $suff_i$.

The above procedure requires two different types of traversals − one in which it is known that there exists a path below that matches the next $|\beta|$ characters of $suff_i$ (type I), and the other in which it is unknown how many subsequent characters of $suff_i$ match a path below (type II). In the latter case, the comparison must proceed character by character until a mismatch occurs. In the former case, however, the traversal can be done by spending only $O(1)$ time per edge irrespective of the length of the edge label. At an internal node during such a traversal, the decision of which edge to follow next is made by comparing the next character of $suff_i$ with the first characters of the edge labels connecting the node to its children. However, once the edge is selected, the entire label or the remaining length of β must match, whichever is shorter. Thus, the traversal can be done in constant time per edge, and if the traversal stops within an edge label, the stopping position can also be determined in constant time.

The insertion procedure during iteration i can now be described as follows: Start with the internal node v to which $suff_{i-1}$ is attached as a leaf. If v has a suffix link, follow it and perform a type II traversal to insert $suff_i$. Otherwise, walk up to v's parent, take the suffix link from it unless it is the root, and perform a type I traversal to either find or create the node u which will be linked from v by a suffix link. Continue with a type II traversal below u to insert $suff_i$.

LEMMA 29.2 The total time spent in type I traversals over all iterations is $O(n)$.

Proof A type I traversal is performed by walking down along a path from root to a leaf in $O(1)$ time per edge. Each iteration consists of walking up at most one edge, following a suffix link, and then performing downward traversals (either type II or both type I and type II). Recall that if $SL(v) = u$, then $\textit{tree-depth}(u) \geq \textit{tree-depth}(v) - 1$. Thus, following a suffix link may reduce the depth in the tree by at most one. It follows that the operations that may cause moving to a higher level in the tree cause a decrease in depth of at most 2 per iteration. As both type I and type II traversals increase the depth in the tree and there are at most n levels in ST, the total number of edges traversed by type I traversals over all the iterations is bounded by $3n$.

LEMMA 29.3 The total time spent in type II traversals over all iterations is $O(n)$.

Proof In a type II traversal, a suffix of the string $suff_i$ is matched along a path in T_{i-1}

until there is a mismatch. When a mismatch occurs, an internal node is created if there does not exist one already. Then, the remaining part of $suff_i$ becomes the edge label connecting leaf labeled i to the internal node. Charge each successful comparison of a character in $suff_i$ to the corresponding character in the original string s. Note that a character that is charged with a successful comparison is never charged again as part of a type II traversal. Thus, the total time spent in type II traversals is $O(n)$.

The above lemmas prove that the total run-time of McCreight's algorithm is $O(n)$. McCreight's algorithm is suitable for constant sized alphabets. The dependence of the run-time and space for storing suffix trees on the size of the alphabet $|\Sigma|$ is as follows: A simple way to allocate space for internal nodes in a suffix tree is to allocate $|\Sigma|+1$ pointers for children, one for each distinct character with which an edge label may begin. With this approach, the edge label beginning with a given character, or whether an edge label exists with a given character, can be determined in $O(\log|\Sigma|)$ time. However, as all $|\Sigma|+1$ pointers are kept irrespective of how many children actually exist, the total space is $O(|\Sigma|n)$. If the tree is stored such that each internal node points only to its leftmost child and each node also points to its next sibling, if any, the space can be reduced to $O(n)$, irrespective of $|\Sigma|$. With this, searching for a child connected by an edge label with the appropriate character takes $O(|\Sigma|)$ time. Thus, McCreight's algorithm can be run in $O(n \log|\Sigma|)$ time using $O(n|\Sigma|)$ space, or in $O(n|\Sigma|)$ time using $O(n)$ space.

Generalized Suffix Trees

McCreight's algorithm can be easily adapted to build the generalized suffix tree for a set $\mathcal{S} = \{s_1, s_2, \ldots, s_k\}$ of strings of total length N in $O(N)$ time. A simple way to do this is to construct the string $S = s_1\$_1 s_2\$_2 \ldots s_k\$_k$, where each $\$_i$ is a unique string termination character that does not occur in any string in $\mathcal{S}$. Using McCreight's algorithm, $ST(S)$ can be computed in $O(N)$ time. This differs from $GST(\mathcal{S})$ in the following way: Consider a suffix $suff_j$ of string s_i in $GST(\mathcal{S})$. The corresponding suffix in $ST(S)$ is $s_i[j..|s_i|]\$_i s_{i+1}\$_{i+1} \ldots s_k\$_k$. Let v be the last internal node on the path from root to leaf representing this suffix in $ST(S)$. As each $\$_i$ is unique and *path-label*(v) must be a common prefix of at least two suffixes in S, *path-label*(v) must be a prefix of $s_i[j..|s_i|]$. Thus, by simply shortening the edge label below v to terminate at the end of the string s_i and attaching a common termination character \$ to it, the corresponding suffix in $GST(\mathcal{S})$ can be generated in $O(1)$ time. Additionally, all suffixes in $ST(S)$ that start with some $\$_i$ should be removed and replaced by a single suffix \$ in $GST(\mathcal{S})$. Note that the suffixes to be removed are all directly connected to the root in $ST(S)$, allowing easy $O(1)$ time removal per suffix. Thus, $GST(\mathcal{S})$ can be derived from $ST(S)$ in $O(N)$ time.

Instead of first constructing $ST(S)$ and shortening edge labels of edges connecting to leaves to construct $GST(\mathcal{S})$, the process can be integrated into the tree construction itself to directly compute $GST(\mathcal{S})$. When inserting the suffix of a string, directly set the edge label connecting to the newly created leaf to terminate at the end of the string, appended by \$. As each suffix that begins with $\$_i$ in $ST(S)$ is directly attached to the root, execution of McCreight's algorithm on S will always result in a downward traversal starting from the root when a suffix starting from the first character of a string is being inserted. Thus, we can simply start with an empty tree, insert all the suffixes of one string using McCreight's algorithm, insert all the suffixes of the next string, and continue this procedure until all strings are inserted. To insert the first suffix of a string, start by matching the unique path in the current tree that matches with a prefix of the string until no more matches are possible, and insert the suffix by branching at this point. To insert the remaining suffixes, continue as described in constructing the tree for one string.

This procedure immediately gives an algorithm to maintain the generalized suffix tree of a set of strings in the presence of insertions and deletions of strings. Insertion of a string is the same as executing McCreight's algorithm on the current tree, and takes time proportional to the length of the string being inserted. To delete a string, we must locate the leaves corresponding to all the suffixes of the string. By mimicking the process of inserting the string in GST using McCreight's algorithm, all the corresponding leaf nodes can be reached in time linear in the size of the string to be deleted. To delete a suffix, examine the corresponding leaf. If it is multiply labeled, it is enough to remove the label corresponding to the suffix. It it has only one label, the leaf and edge leading to it must be deleted. If the parent of the leaf is left with only one child after deletion, the parent and its two incident edges are deleted by connecting the surviving child directly to its grandparent with an edge labeled with the concatenation of the labels of the two edges deleted. As the adjustment at each leaf takes $O(1)$ time, the string can be deleted in time proportional to its length.

Suffix trees were invented by Weiner [23], who also presented the first linear time algorithm to construct them for a constant sized alphabet. McCreight's algorithm is a more space-economical linear time construction algorithm [19]. A linear time on-line construction algorithm for suffix trees is invented by Ukkonen [22]. In fact, our presentation of McCreight's algorithm also draws from ideas developed by Ukkonen. A unified view of these three suffix tree construction algorithms is studied by Giegerich and Kurtz [10]. Farach [7] presented the first linear time algorithm for strings over integer alphabets. The algorithm recursively constructs suffix trees for all odd and all even suffixes, respectively, and uses a clever strategy for merging them. The complexity of suffix tree construction algorithms for various types of alphabets is explored in [8].

29.2.3 Linear Time Construction of Suffix Arrays

Suffix arrays were proposed by Manber and Myers [18] as a space-efficient alternative to suffix trees. While suffix arrays can be deduced from suffix trees, which immediately implies any of the linear time suffix tree construction algorithms can be used for suffix arrays, it would not achieve the purpose of economy of space. Until recently, the fastest known direct construction algorithms for suffix arrays all required $O(n \log n)$ time, leaving a frustrating gap between asymptotically faster construction algorithms for suffix trees, and asymptotically slower construction algorithms for suffix arrays, despite the fact that suffix trees contain all the information in suffix arrays. This gap is successfully closed by a number of researchers in 2003, including Käräkkanen and Sanders [13], Kim *et al.* [15], and Ko and Aluru [16]. All three algorithms work for the case of integer alphabet. Given the simplicity and/or space efficiency of some of these algorithms, it is now preferable to construct suffix trees via the construction of suffix arrays.

Käräkkanen and Sanders' Algorithm

Käräkkanen and Sanders' algorithm is the simplest and most elegant algorithm to date to construct suffix arrays, and by implication suffix trees, in linear time. The algorithm also works for the case of an integer alphabet. Let s be a string of length n over the alphabet $\Sigma = \{1, 2, \ldots, n\}$. For convenience, assume n is a multiple of three and $s[n+1] = s[n+2] = 0$. The algorithm has the following steps:

1. Recursively sort the $\frac{2}{3}n$ suffixes $suff_i$ with $i \; mod \; 3 \neq 0$.
2. Sort the $\frac{1}{3}n$ suffixes $suff_i$ with $i \; mod \; 3 = 0$ using the result of step (1).
3. Merge the two sorted arrays.

To execute step (1), first perform a radix sort of the $\frac{2}{3}n$ triples $(s[i], s[i+1], s[i+2])$ for each $i\ mod\ 3 \neq 0$ and associate with each distinct triple its rank $\in \{1, 2, \ldots, \frac{2}{3}n\}$ in sorted order. If all triples are distinct, the suffixes are already sorted. Otherwise, let $suff_i'$ denote the string obtained by taking $suff_i$ and replacing each consecutive triplet with its corresponding rank. Create a new string s' by concatenating $suff_1'$ with $suff_2'$. Note that all $suff_i'$ with $i\ mod\ 3 = 1$ ($i\ mod\ 3 = 2$, respectively) are suffixes of $suff_1'$ ($suff_2'$, respectively). A lexicographic comparison of two suffixes in s' never crosses the boundary between $suff_1'$ and $suff_2'$ because the corresponding suffixes in the original string can be lexicographically distinguished. Thus, sorting s' recursively gives the sorted order of $suff_i$ with $i\ mod\ 3 \neq 0$.

Step (2) can be accomplished by performing a radix sort on tuples $(s[i], rank(suff_{i+1}))$ for all $i\ mod\ 3 = 0$, where $rank(suff_{i+1})$ denotes the rank of $suff_{i+1}$ in sorted order obtained in step (1).

Merging of the sorted arrays created in steps (1) and (2) is done in linear time, aided by the fact that the lexicographic order of a pair of suffixes, one from each array, can be determined in constant time. To compare $suff_i$ ($i\ mod\ 3 = 1$) with $suff_j$ ($i\ mod\ 3 = 0$), compare $s[i]$ with $s[j]$. If they are unequal, the answer is clear. If they are identical, the ranks of $suff_{i+1}$ and $suff_{j+1}$ in the sorted order obtained in step (1) determines the answer. To compare $suff_i$ ($i\ mod\ 3 = 2$) with $suff_j$ ($i\ mod\ 3 = 0$), compare the first two characters of the two suffixes. If they are both identical, the ranks of $suff_{i+2}$ and $suff_{j+2}$ in the sorted order obtained in step (1) determines the answer.

The run-time of this algorithm is given by the recurrence $T(n) = T\left(\lceil \frac{2n}{3} \rceil\right) + O(n)$, which results in $O(n)$ run-time. Note that the $\frac{2}{3} : \frac{1}{3}$ split is designed to make the merging step easy. A $\frac{1}{2} : \frac{1}{2}$ split does not allow easy merging because when comparing two suffixes for merging, no matter how many characters are compared, the remaining suffixes will not fall in the same sorted array, where ranking determines the result without need for further comparisons. Kim *et al.*'s linear time suffix array construction algorithm is based on a $\frac{1}{2} : \frac{1}{2}$ split, and the merging phase is handled in a clever way so as to run in linear time. This is much like Farach's algorithm for constructing suffix trees [7] by constructing suffix trees for even and odd positions separately and merging them. Both the above linear time suffix array construction algorithms partition the suffixes based on their starting positions in the string. A completely different way of partitioning suffixes based on the lexicographic ordering of a suffix with its right neighboring suffix in the string is used by Ko and Aluru to derive a linear time algorithm [16]. This reduces solving a problem of size n to that of solving a problem of size no more than $\lceil \frac{n}{2} \rceil$, while eliminating the complex merging step. The algorithm can be made to run in only $2n$ words plus $1.25n$ bits for strings over constant alphabet. Algorithmically, Käräkkanen and Sanders' algorithm is akin to mergesort and Ko and Aluru's algorithm is akin to quicksort. Algorithms for constructing suffix arrays in external memory are investigated by Crauser and Ferragina [5].

It may be more space efficient to construct a suffix tree by first constructing the corresponding suffix array, deriving the *Lcp* array from it, and using both to construct the suffix tree. For example, while all direct linear time suffix tree construction algorithms depend on constructing and using suffix links, these are completely avoided in the indirect approach. Furthermore, the resulting algorithms have an alphabet independent run-time of $O(n)$ while using only the $O(n)$ space representation of suffix trees. This should be contrasted with the $O(|\Sigma|n)$ run-time of either McCreight's or Ukkonen's algorithms.

29.2.4 Space Issues

Suffix trees and suffix arrays are space efficient in an asymptotic sense because the memory required grows linearly with input size. However, the actual space usage is of significant concern, especially for very large strings. For example, the human genome can be represented as a large string over the alphabet $\Sigma = \{A, C, G, T\}$ of length over 3×10^9. Because of linear dependence of space on the length of the string, the exact space requirement is easily characterized by specifying it in terms of the number of bytes per character. Depending on the number of bytes per character required, a data structure for the human genome may fit in main memory, may need a moderate sized disk, or might need a large amount of secondary storage. This has significant influence on the run-time of an application as access to secondary storage is considerably slower. It may also become impossible to run an application for large data sizes unless careful attention is paid to space efficiency.

Consider a naive implementation of suffix trees. For a string of length n, the tree has n leaves, at most $n-1$ internal nodes, and at most $2n-2$ edges. For simplicity, count the space required for each integer or a pointer to be one word, equal to 4 bytes on most current computers. For each leaf node, we may store a pointer to its parent, and store the starting index of the suffix represented by the leaf, for $2n$ words of storage. Storage for each internal node may consist of 4 pointers, one each for parent, leftmost child, right sibling and suffix link, respectively. This will require approximately $4n$ words of storage. Each edge label consists of a pair of integers, for a total of at most $4n$ words of storage. Putting this all together, a naive implementation of suffix trees takes $10n$ words or $40n$ bytes of storage.

Several techniques can be used to considerably reduce the naive space requirement of 40 bytes per character. Many applications of interest do not need to use suffix links. Similarly, a pointer to the parent may not be required for applications that use traversals down from the root. Even otherwise, note that a depth first search traversal of the suffix tree starting from the root can be conducted even in the absence of parent links, and this can be utilized in applications where a bottom-up traversal is needed. Another technique is to store the internal nodes of the tree in an array in the order of their first occurrence in a depth first search traversal. With this, the leftmost child of an internal node is found right next to it in the array, which removes the need to store a child pointer. Instead of storing the starting and ending positions of a substring corresponding to an edge label, an edge label can be stored with the starting position and length of the substring. The advantage of doing so is that the length of the edge label is likely to be small. Hence, one byte can be used to store edge labels with lengths < 255 and the number 255 can be used to denote edge labels with length at least 255. The actual values of such labels can be stored in an exceptions list, which is expected to be fairly small. Using several such techniques, the space required per character can be roughly cut in half to about 20 bytes [17].

A suffix array can be stored in just one word per character, or 4 bytes. Most applications using suffix arrays also need the *Lcp* array. Similar to the technique employed in storing edge labels on suffix trees, the entries in *Lcp* array can also be stored using one byte, with exceptions handled using an ordered exceptions list. Provided most of the *lcp* values fit in a byte, we only need 5 bytes per character, significantly smaller than what is required for suffix trees. Further space reduction can be achieved by the use of compressed suffix trees and suffix arrays and other data structures [9, 11]. However, this often comes at the expense of increased run-time complexity.

29.3 Applications

In this section, we present algorithms for several string problems using suffix trees and suffix arrays. While the same run-time bounds can be achieved for many interesting applications with either a suffix tree or a suffix array, there are others which involve a space vs. time trade off. Even in cases where the same run-time bound can be achieved, it is often easier to design the algorithm first for a suffix tree, and then think if the implementation can be done using a suffix array. For this reason, we largely concentrate on suffix trees. The reader interested in reading more on applications of suffix arrays is referred to [1,2].

29.3.1 Pattern Matching

Given a pattern P and a text T, the pattern matching problem is to find all occurrences of P in T. Let $|P| = m$ and $|T| = n$. Typically, $n >> m$. Moreover, T remains fixed in many applications and the query is repeated for many different patterns. For example, T could be a text document and P could represent a word search. Or, T could be an entire database of DNA sequences and P denote a substring of a query sequence for homology (similarity) search. Thus, it is beneficial to preprocess the text T so that queries can be answered as efficiently as possible.

Pattern Matching using Suffix Trees

The pattern matching problem can be solved in optimal $O(m+k)$ time using $ST(T)$, where k is the number of occurrences of P in T. Suppose P occurs in T starting from position i. Then, P is a prefix of $suff_i$ in T. It follows that P matches the path from root to leaf labeled i in ST. This property results in the following simple algorithm: Start from the root of ST and follow the path matching characters in P, until P is completely matched or a mismatch occurs. If P is not fully matched, it does not occur in T. Otherwise, each leaf in the subtree below the matching position gives an occurrence of P. The positions can be enumerated by traversing the subtree in time proportional to the size of the subtree. As the number of leaves in the subtree is k, this takes $O(k)$ time. If only one occurrence is of interest, the suffix tree can be preprocessed in $O(n)$ time such that each internal node contains the label of one of the leaves in its subtree. Thus, the problem of whether P occurs in T or the problem of finding one occurrence can be answered in $O(m)$ time.

Pattern Matching using Suffix Arrays

Consider the problem of pattern matching when the suffix array of the text, $SA(T)$, is available. As before, we need to find all the suffixes that have P as a prefix. As SA is a lexicographically sorted order of the suffixes of T, all such suffixes will appear in consecutive positions in it. The sorted order in SA allows easy identification of these suffixes using binary search. Using a binary search, find the smallest index i in SA such that $suff_{SA[i]}$ contains P as a prefix, or determine that no such suffix is present. If no suffix is found, P does not occur in T. Otherwise, find the largest index $j(\geq i)$ such that $suff_{SA[j]}$ contains P as a prefix. All the elements in the range $SA[i..j]$ give the starting positions of the occurrences of P in T.

A binary search in SA takes $O(\log n)$ comparisons. In each comparison, P is compared with a suffix to determine their lexicographic order. This requires comparing at most $|P| = m$ characters. Thus, the run-time of this algorithm is $O(m \log n)$. Note that while this run-time is inferior to the run-time using suffix trees, the space required by this algorithm is only n words for SA apart from the space required to store the string. Note that the *Lcp* array is not required. Assuming 4 bytes per suffix array entry and one byte per character in the string, the total space required is only $5n$ bytes.

The run-time can be improved to $O(m + \log n)$, by using slightly more space and keeping track of appropriate *lcp* information. Consider an iteration of the binary search. Let $SA[L..R]$ denote the range in the suffix array where the binary search is focused. To begin with, $L = 1$ and $R = n$. At the beginning of an iteration, the pattern P is known to be lexicographically greater than or equal to $suff_{SA[L]}$ and lexicographically smaller than or equal to $suff_{SA[R]}$. Let $M = \lceil \frac{L+R}{2} \rceil$. During the iteration, a lexicographic comparison between P and $suff_{SA[M]}$ is made. Depending on the result, the search range is narrowed to either $SA[L..M]$ or $SA[M..R]$. Assume that $l = |lcp(P, suff_{SA[L]})|$ and $r = |lcp(P, suff_{SA[R]})|$ are known at the beginning of the iteration. Also, assume that $|lcp(suff_{SA[L]}, suff_{SA[M]})|$ and $|lcp(suff_{SA[M]}, suff_{SA[R]})|$ are known. From these values, we wish to determine $|lcp(P, suff_{SA[M]})|$ for use in next iteration, and consequently determine the relative lexicographic order between P and $suff_{SA[M]}$. As SA is a lexicographically sorted array, P and $suff_{SA[M]}$ must agree on at least $min(l, r)$ characters. If l and r are equal, then comparison between P and $suff_{SA[M]}$ is done by starting from the $(l+1)^{th}$ character. If l and r are unequal, consider the case when $l > r$.

> Case I: $l < |lcp(suff_{SA[L]}, suff_{SA[M]})|$. In this case, P is lexicographically greater than $suff_{SA[M]}$ and $|lcp(P, suff_{SA[M]})| = |lcp(P, suff_{SA[L]})|$. Change the search range to $SA[M..R]$. No character comparisons are needed.
>
> Case II: $l > |lcp(suff_{SA[L]}, suff_{SA[M]})|$. In this case, P is lexicographically smaller than $suff_{SA[M]}$ and $|lcp(P, suff_{SA[M]})| = |lcp(Suff_{SA[L]}, suff_{SA[M]})|$. Change the search range to $SA[L..M]$. Again, no character comparisons are needed.
>
> Case III: $l = |lcp(suff_{SA[L]}, suff_{SA[M]})|$. In this case, P agrees with the first l characters of $suff_{SA[M]}$. Compare P and $suff_{SA[M]}$ starting from $(l+1)^{th}$ character to determine $|lcp(P, suff_{SA[M]})|$ and the relative lexicographic order of P and $suff_{SA[M]}$.

Similarly, the case when $r > l$ can be handled such that comparisons between P and $suff_{SA[M]}$, if at all needed, start from $(r+1)^{th}$ character. To start the execution of the algorithm, $lcp(P, suff_{SA[1]})$ and $lcp(P, suff_{SA[n]})$ are computed directly using at most $2|P|$ character comparisons. This ensures $|lcp(P, suff_{SA[L]})|$ and $|lcp(P, suff_{SA[R]})|$ are known at the beginning of the first iteration. This property is maintained for each iteration as L or R is shifted to M but $|lcp(P, suff_{SA[M]})|$ is computed. For now, assume that the required $|lcp(suff_{SA[L]}, suff_{SA[M]})|$ and $|lcp(suff_{SA[R]}, suff_{SA[M]})|$ values are available.

LEMMA 29.4 The total number of character comparisons made by the algorithm is $O(m + \log n)$.

Proof The algorithm makes at most $2m$ comparisons in determining the longest common prefixes between P and $suff_{SA[1]}$ and between P and $suff_{SA[n]}$. Classify the comparisons made in each iteration to determine the longest common prefix between P and $suff_{SA[M]}$ into *successful* and *failed* comparisons. A comparison is considered successful if it contributes the longest common prefix. There is at most one failed comparison per iteration, for a total of at most $\log n$ such comparisons over all iterations. As for successful comparisons, note that the comparisons start with $(max(l, r) + 1)^{th}$ character of P, and each successful comparison increases the value of $max(l, r)$ for next iteration. Thus, each character of P is involved only once in a successful comparison. The total number of character comparisons is at most $3m + \log n = O(m + \log n)$.

It remains to be described how the $|lcp(suff_{SA[L]}, suff_{SA[M]})|$ and $|lcp(suff_{SA[R]}, suff_{SA[M]})|$ values required in each iteration are computed. Suppose the Lcp array of T is known. For any $1 \leq i < j \leq n$,

$$|lcp(suff_{SA[i]}, suff_{SA[j]})| = min_{k=i}^{j-1} Lcp[k]$$

The *lcp* of two suffixes can be computed in time proportional to the distance between them in the suffix array. In order to find the *lcp* values required by the algorithm in constant time, consider the binary tree corresponding to all possible search intervals used by any execution of the binary search algorithm. The root of the tree denotes the interval $[1..n]$. If $[i..j]$ $(j - i \geq 2)$ is the interval at an internal node of the tree, its left child is given by $[i..\lceil\frac{i+j}{2}\rceil]$ and its right child is given by $[\lceil\frac{i+j}{2}\rceil..j]$. The execution of the binary search tree algorithm can be visualized as traversing a path in the binary tree from root to a leaf. If *lcp* value for each interval in the tree is precomputed and recorded, any required *lcp* value during the execution of the algorithm can be retrieved in constant time. The leaf level in the binary tree consists of intervals of the type $[i..i+1]$. The *lcp* values for these $n-1$ intervals is already given by the *Lcp* array. The *lcp* value corresponding to an interval at an internal node is given by the smaller of the *lcp* values at the children. Using a bottom-up traversal, the *lcp* values can be computed in $O(n)$ time. In addition to the *Lcp* array, $n-2$ additional *lcp* values are required to be stored. Assuming approximately 1 byte per *lcp* value, the algorithm requires approximately $2n$ bytes of additional space. As usual, *lcp* values larger than or equal to 255, if any, are stored in an exceptions list and the size of such list should be very small in practical applications.

Thus, pattern matching can be solved in $O(m \log n)$ time using $5n$ bytes of space, or in $O(m + \log n)$ time using $7n$ bytes of space. Abouelhoda *et al.* [2] reduce this time further to $O(m)$ time by mimicking the suffix tree algorithm on a suffix array with some auxiliary information. Using clever implementation techniques, the space is reduced to approximately $6n$ bytes. An interesting feature of their algorithm is that it can be used in other applications based on a top-down traversal of suffix tree.

29.3.2 Longest Common Substrings

Consider the problem of finding a longest substring common to two given strings s_1 of size m and s_2 of size n. To solve this problem, first construct the *GST* of strings s_1 and s_2. A longest substring common to s_1 and s_2 will be the path-label of an internal node with the greatest string depth in the suffix tree which has leaves labeled with suffixes from both the strings. Using a traversal of the *GST*, record the string-depth of each node, and mark each node if it has suffixes from both the strings. Find the largest string-depth of any marked node. Each marked internal node at that depth gives a longest common substring. The total run-time of this algorithm is $O(m + n)$.

The problem can also be solved by using the suffix tree of one of the strings and suffix links. Without loss of generality, suppose the suffix tree of s_2 is given. For each position i in s_1, we find the largest substring of s_1 starting at that position that is also a substring of s_2. For position 1, this is directly computed by matching $suff_1$ of s_1 starting from the root of the suffix tree until no more matches are possible. To determine the longest substring match from position 2, simply walk up to the first internal node, follow the suffix link, and walk down as done in McCreight's algorithm. A similar proof shows that this algorithm runs in $O(m + n)$ time.

Now consider solving the longest common substring problem using the *GSA* and *Lcp* array for strings s_1 and s_2. First, consider a one string variant of this problem − that of computing the longest repeat in a string. This is given by the string depth of the deepest internal node in the corresponding suffix tree. All children of such a node must be leaves. Any consecutive pair of such leaves have the longest repeat as their longest common prefix.

Thus, each largest value in the *Lcp* array reveals a longest repeat in the string. The number of occurrences of a repeat is one more than the number of consecutive occurrences of the corresponding largest value in the *Lcp* array. Thus, all distinct longest repeats, and the number and positions of their occurrences can be determined by a linear scan of the *Lcp* array.

To solve the longest common substring problem, let v denote an internal node with the greatest string depth that contains a suffix from each of the strings. Because such a pair of suffixes need not be consecutive in the suffix array, it might appear that one has to look at nonconsecutive entries in the *Lcp* array. However, the subtree of any internal node that is a child of v can only consist of suffixes from one of the strings. Thus, there will be two consecutive suffixes in the subtree under v, one from each string. Therefore, it is enough to look at consecutive entries in the GSA. In a linear scan of the GSA and *Lcp* arrays, find the largest *lcp* value that corresponds to two consecutive suffixes, one from each string. This gives the length of a longest common substring. The starting positions of the suffixes reveals the positions in the strings where the longest common substring occurs. The algorithm runs in $O(m + n)$ time.

29.3.3 Text Compression

Compression of text data is useful for data transmission and for compact storage. A simple, not necessarily optimal, data compression method is the Ziv-Lempel compression [24, 25]. In this method, the text to be compressed is considered a large string, and a compact representation is obtained by identifying repeats in the string. A simple algorithm following this strategy is as follows: Let T denote the text to be compressed and let $|T| = n$. At some stage during the execution of the compression algorithm, suppose that the string $T[1..i-1]$ is already compressed. The compression is extended by finding the length l_i of a largest prefix of $suff_i$ that is a substring of $T[1..i-1]$. Two cases arise:

1. $l_i = 0$. In this case, a compressed representation of $T[1..i]$ is obtained by appending $T[i]$ to the compressed representation of $T[1..i-1]$.
2. $l_i > 0$. In this case, a compressed representation of $T[1..i+l_i-1]$ is obtained by appending (i, l_i) to the compressed representation of $T[1..i-1]$.

The algorithm is initiated by setting $T[1]$ to be the compressed representation of $T[1..1]$, and continuing the iterations until the entire string is compressed. For example, executing the above algorithm on the string *mississippi* yields the compressed string $mis(3,1)(2,3)(2,1)p$ $(9,1)(2,1)$. The decompression method for such a compressed string is immediate.

Suffix trees can be used to carry out the compression in $O(n)$ time [20]. They can be used in obtaining l_i, the length of the longest prefix of $suff_i$ that is a substring of the portion of the string already seen, $T[1..i-1]$. If j is the starting position of such a substring, then $T[j..j+l_i-1] = T[i..i+l_i-1]$ and $i \geq j + l_i$. It follows that $|lcp(suff_j, suff_i)| \geq l_i$. Let $v = lca(i, j)$, where i and j are leaves corresponding to $suff_i$ and $suff_j$, respectively. It follows that $T[i..i+l_i-1]$ is a prefix of *path-label*(v). Consider the unique path from the root of $ST(T)$ that matches $T[i..i+l_i-1]$. Node v is an internal node in the subtree below, and hence j is a leaf in the subtree below. Thus, l_i is the largest number of characters along the path $T[i..n]$ such that $\exists$ leaf j in the subtree below with $j + l_i \leq i$. Note that any j in the subtree below that satisfies the property $j + l_i \leq i$ is acceptable. If such a j exists, the smallest leaf number in the subtree below certainly satisfies this property, and hence can be chosen as the starting position j.

This strategy results in the following algorithm for finding l_i: First, build the suffix tree of T. Using an appropriate linear time tree traversal method, record the string depth of each node and mark each internal node with the smallest leaf label in its subtree. Let $min(v)$ denote the smallest leaf label under internal node v. To find l_i, walk along the path $T[i..n]$ to identify two consecutive internal nodes u and v such that $min(u) + string\text{-}depth(u) < i$ and $min(v) + string\text{-}depth(v) \geq i$. If $min(v) + string\text{-}depth(u) > i$, then set $l_i = string\text{-}depth(u)$ and set the starting position to be $min(u)$. Otherwise, set $l_i = i - min(v)$ and set the starting position to be $min(v)$.

To obtain $O(n)$ run-time, it is enough to find l_i in $O(l_i)$ time as the next l_i characters of the string are compressed into an $O(1)$ space representation of an already seen substring. Therefore, it is enough to traverse the path matching $T[i..n]$ using individual character comparisons. However, as the path is guaranteed to exist, it can be traversed in $O(1)$ time per edge, irrespective of the length of the edge label.

29.3.4 String Containment

Given a set of strings $\mathcal{S} = \{s_1, s_2, \ldots, s_k\}$ of total length N, the string containment problem is to identify each string that is a substring of some other string. An example application could be that the strings represent DNA sequence fragments, and we wish to remove redundancy. This problem can be easily solved using suffix trees in $O(N)$ time. First, construct the $GST(\mathcal{S})$ in $O(N)$ time. To find if a string s_i is contained in another, locate the leaf labeled $(s_i, 1)$. If the label of the edge connecting the leaf to its parent is labeled with the string \$, s_i is contained in another string. Otherwise, it is not. This can be determined in $O(1)$ time per string.

29.3.5 Suffix-Prefix Overlaps

Suppose we are given a set of strings $\mathcal{S} = \{s_1, s_2, \ldots, s_k\}$ of total length N. The suffix-prefix overlap problem is to identify, for each pair of strings (s_i, s_j), the longest suffix of s_i that is a prefix of s_j. Suffix-prefix overlaps are useful in algorithms for finding the shortest common superstring of a given set of strings. They are also useful in applications such as genome assembly where significant suffix-prefix overlaps between pairs of fragments are used to assemble fragments into much larger sequences.

The suffix-prefix overlap problem can be solved using $GST(\mathcal{S})$ in optimal $O(N + k^2)$ time. Consider the longest suffix α of s_i that is a prefix of s_j. In $GST(\mathcal{S})$, α is an initial part of the path from the root to leaf labeled $(j, 1)$ that culminates in an internal node. A leaf that corresponds to a suffix from s_i should be a child of the internal node, with the edge label \$. Moreover, it must be the deepest internal node on the path from root to leaf $(j, 1)$ that has a suffix from s_i attached in this way. The length of the corresponding suffix-prefix overlap is given by the string depth of the internal node.

Let M be a $k \times k$ output matrix such that $M[i, j]$ should contain the length of the longest suffix of s_i that overlaps a prefix of s_j. The matrix is computed using a depth first search (DFS) traversal of $GST(\mathcal{S})$. The GST is preprocessed to record the string depth of every node. During the DFS traversal, k stacks $A_1, A_2, \ldots, A_k$ are maintained, one for each string. The top of the stack A_i contains the string depth of the deepest node along the current DFS path that is connected with edge label \$ to a leaf corresponding to a suffix from s_i. If no such node exists, the top of the stack contains zero. Each stack A_i is initialized by pushing zero onto an empty stack, and is maintained during the DFS as follows: When the DFS traversal visits a node v from its parent, check to see if v is attached to a leaf with edge label \$. If so, for each i such that string s_i contributes a suffix labeling the leaf, *push* $string\text{-}depth(v)$ on to stack A_i. The string depth of the current node can be easily maintained during the DFS traversal. When the DFS traversal leaves the node v to return back to its parent, again identify each i that has the above property and *pop* the top element from the corresponding stack A_i.

The output matrix M is built one column at a time. When the DFS traversal reaches a leaf labeled $(j, 1)$, the top of stack A_i contains the longest suffix of s_i that matches a prefix of s_j. Thus, column j of matrix M is obtained by setting $M[i, j]$ to the top element of stack S_i. To analyze the run-time of the algorithm, note that each *push* (similarly, *pop*) operation on a stack corresponds to a distinct suffix of one of the input strings. Thus, the total number of *push* and *pop* operations is bounded by $O(N)$. The matrix M is filled in $O(1)$ time per element, taking $O(k^2)$ time. Hence, all suffix-prefix overlaps can be identified in optimal $O(N + k^2)$ time.

29.4 Lowest Common Ancestors

Consider a string s and two of its suffixes $suff_i$ and $suff_j$. The longest common prefix of the two suffixes is given by the path label of their lowest common ancestor. If the string-depth of each node is recorded in it, the length of the longest common prefix can be retrieved from the lowest common ancestor. Thus, an algorithm to find the lowest common ancestors quickly can be used to determine longest common prefixes without a single character comparison. In this section, we describe how to preprocess the suffix tree in linear time and be able to answer lowest common ancestor queries in constant time [3].

Bender and Farach's *lca* ***algorithm***

Let T be a tree of n nodes. Without loss of generality, assume the nodes are numbered $1 \dots n$. Let $lca(i, j)$ denote the lowest common ancestor of nodes i and j. Bender and Farach's algorithm performs a linear time preprocessing of the tree and can answer lca queries in constant time.

Let E be an Euler tour of the tree obtained by listing the nodes visited in a depth first search of T starting from the root. Let L be an array of level numbers such that $L[i]$ contains the tree-depth of the node $E[i]$. Both E and L contain $2n - 1$ elements and can be constructed by a depth first search of T in linear time. Let R be an array of size n such that $R[i]$ contains the index of the first occurrence of node i in E. Let $RMQ_A(i, j)$ denote the position of an occurrence of the smallest element in array A between indices i and j (inclusive). For nodes i and j, their lowest common ancestor is the node at the smallest tree-depth that is visited between an occurrence of i and an occurrence of j in the Euler tour. It follows that

$$lca(i, j) = E[RMQ_L(R[i], R[j])]$$

Thus, the problem of answering lca queries transforms into answering range minimum queries in arrays. Without loss of generality, we henceforth restrict our attention to answering range minimum queries in an array A of size n.

To answer range minimum queries in A, do the following preprocessing: Create $\lfloor \log n \rfloor + 1$ arrays $B_0, B_1, \dots, B_{\lfloor \log n \rfloor}$ such that $B_j[i]$ contains $RMQ_A(i, i + 2^j)$, provided $i + 2^j \le n$. B_0 can be computed directly from A in linear time. To compute $B_l[i]$, use $B_{l-1}[i]$ and $B_{l-1}[i + 2^{l-1}]$ to find $RMQ_A(i, i + 2^{l-1})$ and $RMQ_A(i + 2^{l-1}, i + 2^l)$, respectively. By comparing the elements in A at these locations, the smallest element in the range $A[i..i+2^l]$ can be determined in constant time. Using this method, all the $\lfloor \log n \rfloor + 1$ arrays are computed in $O(n \log n)$ time.

Given an arbitrary range minimum query $RMQ_A(i, j)$, let k be the largest integer such that $2^k \le (j - i)$. Split the range $[i..j]$ into two overlapping ranges $[i..i + 2^k]$ and $[j - 2^k..j]$. Using $B_k[i]$ and $B_k[j - 2^k]$, a smallest element in each of these overlapping ranges can be located in constant time. This will allow determination of $RMQ_A(i, j)$ in constant time. To avoid a direct computation of k, the largest power of 2 that is smaller than or equal to each integer in the range $[1..n]$ can be precomputed and stored in $O(n)$ time. Putting all of this together, range minimum queries can be answered with $O(n \log n)$ preprocessing time and $O(1)$ query time.

The preprocessing time is reduced to $O(n)$ as follows: Divide the array A into $\frac{2n}{\log n}$ blocks of size $\frac{1}{2} \log n$ each. Preprocess each block such that for every pair (i, j) that falls within a block, $RMQ_A(i, j)$ can be answered directly. Form an array B of size $\frac{2n}{\log n}$ that contains the minimum element from each of the blocks in A, in the order of the blocks in A, and record the locations of the minimum in each block in another array C. An arbitrary query $RMQ_A(i, j)$ where i and j do not fall in the same block is answered as follows: Directly find the location of the minimum in the range from i to the end of the block containing it, and also in the range from the beginning of the block containing j to index j. All that remains is to find the location of the minimum in the range of blocks completely contained between i and j. This is done by the corresponding range minimum query in B and using C to find the location in A of the resulting smallest element. To answer range queries in B, B is preprocessed as outlined before. Because the size of B is only $O\left(\frac{n}{\log n}\right)$, preprocessing B takes $O\left(\frac{n}{\log n} \times \log \frac{n}{\log n}\right) = O(n)$ time and space.

It remains to be described how each of the blocks in A is preprocessed to answer range minimum queries that fall within a block. For each pair (i, j) of indices that fall in a block, the corresponding range minimum query is precomputed and stored. This requires computing $O(\log^2 n)$ values per block and can be done in $O(\log^2 n)$ time per block. The total run-time over all blocks is $\frac{2n}{\log n} \times O(\log^2 n) = O(n \log n)$, which is unacceptable. The run-time can be reduced for the special case where the array A contains level numbers of nodes visited in an Euler Tour, by exploiting its special properties. Note that the level numbers of consecutive entries differ by $+1$ or -1. Consider the $\frac{2n}{\log n}$ blocks of size $\frac{1}{2} \log n$. Normalize each block by subtracting the first element of the block from each element of the block. This does not affect the range minimum query. As the first element of each block is 0 and any other element differs from the previous one by $+1$ or -1, the number of distinct blocks is $2^{\frac{1}{2}\log n - 1} = \frac{1}{2}\sqrt{n}$. Direct preprocessing of the distinct blocks takes $\frac{1}{2}\sqrt{n} \times O(\log^2 n) = O(n)$ time. The mapping of each block to its corresponding distinct normalized block can be done in time proportional to the length of the block, taking $O(n)$ time over all blocks.

Putting it all together, a tree T of n nodes can be preprocessed in $O(n)$ time such that lca queries for any two nodes can be answered in constant time. We are interested in an application of this general algorithm to suffix trees. Consider a suffix tree for a string of length n. After linear time preprocessing, lca queries on the tree can be answered in constant time. For a given pair of suffixes in the string, the string-depth of their lowest common ancestor gives the length of their longest common prefix. Thus, the longest common prefix can be determined in constant time, without resorting to a single character comparison! This feature is exploited in many suffix tree algorithms.

29.5 Advanced Applications

29.5.1 Suffix Links from Lowest Common Ancestors

Suppose we are given a suffix tree and it is required to establish suffix links for each internal node. This may become necessary if the suffix tree creation algorithm does not construct suffix links but they are needed for an application of interest. For example, the suffix tree may be constructed via suffix arrays, completely avoiding the construction and use of suffix links for building the tree. The links can be easily established if the tree is preprocessed for *lca* queries.

Mark each internal node v of the suffix tree with a pair of leaves (i, j) such that leaves labeled i and j are in the subtrees of different children of v. The marking can be done in linear time by a bottom-up traversal of the tree. To find the suffix link from an internal node v (other than the root) marked with (i, j), note that $v = lca(i, j)$ and $lcp(suff_i, suff_j) = path\text{-}label(v)$. Let $path\text{-}label(v) = c\alpha$, where c is the first character and α is a string. To establish a suffix link from v, node u with path label α is needed. As $lcp(suff_{i+1}, suff_{j+1}) = \alpha$, node u is given by $lca(i+1, j+1)$, which can be determined in constant time. Thus, all suffix links can be determined in $O(n)$ time. This method trivially extends to the case of a generalized suffix tree.

29.5.2 Approximate Pattern Matching

The simpler version of approximate pattern matching problem is as follows: Given a pattern P ($|P| = m$) and a text T ($|T| = n$), find all substrings of length $|P|$ in T that match P with at most k mismatches. To solve this problem, first construct the GST of P and T. Preprocess the GST to record the string-depth of each node, and to answer *lca* queries in constant time. For each position i in T, we will determine if $T[i..i+m-1]$ matches P with at most k mismatches. First, use an *lca* query $lca((P, 1), (T, i))$ to find the largest substring from position i of T that matches a substring from position 1 and P. Suppose the length of this longest exact match is l. Thus, $P[1..l] = T[i..i+l-1]$, and $P[l+1] \neq T[i+l]$. Count this as a mismatch and continue by finding $lca((P, l+2), (T, i+l+1))$. This procedure is continued until either the end of P is reached or the number of mismatches crosses k. As each *lca* query takes constant time, the entire procedures takes $O(k)$ time. This is repeated for each position i in T for a total run-time of $O(kn)$.

Now, consider the more general problem of finding the substrings of T that can be derived from P by using at most k character insertions, deletions or substitutions. To solve this problem, we proceed as before by determining the possibility of such a match for every starting position i in T. Let $l = string\text{-}depth(lca((P, 1), (T, i)))$. At this stage, we consider three possibilities:

1. Substitution − $P[l+1]$ and $T[i+l]$ are considered a mismatch. Continue by finding $lca((P, l+2), (T, i+l+1))$.
2. Insertion − $T[i+l]$ is considered an insertion in P after $P[l]$. Continue by finding $lca((P, l+1), (T, i+l+1))$.
3. Deletion − $P[l+1]$ is considered a deletion. Continue by finding $lca((P, l+2), (T, i+l))$.

After each *lca* computation, we have three possibilities corresponding to substitution, insertion and deletion, respectively. All possibilities are enumerated to find if there is a sequence of k or less operations that will transform P into a substring starting from position i in T. This takes $O(3^k)$ time. Repeating this algorithm for each position i in T takes $O(3^k n)$ time.

The above algorithm always uses the longest exact match possible from a given pair of positions in P and T before considering the possibility of an insertion or deletion. To prove the correctness of this algorithm, we show that if there is an approximate match of P starting from position i in T that does not use such a longest exact match, then there exists another approximate match that uses only longest exact matches. Consider an approximate match that does not use longest exact matches. Consider the leftmost position j in P and the corresponding position $i+k$ in T where the longest exact match is violated. i.e., $P[j] = T[i+k]$ but this is not used as part of an exact match. Instead, an insertion or deletion is used. Suppose that an exact match of length r is used after the insertion or deletion. We can come up with a corresponding approximate match where the longest match is used and the insertion/deletion is taken after that. This will either keep the number of insertions/deletions the same or reduce the count. If the value of k is small, the above algorithms provide a quick and easy way to solve the approximate pattern matching problem. For sophisticated algorithms with better run-times, see [4, 21].

29.5.3 Maximal Palindromes

A string is called a palindrome if it reads the same forwards or backwards. A substring $s[i..j]$ of a string s is called a maximal palindrome of s, if $s[i..j]$ is a palindrome and $s[i-1] \neq s[j+1]$ (unless $i = 1$ or $j = n$). The maximal palindrome problem is to find all maximal palindromes of a string s.

For a palindrome of odd length, say $2k+1$, define the center of the palindrome to be the $(k+1)^{th}$ character. For a palindrome of even length, say $2k$, define the center to be the position between characters k and $k+1$ of the palindrome. In either case, the palindrome is said to be of radius k. Starting from the center, a palindrome is a string that reads the same in both directions. Observe that each maximal palindrome in a string must have a distinct center. As the number of possible centers for a string of length n is $2n-1$, the total number of maximal palindromes of a string is $2n-1$. All such palindromes can be identified in linear time using the following algorithm.

Let s^r denote the reverse of string s. Construct a GST of the strings s and s^r and preprocess the GST to record string depths of internal nodes and for answering lca queries. Now, consider a character $s[i]$ in the string. The maximal odd length palindrome centered at $s[i]$ is given by the length of the longest common prefix between $suff_{i+1}$ of s and $suff_{n-i+2}$ of s^r. This is easily computed as the string-depth of $lca((s, i+1), (s^r, n-i+2))$ in constant time. Similarly, the maximal even length palindrome centered between $s[i]$ and $s[i+1]$ is given by the length of the longest common prefix between $suff_{i+1}$ of s and $suff_{n-i+1}$ of s^r. This is computed as the string-depth of $lca((s, i+1), (s^r, n-i+1))$ in constant time.

These and many other applications involving strings can be solved efficiently using suffix trees and suffix arrays. A comprehensive treatise of suffix trees, suffix arrays and string algorithms can be found in the textbooks by Gusfield [12], and Crochemore and Rytter [6].

Acknowledgment

This work was supported, in part, by the NSF under grant ACI-0203782.

References

[1] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. The enhanced suffix array and its applications to genome analysis. In *2nd Workshop on Algorithms in Bioinformatics*, pages 449–63, 2002.

[2] M. I. Abouelhoda, E. Ohlebusch, and S. Kurtz. Optimal exact string matching based on suffix arrays. In *International Symposium on String Processing and Information Retrieval*, pages 31–43. IEEE, 2002.

[3] M.A. Bender and M. Farach-Colton. The LCA problem revisited. In *Latin American Theoretical Informatics Symposium*, pages 88–94, 2000.

[4] R. Cole and R. Hariharan. Approximate string matching: A simpler faster algorithm. *SIAM Journal on Computing*, 31:1761–1782, 2002.

[5] A. Crauser and P. Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32(1):1–35, 2002.

[6] M. Crochemore and W. Rytter. *Jewels of Stringology.* World Scientific Publishing Company, Singapore, 2002.

[7] M. Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science*, pages 137–143. IEEE, 1997.

[8] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47, 2000.

[9] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *41th Annual Symposium on Foundations of Computer Science*, pages 390–398. IEEE, 2000.

[10] R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19:331–353, 1997.

[11] R. Grossi and J.S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Symposium on the Theory of Computing*, pages 397–406. ACM, 2000.

[12] D. Gusfield. *Algorithms on Strings Trees and Sequences.* Cambridge University Press, New York, New York, 1997.

[13] J. Kärkkänen and P. Sanders. Simpler linear work suffix array construction. In *International Colloquium on Automata, Languages and Programming*, 2003.

[14] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *12th Annual Symposium, Combinatorial Pattern Matching*, pages 181–92, 2001.

[15] D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *14th Annual Symposium, Combinatorial Pattern Matching*, 2003.

[16] P. Ko and S. Aluru. Space-efficient linear-time construction of suffix arrays. In *14th Annual Symposium, Combinatorial Pattern Matching*, 2003.

[17] S. Kurtz. Reducing the space requirement of suffix trees. *Software - Practice and Experience*, 29(13):1149–1171, 1999.

[18] U. Manber and G. Myers. Suffix arrays: a new method for on-line search. *SIAM Journal on Computing*, 22:935–48, 1993.

[19] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23:262–72, 1976.

[20] M. Rodeh, V.R. Pratt, and S. Even. A linear algorithm for data compression via string matching. *Journal of the ACM*, 28:16–24, 1981.

[21] E. Ukkonen. Approximate string-matching over suffix trees. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *Combinatorial Pattern Matching, 4th Annual Symposium*, volume 684 of *Lecture Notes in Computer Science*, pages 228–242, Padova, Italy, 1993. Springer.

[22] E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14:249–60, 1995.

[23] P. Weiner. Linear pattern matching algorithms. In *14th Symposium on Switching and Automata Theory*, pages 1–11. IEEE, 1973.

[24] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337–343, 1977.

[25] J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Transactions on Information Theory*, 24:530–536, 1978.

30

String Searching

Andrzej Ehrenfeucht
University of Colorado, Boulder

Ross M. McConnell
Colorado State University

30.1 Introduction

Searching for occurrences of a substring in a text is a common operation familiar to anyone who uses a text editor, word processor, or web browser. It is also the case that algorithms for analyzing textual databases can generate a large number of searches. If a text, such as a portion of the genome of an organism, is to be searched repeatedly, it is sometimes the case that it pays to preprocess the text to create a data structure that facilitates the searches. The suffix tree [5] and suffix array [4] discussed in Chapter 29 are examples.

In this chapter, we give some alternatives to these data structures that have advantages over them in some circumstances, depending on what type of searches or analysis of the text are desired, the amount of memory available, and the amount of effort to be invested in an implementation.

In particular, we focus on the problem of finding the locations of all occurrences of a string x in a text t, where the letters of t are drawn from a fixed alphabet Σ, such as the ASCII letter codes.

The *length* of a string x, denoted $|x|$, is the number of characters in it. The *empty string*, denoted λ is the string of length 0 that has no characters in it. If $t = a_1a_2, ..., a_n$ is a text and $p = a_ia_{i+1}...a_j$ is a substring of it, then i is a *starting position* of p in t, and j is an *ending position* of p in t. For instance, the starting positions of abc in $aabcabcaac$ are $\{2, 5\}$, and its ending positions are $\{5, 8\}$. We consider the empty string to have starting and ending positions at $\{0, 1, 2, ..., n\}$, once at each position in the text, and once at position 0, preceding the first character of the text. Let $EndPositions(p, t)$ denote the ending positions of p in t; when t is understood, we may denote it $EndPositions(p)$.

A *deterministic finite automaton* on Σ is a directed graph where each directed edge is labeled with a letter from Σ, and where, for each node, there is at most one edge directed out of the node that is labeled with any given letter. Exactly one of the nodes is designated as a *start node*, and some of the nodes are designated as *accept nodes*. The *label* of a directed path is the word given by the sequence of letters on the path. A deterministic finite automaton is used for representing a set of words, namely, the set of the set of labels of paths from the start node to an accept node.

The first data structure that we examine is the *directed acyclic word graph*. The DAWG is just the deterministic finite automaton representing the set of subwords of a text t. All of its states except for one are accept states. There is no edge from the non-accepting state to any accepting state, so it is convenient to omit the non-accept state when representing the DAWG. In this representation, a string p is a substring of t iff it is the label of a directed path originating at the start node.

There exists a labeling of each node of the DAWG with a set of positions so that the DAWG has the following property:

- Whenever p is a substring of t, its ending positions in t are given by the label of the last node of the path of label p that originates at the start node.

To find the locations where p occurs, one need only begin at the start node, follow edges that match the letters of p in order, and retrieve the set of positions at the node where this process halts.

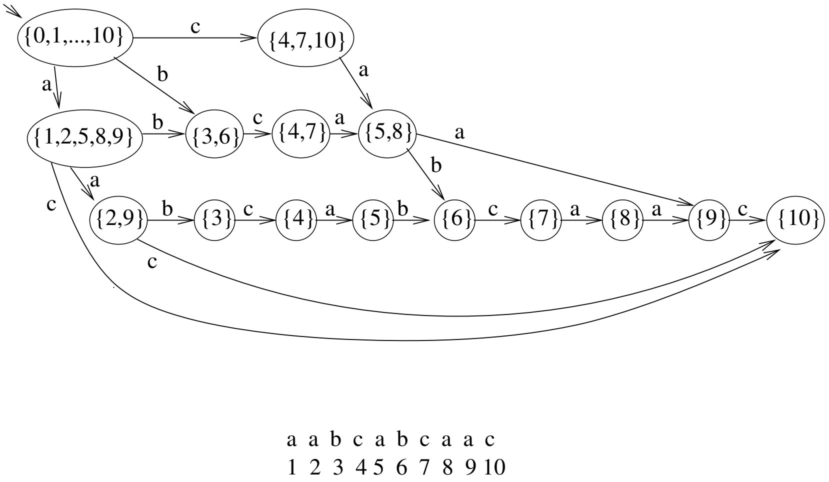


FIGURE 30.1: The DAWG of the text *aabcabcaac*. The starting node is at the upper left. A string p is a substring of the text if and only if it is the label of a path originating at the start node. The nodes can be labeled so that whenever p is the label of such a path, the last node of the path gives $EndPositions(p)$. For instance, the strings that lead to the state labeled $\{5, 8\}$ are *ca*, *bca*, and *abca*, and these have occurrences in the text with their last letter at positions 5 and 8.

In view of the fact that there are $\Theta(|t|^2)$ intervals on t, each of which represents a substring that is contained in the interval, it is surprising that the number of nodes and edges of the `DAWG` of t is $O(|t|)$. The reason for this is that all possible query strings fall naturally into *equivalence classes*, which are sets of strings such that two strings are in the same set if they have the same set of ending positions. The size of an equivalence class can be large, and this economy makes the $O(|t|)$ bound possible.

In an application such as a search engine, one may be interested not in the locations of a string in a text, but the number of occurrences of a string in the text. This is one criterion for deciding which texts are most relevant to a query. Since all strings in an equivalence class have the same number of occurrences, each state can be labeled not with the position set, but with the cardinality of its position set. The label of the node reached on the path labeled p originating at the start node tells the number of occurrences of p in t in $O(|p|)$ time. This variant require $O(|t|)$ space and can be constructed in $O(|t|)$ time.

Unfortunately, the sum of cardinalities of the position sets of the nodes of the DAWG of t is not $O(|t|)$. However, a second data structure that we describe, called the *compact DAWG* does use $O(|t|)$ space. If a string p has k occurrences in t, then it takes $O(|p| + k)$ time to return the set of occurrences where p occurs in t, given the compact DAWG of t. It can be built in $O(|t|)$ time. These bounds are the same as that for the suffix tree and suffix array, but the compact DAWG requires substantially less space in most cases. An example is illustrated in Figure 30.2.

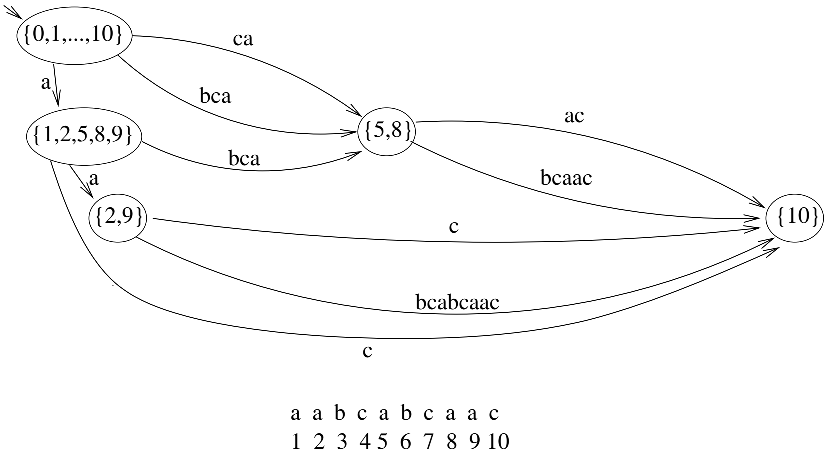


FIGURE 30.2: The compact DAWG of the text *aabcabcaac*. (Compare to Figure 30.1.) The labels depicted in the nodes are the ending positions of the corresponding principal nodes of the DAWG. The compact DAWG is obtained from the DAWG by deleting nodes that have only one outgoing edge, and representing deleted paths between the remaining nodes with edges that are labeled with the path's label.

Another important issue is the ease with which a programmer can understand and program the construction algorithm. Like the computer time required for queries, the time spent by a programmer understanding, writing, and maintaining a program is also a resource that must be considered. The third data structure that we present, called the *position heap*, has worse worst-case bounds for construction and queries, but has the advantage of being as easy to understand and construct as elementary data structures such as unbalanced binary search trees and heaps. One tradeoff is that the worst-case bounds for a query is $O(|p|^2 + k)$, rather than $O(|p| + k)$. However, on randomly generated strings, the expected time for a query is $O(|p| + k)$, and on most practical applications, the query time can be expected not to differ greatly from this. Like the other structures, it can be constructed in linear time. However, an extremely simple implementation takes $O(|t| \log |t|)$ expected time on randomly generate strings, and does not depart much from this in most practical applications. Those who wish to expend minimal programming effort may wish to consider this simple variant of the construction algorithm.

The position heap for the string of Figure 30.1 is illustrated in Figure 30.3.

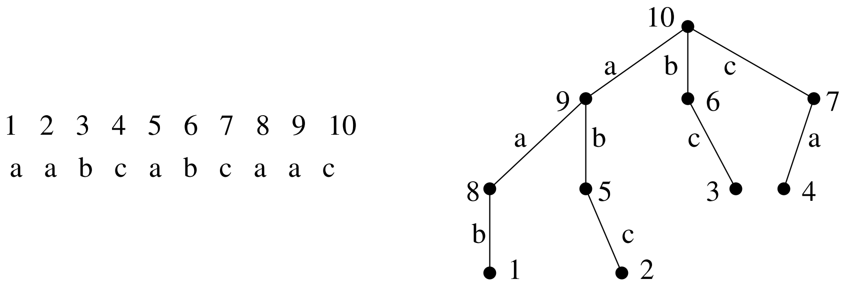


FIGURE 30.3: The position heap of *aabcabcaa*.

30.2 Preliminaries

The infinite set of all strings that can be formed from letters of an alphabet Σ is denoted Σ^*. If $a \in \Sigma$, let a^n denote the string that consists of n repetitions of a.

If x is a string, then for $1 \leq j \leq |x|$, let x_j denote the character in position j. Thus, x can be written as $x_1 x_2, ..., x_{|x|}$. The *reversal* x^R of x is the string $x_{|x|} x_{|x|-1} ... x_1$. Let $x[i:j]$ denote the substring $x_i x_{i+1}, ..., x_j$.

The *prefixes* of a string $x = x_1 x_2, ..., x_k$ are those with a starting position at the leftmost position of x, namely, the empty string and those strings of the form $x[1:j]$ for $1 \leq j \leq k$. Its *suffixes* are those with an ending position at the rightmost position of x, namely, the empty string and those of the form $x[j:k]$.

A *trie* on Σ is a deterministic finite automaton that is a rooted tree whose start node is the root.

Given a family $\mathcal{F}$ of subsets of a domain $\mathcal{V}$, the *transitive reduction* of the subset relation can be viewed as a pointer from each $X \in \mathcal{F}$ to each $Y \in \mathcal{F}$ such that $X \subset Y$ and there exists no Z such that $X \subset Z \subset Y$. This is sometimes referred to as the *Hasse diagram* of the subset relation on the family. The Hasse diagram is a tree if $V \in \mathcal{F}$, $\emptyset \not\in \mathcal{F}$, and for each $X, Y \in \mathcal{F}$, either $X \subseteq Y$, $Y \subset X$, or $X \cap Y = \emptyset$.

30.3 The DAWG

LEMMA 30.1 Let x and y be two strings such that $EndPositions(x) \cap EndPositions(y) \neq \emptyset$. One of x and y must be a suffix of the other, and either $EndPositions(x) = EndPositions(y)$, $EndPositions(x) \subset EndPositions(y)$ or $EndPositions(y) \subset EndPositions(x)$.

Proof If x and y have a common ending position i, then the two occurrences coincide in a way that forces one to be a suffix of the other. Suppose without loss of generality that y is a suffix of x. Then every occurrence of x contains an occurrence of y inside of it that ends at the same position, so $Endpositions(x) \subseteq Endpositions(y)$. $\diamondsuit$

For instance, in the string $aabcabcaac$, the string ca has ending positions $\{5, 8\}$, while the string $aabca$ has ending positions $\{5\}$, and ca is a suffix of $aabca$.

Let x's *right-equivalence class* in t be the set $\{y|EndPositions(y) = EndPositions(x)\}$. The only infinite class is *degenerate class* of strings with the empty set as ending positions, namely those elements of Σ^* that are not substrings of t.

The right-equivalence classes on t are a partition of Σ^*: each member of Σ^* is in one and only one right-equivalence class. By Lemma 30.1, whenever two strings are in the same nondegenerate right-equivalence class, then one of them is a suffix of the other. It is easily seen that if y is the shortest string in the class and x is the longest, then the class consists of the suffixes of x whose length is at least $|y|$. For instance, in Figure 30.1, the class of strings with end positions $\{5, 8\}$ consists of $y = ca$, $x = abca$, and since bca is a longer suffix of x than y is.

LEMMA 30.2 A text t of length n has at most $2n$ right-equivalence classes.

Proof The degenerate class is one right equivalence class. All others have nonempty ending positions, and we must show that there are at most $2n - 1$ of them. The set $V = \{0, 1, 2, ..., n\}$ is the set of ending positions of the empty string. If X and Y are sets of ending positions of two right-equivalence classes, then $X \subseteq Y$, $Y \subseteq X$, or $Y \cap X = \emptyset$, by Lemma 30.1. Therefore, the transitive reduction (Hasse diagram) of the subset relation on the nonempty position sets is a tree rooted at V. For any i such that $\{i\}$ is not a leaf, we can add $\{i\}$ as a child of the lowest set that contains i as a member. The leaves are now a partition of $\{1, 2, ..., n\}$ so it has at most n leaves. Since each node of the tree has at least two children, there are at most $2n - 1$ nodes. $\diamondsuit$

DEFINITION 30.1 The DAWG is defined as follows. The states of the DAWG are the nondegenerate right-equivalence classes that t induces on its substrings. For each $a \in \Sigma$ and $x \in \Sigma^*$ such that xa is a substring of t, there is an edge labeled a from x's right-equivalence class to xa's right-equivalence class.

Figure 30.1 depicts the DAWG by labeling each right-equivalence class with its set of ending positions. The set of words in a class is just the set of path labels of paths leading from the source to a class. For instance, the right-equivalence class represented by the node labeled $\{5, 8\}$ is $\{ca, bca, abca\}$.

It would be natural to include the infinite degenerate class of strings that do not occur in t. This would ensure that every state had an outgoing edge for every letter of Σ. However, it is convenient to omit this state when representing the DAWG: for each $a \in \Sigma$, there is an edge from the degenerate class to itself, and this does not need to be represented explicitly. An edge labeled a from a nondegenerate class to the degenerate class is implied by the absence of an edge out of the state labeled a in the representation.

For each node X and each $a \in \Sigma$, there is at most one transition out of X that is labeled a. Therefore, the DAWG is a deterministic finite automaton. Any word p such that $EndPositions(p) \neq \emptyset$ spells out the labels of a path to the state corresponding to $EndPositions(p)$. Therefore, all states of the DAWG are reachable from the start state.

The DAWG cannot have a directed cycle, as this would allow an infinite set of words to spell out a path, and the set of subwords of t is finite. Therefore, it can be represented by a directed acyclic graph.

A state is a *sink* if it has no outgoing edges. A sink must be the right-equivalence class containing position n, so there is exactly one sink.

THEOREM 30.1 *The DAWG for a text of length n has at most $2n-1$ nodes and $3n-3$ edges.*

Proof The number of nodes follows from Lemma 30.2. There is a single sink, namely, the one that has position set $\{|t|\}$, this represents the equivalence class containing those suffixes of t that have a unique occurrence in t. Let T be a directed spanning tree of the DAWG rooted at the start state. T has one fewer edges than the number of states, hence $2n-2$ edges. For every $e \notin T$, let $P_1(e)$ denote the path in T from the start state to the tail of e, let $P_2(e)$ denote an arbitrary path in the DAWG from the head of e to the sink, and let $P(e)$ denote the concatenation of $(P_1(e), e, P_2(e))$. Since $P(e)$ ends at the sink, the labels of its edges yield a suffix of t. For $e_1, e_2 \notin T$ with $e_1 \neq e_2$, $P(e_1) \neq P(e_2)$, since they differ in their first edge that is not in T. One suffix is given by the labels of the path in T to the sink. Each of the remaining $n-1$ suffixes is the sequence of labels of $P(e)$ for at most one edge $e \notin T$, so there are at most $n-1$ edges not in T.

The total number of edges of the DAWG is bounded by $2n-2$ tree edges and $n-1$ non-tree edges. ♢

To determine whether a string p occurs as a substring of t, one may begin at the start state and either find the path that spells out the letters of p, thereby accepting p, or else reject p if there is no such path. This requires finding, at each node x, the transition labeled a leaving x, where a is the next letter of p. If $|\Sigma| = O(1)$, this takes $O(1)$ time, so it takes $O(|p|)$ time to determine whether p is a subword of t. Note that, in contrast to naive approaches to this problem, this time bound is independent of the length of t.

If the nodes of the DAWG are explicitly labeled with the corresponding end positions, as in Figure 30.1, then it is easy to find the positions where a substring occurs: it is the label of the state reached on the substring. However, doing this is infeasible if one wishes to build the DAWG in $O(|t|)$ time and use $O(|t|)$ storage, since the sum of cardinalities of the position sets can be greater than this. For this problem, it is preferable to use the compact DAWG that is described below.

For the problem of finding the number of occurrences of a substring in t, it suffices to label each node with the *number* of positions in its position set. This may be done in postorder in a depth-first search, starting at the start node, and applying the following rule: the label of a node v is the sum of labels of its out-neighbors, which have already been labeled by the time one must label v. Handling v takes time proportional to the number of edges originating at v, which we have already shown is $O(|t|)$.

30.3.1 A Simple Algorithm for Constructing the DAWG

DEFINITION 30.2 If x is a substring of t, let us say that x's *redundancy* in t in t is the number of ending (or beginning) positions it has in t. If i is a position in t, let $h(i)$ be the longest substring x of t with an ending position at i whose redundancy is at least as great as its length, $|x|$. Let $h(t)$ be the average of $h(i)$ over all i, namely $(\sum_{i=1}^{|t|} h(i))/|t|$.

Clearly, $h(t)$ is a measure of how redundant t is; the greater the value of $h(t)$, the less information it can contain.

In this section, we given an $O(|t|h(t))$ algorithm for constructing the DAWG of a string t. This is quadratic in the worst case, which is illustrated by the string $t = a^n$, consisting of n copies of one letter. However, we claim that the algorithm is a practical one for most applications, where $h(t)$ is rarely large even when t has a long repeated substring. In most applications, $h(t)$ can be expected to behave like an $O(log|t|)$ function.

The algorithm explicitly labels the nodes of the DAWG with their ending positions, as illustrated in Figure 30.1. Each set of ending positions is represented with a list, where the positions appear in ascending order. It begins by creating a start node, and then iteratively *processes* an unprocessed node by creating its neighbors. To identify an unprocessed node, it is convenient to keep a list of the unprocessed nodes, insert a node in this list, and remove a node from the front of the list when it is time to process a new node.

Algorithm 30.2

```
DAWGBuild($t$)
    Create a start node with position set $\{0, 1, ..., n\}$
    While there is an unprocessed node $v$
        Create a copy of $v$'s position set
        Add 1 to every element of this set
        Remove $n + 1$ from this copy if it occurs
        Partition the copy into sets of positions that have a common letter
        For each partition class $W$
            If $W$ is already the position set of a node, then let $w$ denote that node
            Else create a new node $w$ with position set $W$
            Let $a$ be the letter that occurs at the positions in $W$
            Install an edge labeled $a$ from $v$ to $w$
```

Figure 30.4 gives an illustration. For the correctness, it is easy to see by induction on k that every substring w of the text that has length k leads to a node whose position set is the ending positions of w.

LEMMA 30.3 The sum of cardinalities of the position sets of the nodes of the DAWG is $O(|t|h(t))$.

Proof For a position i, let $N(i)$ be the number of ending position sets in which position i appears. By Lemma 30.1, position sets that contain i form a chain $\{X_1, X_2, ..., X_{N(i)}\}$, where for each i from 1 to $N(i)-1$, $|X_i| > |X_{i+1}|$, and a string with X_i as its ending positions must be shorter than one with X_{i+1} as its ending positions. Therefore, $|X_{\lfloor N(i)/2 \rfloor}| \geq N(i)/2$, and any string with this set as its ending position set must have length at least $\lfloor (N(i)/2 \rfloor - 1$. This is a string whose set of ending positions is at least as large as its length, so $N(i) = O(h(t))$,

The sum of cardinalities of the position sets is given by $\sum_{i=0}^{|t|} N(i)$, since each appearance of i in a position set contributes 1 to the sum, and this sum is $O(|t|h(T))$. ♢

It is easy to keep the classes as sorted linked lists. When a class X is partitioned into smaller classes, these fall naturally into smaller sorted lists in time linear in the size of

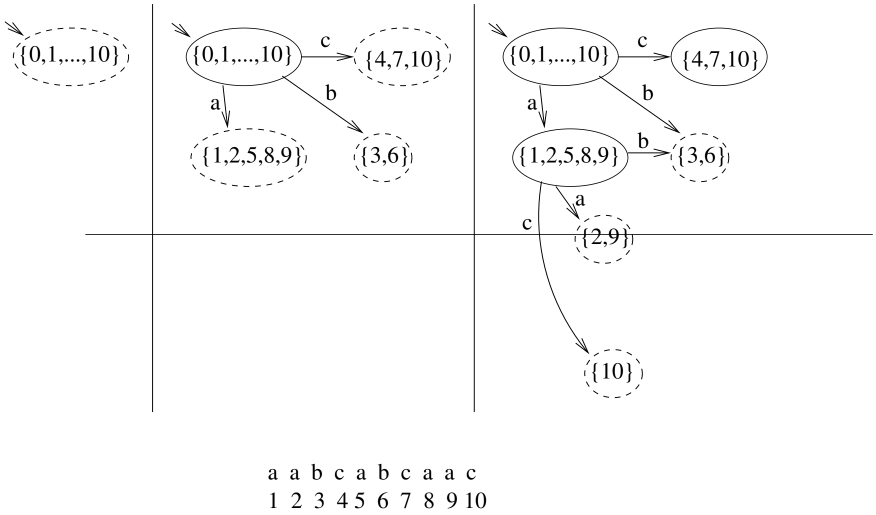


FIGURE 30.4: Illustration of the first three iterations of Algorithm 30.2 on *aabcabcaac*. Unprocessed nodes are drawn with dashed outlines. The algorithm initially creates a start state with position set $\{0, 1, ..., n\}$ (left figure). To process the start node, it creates a copy of this position set, and adds 1 to each element, yielding $\{1, 2, ..., n + 1\}$. It discards $n + 1$, yielding $\{1, 2, ..., n\}$. It partitions this into the set $\{1, 2, 5, 8, 9\}$ of positions that contain a, the set $\{3, 6\}$ of positions that contain b, and the set $\{4, 7, 10\}$ of positions that contain c, creates a node for each, and installs edges labeled with the corresponding letters to the new nodes (middle figure). To process the node v labeled $\{1, 2, 5, 8, 9\}$, it adds 1 to each element of this set to obtain $\{2, 3, 6, 9, 10\}$, and partitions them into $\{2, 9\}$, $\{3, 6\}$, and $\{10\}$. Of these, $\{2, 9\}$ and $\{10\}$ are new position sets, so a new node is created for each. It then installs edges from v to the nodes with these three position sets.

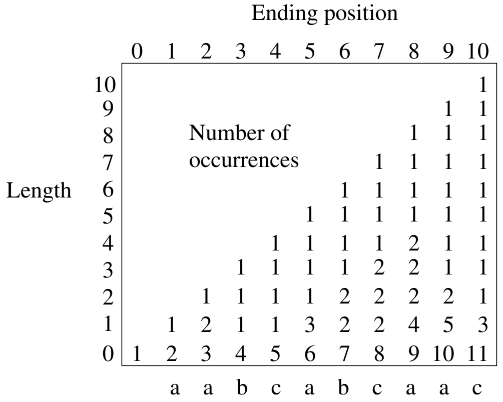
X. A variety of data structures, such as tries, are suitable for looking up whether the sorted representation of a class W already occurs as the position set of a node. The time is therefore linear in the sum of cardinalities of the position sets, which is $O(|t|h(t))$ by Lemma 30.3.

30.3.2 Constructing the DAWG in Linear Time

The linear-time algorithm given in [1] to construct the DAWG works incrementally by induction on the length of the string. The DAWG of a string of length 0 (the null string) is just a single start node. For $k = 0$ to $n - 1$, it iteratively performs an *induction step* that modifies the DAWG of $t[1 : k]$ to obtain the DAWG of $t[1 : k + 1]$.

To gain insight into how the induction step must be performed, consider Figure 30.5. An occurrence of a substring of t can be specified by giving its ending position and its length. For each occurrence of a substring, it gives the number of times the substring occurs up to that point in the text, indexed by length and position. For instance, the string that has length 3 and ends at position 5 is *bca*. The entry in row 3, column 5 indicates that there is one occurrence of it up through position 5 of the text. There is another at position 8, and the entry at row 3 column 8 indicates that it his two occurrences up through position 8.

The lower figure, which we may call the *incremental landscape*, gives a simplified representation of the table, by giving an entry only if it differs from the entry immediately above it. Let $L[i, j]$ denote the entry in row i, column k of the incremental landscape. Some of these entries are blank; the implicit value of such an entry is the value of the first non-blank entry above it.



Length \ Ending position	0	1	2	3	4	5	6	7	8	9	10
10											1
9										1	1
8									1	1	1
7								1	1	1	1
6							1	1	1	1	1
5						1	1	1	1	1	1
4					1	1	1	1	2	1	1
3				1	1	1	1	2	2	1	1
2			1	1	1	1	2	2	2	2	1
1		1	2	1	1	3	2	2	4	5	3
0	1	2	3	4	5	6	7	8	9	10	11
		a	a	b	c	a	b	c	a	a	c

Ending position
Length
Number of
occurrences

FIGURE 30.5: Displaying the number of occurrences of substrings in a text. In the upper figure, the entry in row i column j corresponds to the substring of length j that ends at position i in the text t, and gives the number of occurrences of the substring at position i or before. That is, it gives the number of occurrences of the substring in $t[1 : i]$. Row 0 is included to reflect occurrences of the null substring, which has occurrences at $\{0, 1, ..., n\}$.

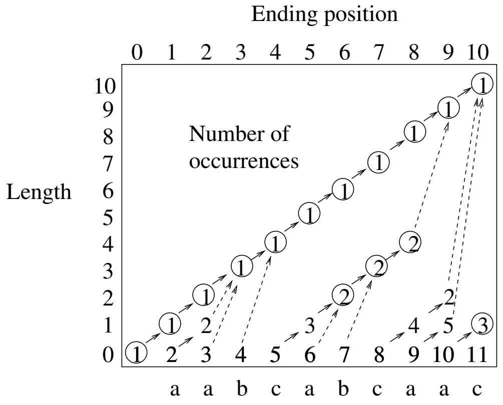


FIGURE 30.6: The incremental landscape is a simplification of the table of Figure 30.5, where an entry is displayed only if it differs from the entry above it. The entries in column i are right-equivalence classes of $t[1 : i]$. These are right-equivalence classes that may be affected during the induction step, when the DAWG of $t[1 : i-1]$ is modified to obtain the DAWG of $t[1 : i]$. Equivalence classes of $t[i : 1]$ that are not right-equivalence classes in $t[1 : i]$ are circled; these correspond to nodes of the DAWG that must be created during the induction step. Edges of the DAWG of $t[1 : i]$ from classes in column $i-1$ are depicted as arrows. (The distinction between solid and dashed arrows is used in the proof of Theorem 30.4.)

Column k has one entry for each right-equivalence class of $t[1:k]$ that has k as an ending position. For instance, in column 8, we see the following:

1. $L[0,8]$: A right-equivalence class for the suffix of $t[1:8]$ of length 0, namely, the empty string, which has 9 occurrences ($\{0,1,...,8\}$) in $t[1:8]$.
2. $L[1,8]$: A right-equivalence class for the suffix of $t[1:8]$ of length 1, namely, the suffix a, which has four occurrences ($\{1,2,5,8\}$) in $t[1:8]$.
3. $L[4,8]$: A right-equivalence class for suffixes of $t[1:8]$ of lengths 2 through 4, namely, $\{ca, bca, abca\}$, which have two occurrences ($\{5,8\}$) in $t[1:8]$. The longest of these, $abca$, is given by the non-blank entry at $L[4,8]$, and membership of the others in the class is given implicitly by the blank entries immediately below it.
4. $L[8,8]$: A right-equivalence class for suffixes of $t[1:k]$ of lengths 5 through 8, namely, $\{cabca, bcabca, abcabca, abcabca\}$ that have one occurrence in $t[1:8]$.

We may therefore treat non-blank entries in the incremental landscape as nodes of the DAWG. Let the *height* of a node denote the length of the longest substring in its right-equivalence class; this is the height (row number) where it appears in the incremental landscape.

When modifying the DAWG of $t[1:k]$ to obtain the DAWG of $t[1:k+1]$, all new nodes that must be added to the DAWG appear in column $k+1$. However, not every node in column $k+1$ is a new node, as some of the entries reflect nodes that were created earlier.

For instance, consider Figure 30.7, which shows the incremental step from $t[1:6]$ to $t[1:7]$. One of the nodes, which represents the class $\{cabc, bcabc, abcabc, aabcabc\}$ of substrings of $t[1:7]$ that are not substrings of $t[1:6]$. It is the top circled node of column 7 in Figure 30.6. Another represents the class $Z_2 = \{c, bc, abc\}$. This appears in $L[3,7]$. To see why this is new, look at the previous occurrence of its longest substring, abc, which is represented by $L[3,4]$, which is blank. Therefore, in the DAWG of $t[1:6]$, it is part of a right-equivalence Z, which appears at $L[4,4]$, and which contains a longer word, $aabc$. Since $\{c, bc, abc\}$ are suffixes of $t[1:7]$ and $aabc$ is not, they cease to be right-equivalent in $t[1:7]$. Therefore, Z must be split into two right-equivalence classes, $Z_2 = \{c, bc, abc\}$ and $Z_1 = Z - Z_2 = \{aabc\}$. Let us call this operation a *split.*

Let us say that a node is *new* in column k if it is created when the DAWG of $t[1:k]$ is modified to obtain the DAWG of $t[1:k+1]$. In Figure 30.6, a node in a column is circled if it is new in that column. In general, a node is new in column k iff it is the top node of the column or the previous occurrence of its longest member corresponds to a blank space in the incremental landscape.

An important point is that only the top two nodes of a column can be new:

LEMMA 30.4 If a new node is the result of a split, only one node lies above it in its column.

Proof Let a be the character that causes the split, and let xa be the largest string in Z_2, and let bxa be the smallest string in $Z_1 = Z - Z_2$. Since bxa previously had the same set of ending positions as xa and now it does not, it must be that xa occurs as a suffix of t_k, but bxa does not. Let cxa be the smallest string in the next higher class C in column $k+1$. On all previous occurrences of xa, it was inside bxa, so the first occurrence of cxa is at position $k+1$. The frequency of the strings in C must be 1, so C is the top class of the column. $\diamondsuit$

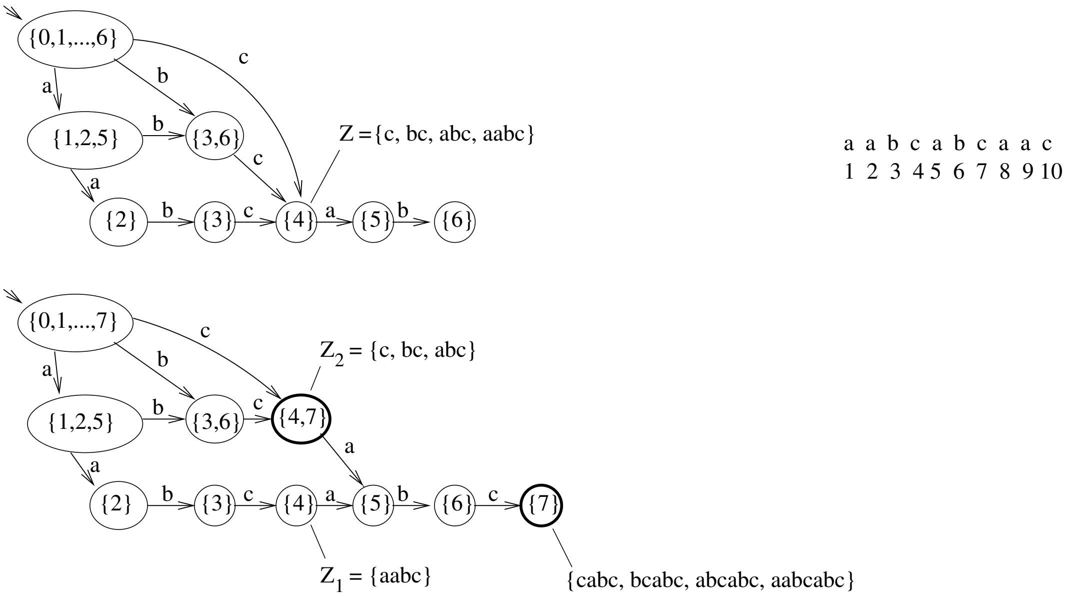


FIGURE 30.7: Modifying the DAWG of $t[1:6] = aabcab$ to obtain the DAWG of $t[1:7] = aabcabc$. New nodes are shown with bold outlines. The sixth column of the incremental landscape, from top to bottom, consists of the nodes $\{6\}$, $\{3,6\}$, and the start node. The seventh column, from top to bottom, consists of $\{7\}$, $\{4,7\}$, and the start node. The node $\{4,7\}$ is *split* from $\{4\}$; of the strings $\{c, bc, abc, aabc\}$ that end at node $\{4\}$, only $\{c, bc, abc\}$ also occur at position 7, so these must now be handled by a different node from the one that handles $aabc$. All edges from nodes in the previous column to $\{4\}$ are redirected to the new node $\{4,7\}$.

The foregoing shows how nodes must be added to the DAWG in the inductive step. In order to understand how the edges of the DAWG must be modified, note that every edge directed into a node in column $k+1$ comes from a node in column k. These edges are given by the following rule:

LEMMA 30.5 In the DAWG of $t[1:k+1]$, a node of height i in column k has an edge labeled t_{k+1} to the lowest node of column $k+1$ that has height greater than i.

These edges are drawn in as solid and dashed arrows in Figure 30.6. According to the figure, when the DAWG of $t[1:7]$ is obtained from $t[1:6]$, the new top node in the column must have an incoming edge from the top node of column 6, which is labeled $\{6\}$ in Figure 30.7. The second new node in column 7, which is labeled $\{4,7\}$ in the figure, must have edges from the nodes at $L[0,6]$ and $L[2,6]$, which are the source and the node labeled $\{3,6\}$. These are obtained by diverting edges into Z.

The induction step consists of implicitly marching in parallel down columns k and $k+1$, creating the required nodes in column t_{k+1} and installing the appropriate edges from right-equivalence classes in column k to right-equivalence classes of column $k+1$, as well as the appropriate outgoing edges and parent pointers on each right-equivalence class in column $k+1$ that is new in t_{k+1}. The new nodes in column $k+1$ are the one with frequency one, and possibly one other, Z_2, that results from a split. By Lemma 30.4, this requires marching down through at most two nodes of column $k+1$, but possibly through many nodes of column k.

To facilitate marching down a column k efficiently, the algorithm needs a *suffix pointer* `Suffix`(x) on each node x of column k to the next lower node in column k. If $y = y_1y_2y_3...y_j$ is the shortest string in the x's right-equivalence class, then `Suffix`(x) points to the right-equivalence class that contains the longest proper suffix $y_2y_3...y_j$ of y. The suffix pointer for each node is uniquely defined, so the algorithm ensures that suffix pointers are available on nodes of column k by keeping suffix pointers current on all nodes of the DAWG.

The induction step is given in Algorithm 30.3. The algorithm does not build the incremental landscape. However, we may identify the nodes by where they would go in the incremental landscape. The meanings of the variables can be summarized as follows. $\texttt{Top}_k$ is the top node of column k, and $\texttt{Top}_{k+1}$ is the top node of column $k+1$. `Mid` denotes the highest node of column k that already has an outgoing labeled with the $(k+1)^{th}$ letter. The variable `curNode` is a looping variable that travels down through nodes of column k, becoming undefined if it travels past the bottom node of the column.

Algorithm 30.3 $\texttt{Update}(\texttt{Top}_k)$*: Update the DAWG of* $t[1:k]$ *to obtain the DAWG of* $t[1:k+1]$.

```
Create a node $\texttt{Top}_{k+1}$ of frequency 1 and height $k+1$
Let curNode = $\texttt{Top}_k$.
While curNode is defined and has no outgoing edge labeled $t_{k+1}$
    Install an edge labeled $t_{k+1}$ from curNode to $\texttt{Top}_{k+1}$.
    curNode := Suffix(curNode)
If curNode is defined
    Mid := curNode
    Let $Z$ be the neighbor of Mid on $t_{k+1}$
    Define Suffix($\texttt{Top}_{k+1}$) to be $Z$
    If height($Z$) > height(Mid) + 1
        Split($k$, Mid, $Z$); Create a second new node in Column $k+1$
Else define Suffix($\texttt{Top}_{k+1}$) to be the start node
Return $Top_{k+1}$
```

Procedure `Split`$(k, \texttt{Mid}, Z)$

```
Create a copy $Z_2$ of the node representing $Z$, together with its outgoing edges
Let the height of $Z_2$ be one plus the height of Mid
Let $curNode$ = Mid

While curNode is defined and $Z$ is its neighbor on $t_{k+1}$
    Divert curNode's edge labeled $t_{k+1}$ so that it points to $Z_2$
    curNode := Suffix(curNode)
Redefine Suffix($Z_2$) to be Suffix($Z$)
Redefine Suffix($Z$) to be $Z_2$
```

THEOREM 30.4 *It takes $O(|t|)$ time to build the DAWG of a text t of length n.*

Proof No node is ever discarded once it is created, and the final DAWG has $O(|t|)$ nodes.

Therefore, the cost of creating nodes is $O(|t|)$. Once an edge is created it remains in the DAWG, though it might be diverted in calls to `Split`. No edge is ever discarded and the final DAWG has $O(|t|)$ edges, so the cost of creating edges is $O(|t|)$.

It remains to bound the cost of diverting edges in calls to `Split`. Let an edge that appears in the incremental landscape be *solid* if it goes from a node of height i to one of height $i+1$, and *dashed* otherwise. (See Figure 30.6.) We may partition the edges in the landscape into *terminating paths*, each of which starts in row 0, and contains zero or more solid edges, and either followed by a dashed edge or ending in the last column. At most one terminating path begins in any column, and every dashed edge terminates a path. Thus, there are at most n dashed edges.

When Z_2 is created in `Split`, at most one of the edges diverted into it is solid. The cost of diverting this edge is subsumed in the cost of creating Z_2. The cost of diverting other edges is $O(|t|)$ over all calls to `Split`, since each of them is one of the at most n dashed edges that appear in the incremental landscape. ♢

30.4 The Compact DAWG

By Theorem 30.1 and Lemma 30.3, we cannot assume that the DAWG requires linear space if the nodes are explicitly labeled with their position sets. The algorithm for building the DAWG in linear time does not label the nodes with their position sets. However, without the labels, it is not possible to use the DAWG to find the k locations where a substring p occurs in t in $O(|p| + k)$ time.

One remedy for this problem is to label a node with a position i if it represents the smallest position set that contains i as a member. The total number of these labels is n. We can reverse the directions of the suffix pointers that are installed during the DAWG construction algorithm, yielding a tree on the position sets. If a node represents a set X of positions, the members of X can be returned in $O(|X|)$ time by traversing the subtree rooted at X, assembling a list of these labels. (This tree is isomorphic to the suffix tree of the reverse of the text, but there is no need to adopt the common practice of labeling each of its edges with a string.)

Another alternative, which has considerable advantage in space requirements over the suffix tree, is to "compact" the DAWG, yielding a smaller data structure that still supports a query about the positions of a substring $O(|p| + k)$ time. The algorithm for compacting it runs in $O(|t|)$ time.

If x is a substring of t, let $\alpha(x)$ denote the longest string y such every ending position of x is also an ending position of yx. That is, y is the maximal string that precedes every occurrence of x in t. Note that $\alpha(x)$ may be the null string. Similarly, let $\beta(x)$ denote the longest string z such that every starting position of x is a starting position of xz. This is the longest string that follows every occurrence of x.

For instance, if $t = aabcabcaac$ and $x = b$, then $\alpha(x) = a$ and $\beta(x) = ca$.

LEMMA 30.6

1. For x and y in a right-equivalence class, $\alpha(x)x = \alpha(y)y$ is the longest string in the class.
2. For x and y in a right-equivalence class, $\beta(x) = \beta(y)$.

Let a substring x of t be *prime* if $\alpha(x)$ and $\beta(x)$ are both the empty string. For any substring x of t, $\alpha(x)x\beta(x)$ is prime; this is the *prime implicant* of x. If x is prime, it is its own prime implicant.

DEFINITION 30.3 The *compact DAWG* of a text t is defined as follows. The nodes are the prime substrings of t. If x is a prime substring, then for each $a \in \Sigma$ such that xa is a substring of t, let $y = \alpha(xa)$ and $z = a\beta(xa)$. There is an edge labeled z from x to the prime implicant yxz of xa.

If a right-equivalence class contains a prime substring x, then x is the longest member of the class. Stretching the terminology slightly, let us call a class *prime* if it contains a prime substring. If C is a right-equivalence class in t, we may define $\beta(C) = \beta(x)$ such that $x \in C$. By Part 2 of Lemma 30.6, $\beta(C)$ is uniquely defined. We may define C's *prime implicant* to be the right-equivalence class D that contains $x\beta(x)$ for $x \in C$. D is also uniquely defined and contains the prime implicant of the members of C.

The nodes of the DAWG may therefore be partitioned into groups that have the same prime implicant. This is illustrated in Figure 30.8.

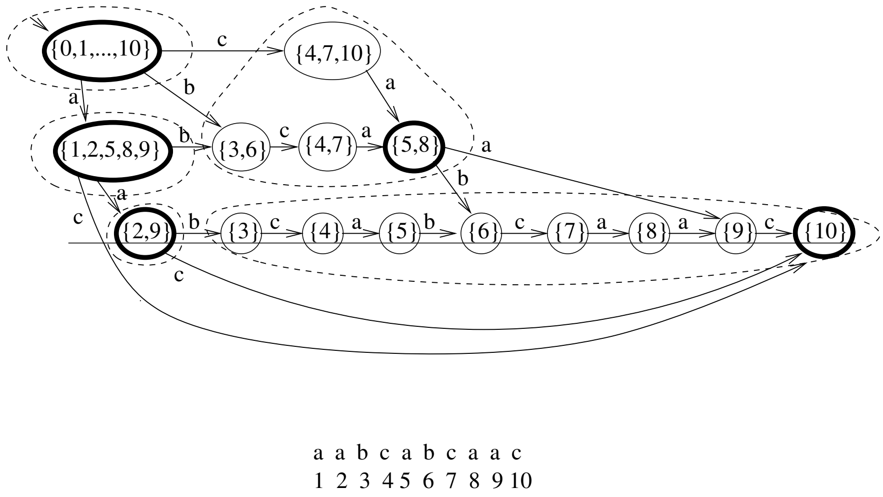


FIGURE 30.8: Partition of nodes into groups with the same prime implicant.

LEMMA 30.7 A right-equivalence class is non-prime if and only if it has exactly one outgoing edge in the DAWG.

We now describe how to obtain the compact DAWG from the DAWG in linear time. For ease of presentation, we describe how to carry it out in four depth-first traversals of the DAWG. However, in practice, only two depth-first traversals are required, since the operations of the first three traversals can be carried out during a single depth-first traversal.

In the first depth-first traversal, we may label each class with a single position from its set of ending positions. This is done in postorder: when retreating from a node, copy its label from the label of any of its successors, which have already been labeled, and subtract

1 from it.

By Lemma 30.7, the prime implicant of a class is the class itself if it is prime; otherwise, it is the unique successor that is prime. Let the *distance* to its prime implicant be the length of this unique path.

In postorder during the second traversal, we may label each node with a pointer to its prime implicant and label this pointer with the distance to the prime implicant. If the class C is a sink or has more than one outgoing edge, this is just a pointer from C to itself with distance label 0. Otherwise, C has a unique successor D, which is already labeled with a pointer to D's prime implicant A with distance label i. Label C with a pointer to A with distance label $i + 1$.

In the third traversal, we install the compact DAWG edges. If we label the edges explicitly with their string labels, we will exceed linear time and storage. Instead, we may take advantage of the fact that the label of every edge is a substring of t. We label each edge with the length of its label. (See Figure 30.9.) When retreating from a prime node B during the traversal, for each DAWG edge (BC) out of B, let D be C's prime implicant, let i be the distance of D from C. Install a compact DAWG edge from B to D that has length label $i + 1$.

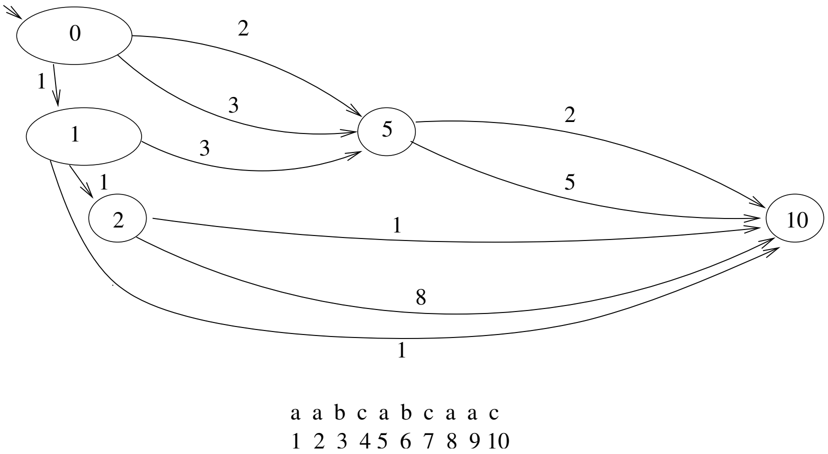


FIGURE 30.9: Representing the edge labels of the compact DAWG. (Compare to Figure 30.2.) Each edge label is a substring of t with end positions at the end position labels of the principal nodes. The label of the edge can therefore be represented implicitly, by labeling each node with one member of its position set, and labeling each edge with the length of its label. For instance, the edge labeled 3 from the source to the node labeled “5” is labeled with the substring of length 3 that ends at position 5, hence, the one occupying positions 3, 4, and 5 of the text. Since the text can be randomly accessed, the text can be used to look up the label of the edge. This ensures that the edge labels take $O(|t|)$ space, since they take $O(1)$ for each node and edge.

On the final traversal, we may remove the DAWG nodes, DAWG edges, and the prime implication pointers.

30.4.1 Using the Compact DAWG to Find the Locations of a String in the Text

Let v be a node of the compact DAWG, and let x be the corresponding prime implicant. Let the *total length* of a path from v to the sink be the sum of the length labels of the edges on the path. Observe that there is a path of total length i from v to the sink iff x has an ending position at $n - i + 1$.

LEMMA 30.8 Let x be a prime substring of t, and let k be the number of occurrences of x in t. Given x's node in the compact DAWG of t, it takes $O(k)$ time to retrieve the ending positions of x in t.

Proof Recursively explore all paths out of the node, and whenever the sink is encountered, subtract the total length of the current path from $n + 1$ and return it.

The running time follows from the following observations: One position is returned for each leaf of the recursion tree; the sets of positions returned by recursive calls are disjoint; and every internal node of the recursion tree has at least two children since every node of the compact DAWG has at least two outgoing edges. ♢

The representation of Figure 30.9 is therefore just as powerful as that of of Figure 30.2: the edge labels are implied by accessing t using the numbers on edges and nodes, while the position labels of the vertices can be retrieved in linear time by the algorithm of Lemma 30.8.

The representation of Figure 30.9 now gives an $O(|p| + k)$ algorithm for finding the k occurrences of a substring p in a text t. One must index into the compact DAWG from the source, matching letters of p with letters of the implicit word labels of the compact edges. If a letter of p cannot be matched, then p does not occur as a subword of t. Otherwise, p is the concatenation of a set of word labels on a path, followed by part of the word label of a final edge (u, v). This takes $O(|p|)$ time. Let i be the number of remaining unmatched letters of the word label of (u, v). The k ending positions of p are given by subtracting i from the k ending positions of v, which can be retrieved in $O(k)$ time using the algorithm of Lemma 30.8.

For instance, using the compact DAWG of Figure 30.2 to find the locations where *abc* occurs, we match a to the label a of an edge out of the source to the node with position set $\{1, 2, 5, 8, 9\}$, then *bc* to the word label of the edge to the node labeled $\{5, 8\}$. Though the node is labeled with the position set in the picture, this position set is not available in the linear-space data structure. Instead, we find two paths of length 2 and 5 from this node to the sink, and subtracting 2 and 5 from $n = 10$ yields the position set $\{5, 8\}$. Then, since one letter in the word label *bca* remains unmatched, we subtract 1 from each member of $\{5, 8\}$ to obtain $\{4, 7\}$, which is the desired answer.

30.4.2 Variations and Applications

In [1], a variation of the compact DAWG is given for a collection $\{t_1, t_2, ..., t_k\}$ of texts, and can be used to find the k occurrences of a string p in the texts in $O(|p| + k)$ time.

That paper also gives a symmetric version of the compact DAWG. By the symmetry in the definition of the prime subwords of t, the set of prime subwords of the reversal of t are given by reversing the set of prime subwords of t. The compact DAWG of t and of the reversal of t therefore have essentially the same set of nodes; only the edges are affected by the reversal. The symmetric version has a single set of nodes and two sets of edges, one corresponding to the edges of the compact DAWG of t and one corresponding to the edges of the reversal of t. The utility of this structure as a tool for exploring the subword structure of t is described in the paper.

Another variant occurs when t is a cyclic ordering of characters, rather than a linear one. A string p has an occurrence anywhere where it matches the subword contained in an interval on this cycle. A variant of the DAWG, compact DAWG, and compact symmetric DAWG for retrieving occurrences of subwords for t in this case is given in [1]. The paper gives algorithms that have time bounds analogous to those given here.

Variations of landscapes, such as that of Figure 30.6 are explored in [2]. They give a graphical display of the structure of repetitions in a text. The suffix tree can be used to find the longest common substring of two texts t_1 and t_2 efficiently. The paper gives $O(|t|h(t))$ algorithms that use the DAWG to generate the landscape of t (see Definition 30.2), which can be used to help identify functional units in a genomic sequence. One variation of the landscape explored in the paper inputs two texts t_1 and t_2, and gives a graphical display of the number of occurrences of every substring of t_1 in t_2, which has obvious applications to the study of evolutionary relationships among organisms.

Mehta and Sahni give a generalization of the compact DAWG and the compact symmetric DAWG to circular sequences is given in [6], and give techniques for analyzing and displaying the structure of strings using the compact symmetric DAWG in [7, 8].

30.5 The Position Heap

We now give a data structure that gives much simpler algorithms, at a cost of slightly increasing the worst-case time required for a query. The algorithms can easily be programmed by undergraduate data-structures students.

The data structure is a trie, and has one node for each position in the text. The data structures and algorithms can be modified to give the same bounds for construction and searching, but this undermines the principal advantages, which are simplicity and low memory requirements.

The data structure is closely related to trees that are used for storing hash keys in [3].

30.5.1 Building the Position Heap

Let a string be *represented* by a trie if it is the label of a path from the root in the trie.

For analyzing the position heap us adopt the convention of indexing the characters of t in descending order, so $t = t_n t_{n-1} ... t_1$. In this case, we let $t[i : j]$ denote $t_i t_{i-1} ... t_j$.

The algorithm for constructing the position heap can be described informally as follows. The positions of t are visited from right to left as a trie is built. At each position i, a new substring z is added to the set of words represented by the trie. To do this, the longest prefix $t[i : j]$ of $t[i : 1]$ that is already represented in the trie is found by indexing into the trie from the root, using the leading letters of $t[i : 1]$, until one can advance no further. A leaf child of the last node of this path is added, and the edge to it is labeled t_{i+1}.

The procedure, `PHBuild`, is given in Table 30.1. Figure 30.10 gives an illustration.

30.5.2 Querying the Position Heap

Table 30.2 gives a procedure, `PHFind`, to find all starting positions of p in t, and Figure 30.11 gives an illustration. The worst-case running time of $O(|p|^2 + k)$ to find the k occurrences of p is worse than the $O(|p| + k)$ bound for the suffix tree or DAWG.

TABLE 30.1 Constructing the position heap for a string $t = t_i t_{i-1} ... t_1$.

```
PHBuild($t, i$)
    If $i = 1$ return a single tree node labeled 1
    Else
        Recursively construct the position heap $H'$ for the suffix $t[i-1, 1]$.
        Let $t' = t[i : k]$ be the maximal prefix of $t$ that is the
            label of a path originating at the root in the tree.
        Let $u$ be the last node of this path.
        Add a child of $u$ to the tree on edge labeled $t_{k-1}$, and give it label $i$.
```

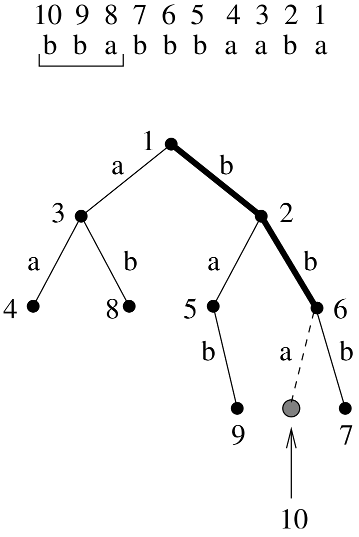


FIGURE 30.10: Construction of the position heap with PHBuild (Table 30.1). The solid edges reflect the recursively-constructed position heap for the suffix $t[9:1]$ of t. To get the position heap for $t[10:1]$, we use the heap to find the largest prefix *bb* of $t[10:1]$ that is the label of a path in the tree, and add a new child at this spot to record the next larger prefix *bba*.

TABLE 30.2 Find all places in a text t where a substring p occurs, given the position heap H for t.

```
PHFind($p, t, H$)
    Let $p'$ be the maximal prefix of $p$ that is the label of a path $P'$ from the root of $H$.
    $S_1$ be the set of position labels in $P'$.
    Let $S_2$ be the subset of $S_1$ that are the positions of occurrences of $p$ in $t$.
    If $p' \neq p$ then let $S_3$ be the empty set
    Else let $S_3$ be the position labels of descendants of the last node of $P'$.
    Return $S_2 \cup S_3$.
```

LEMMA 30.9 PHFind returns all positions where p occurs in t.

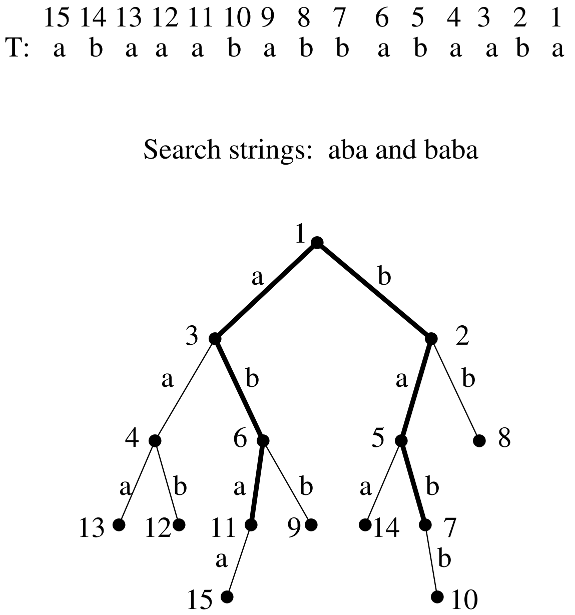


FIGURE 30.11: Searching for occurrences of a string in the text t in $O(|p|^2 + k)$ time with `PHFind` (Table 30.2). How the search is conducted depends on whether the search string is the path label of a node in the position heap. One case is illustrated by search string *aba*, which is the path label of position 11. The only places where *aba* may match t are at positions given by ancestors and descendants of t. The descendants $\{11, 15\}$ do not need to be verified, but the proper ancestors $\{1, 3, 6\}$ must be verified against t at those positions. Of these, only 3 and 6 are matches. The algorithm returns $\{3, 6, 11, 15\}$. The other case is illustrated by *baba*, which is not the path label of a node. Indexing on it yields position 7 and path label $bab \neq baba$. Only the ancestors $\{2, 5, 7\}$ are candidates, and they must be verified against t. Of these, only 7 is a match. The algorithm returns $\{7\}$. Since the ancestor positions occur on the search path, there are $O(|p|)$ of them, and each takes $O(|p|)$ time to verify each of them against t. Descendants can be more numerous, but take $O(1)$ time apiece to retrieve and return, since they do not need to be verified.

Proof Let $p = p_1p_2...p_m$ and let $t = t_nt_{n-1}...t_1$. Suppose that i is a position in t where p does not occur. Then $i \notin S_2$. Any node u with position label i has a path label that is a prefix of $t[i : 1]$. Since p is not a prefix of this string, it is not a prefix of the path label of u, so $i \notin S_3$. We conclude that i is not among the positions returned by `PHFind`.

Next, let h be the position of an occurrence of p. Let $x = p[1 : j]$ be the maximal prefix of p that is represented in the position heap of $t[h-1 : 1]$, where $j = 0$ if $x = \lambda$. If $x \neq p$, then `PHBuild` created a node with position label h and path label xp_{j+1}. This is a prefix of p, so $h \in S_1$, and, since p occurs at position h, $h \in S_2$. If $x = p$, let $y = t[h : k]$ be the largest prefix of $t[h : 1]$ that is active in $t[h-1 : 1]$. Then `PHBuild` created a node with position label h and path label yt_{k-1}, and $h \in S_3$. In either case, h is returned as a position where P occurs. $\diamondsuit$

30.5.3 Time Bounds

LEMMA 30.10 PHFind takes $O(|p|^2+k)$ worst-case time time to return the k occurrences of p in t.

Proof The members of S_3 can be retrieved in $O(1)$ time apiece using a depth-first traversal of the subtree rooted at the last node on path P'. Since all nodes of S_1 occur on a path whose label is a prefix of p, there are at most $m+1$ members of S_1. Checking them against t to see which are members of S_2 takes $O(|p|)$ time apiece, for a total of $O(|p|^2)$ time in the worst case. ♢

This time bound overstates what can be expected in practice, since, in most cases, the string is known to match on a prefix, but there is no reason to expect that it will be similar to the position that it is supposed to match in the region beyond this prefix. A good heuristic is to match the string from the end, rather than from the beginning, since the string has a prefix that is already known to match at the position. Checking to see whether a string matches at a given position will usually require examining one or two characters, discovering a mismatch, and rejecting the string.

LEMMA 30.11 PHBuild takes $O(|t|h(t^R))$ time.

Proof If $P = (v_0, v_1, ..., v_k)$ be a path from the root v_0 in the position heap, let $P_1 = (v_0, v_1, ..., v_{\lfloor k/2 \rfloor})$, and let $P_2 = (v_{\lfloor k/2 \rfloor+1}, v_{\lfloor k/2 \rfloor+2}, ..., v_k)$ be the remainder of the path. Let i be the position label of v_k, and let $h'(i)$ denote the length of the maximum prefix x of $t[i:1]$ that occurs at least $|x|$ times in t. The path label y of P_1 has an occurrence at the positions labels of each of its descendants, including those on P_2, of which there are at least $|y|$. Therefore, Therefore, $|y| = O(h'(i))$. The time spent by the algorithm at position i of t is proportional to the length of P, which is $O(|y|)$. Therefore, the time spent by the algorithm adding the node for position i is $O(h'(i))$, hence the time to build the whole heap is $O(\sum_{i=1}^{|t|} h'(i)) = O(|t|h(t^R))$ by Definition 30.2.

As with the $O(|t|h(t))$ algorithm for building the DAWG, this time bound is a practical one in most settings, since $h(t)$ is relatively insensitive to long repeated strings or localized areas of the string with many repetitions. Only strings where most areas of the string are repeated many times elsewhere have high values of $h(t)$, and $h(t)$ can be expected to behave like an $O(\log n)$ function in most settings.

30.5.4 Improvements to the Time Bounds

In this section, we have given an algorithm for constructing the position heap to $O(|t|)$. We also sketch an approach for finding the occurrences of a string p in t to $O(|p|+k)$ using position heaps. Each of these have tradeoff costs, such as having greater space requirements and being harder to understand.

The position heap has a dual, which we may call the **dual heap** (see Figure 30.12). They have the same node set: a node has path label x in the heap iff its path label in the dual is the reversal x^R of x. We will refer to the position heap as the **primal heap** when we wish to contrast it to the dual.

It is tempting to think that the dual is just the position heap of the reversal t^R of t, but this is not the case. As in the primal heap, the rightmost positions of t are near the root of the dual, but in the primal heap of t^R, the leftmost positions of t are near the root. In the primal heap of t^R the heap order is inverted, which affects the shape of the tree. Neither the primal nor the dual heap of t is necessarily isomorphic to the primal or dual heap of t^R.

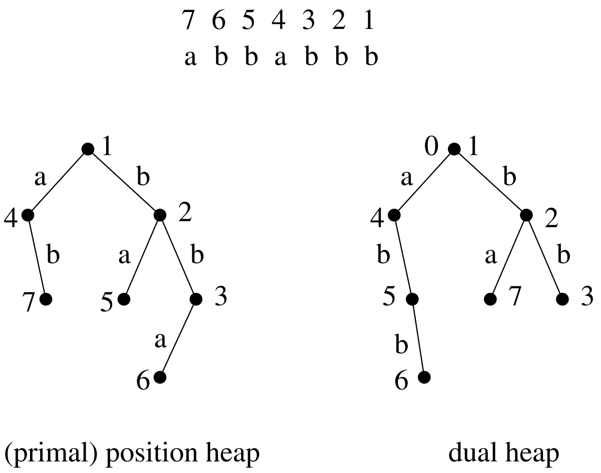


FIGURE 30.12: The position heap and the dual heap of the string *abbabbb*. The node set of both heaps is the same, but the label of the path leading to a node in the dual heap is the reverse of the label of the path leading to it in the position heap.

For `PHBuild`, the bottleneck is finding the node u whose path label is $t' = t_i t_{i+1} ... t_k$. The dual heap allows us to carry out this step more efficiently. We get an $O(|t|)$ time bound for constructing the position heap by simultaneously constructing the position heap and its dual. It is also necessary to label each node v with its depth d_v during construction, in addition to its position label, p_v. This gives a compact representation of the path label of v if v is not the root: it is $t[p_v : p_v - d_v + 1]$.

During construction, the primal edges are directed from child to parent, while the dual edges are directed from parent to child. The modified procedure, `FastPHBuild`, is given in Table 30.3.

LEMMA 30.12 `FastPHBuild` is correct.

Proof The path label of v is $t[i-1 : i-1-d_v+1] = t[i-1 : i-d_v]$. Let $d = d_w$ be the depth of w. Since w is an ancestor of v, its path label is a prefix of this, so w's path label is $t[i-1 : i-d]$. Since v' is the parent of w, the path label of v' is the next shorter prefix, $t[i-1 : i-d+1]$. The path label of v' in the dual is the reversal of this, and since u is reachable on the dual edge out of v' that is labeled t_i, the path label of u is the reversal of $t[i : i-d+1]$ in the dual, hence $t[i : i-d+1]$ in the primal heap. Since w has no child labeled t_i in the dual, there is no node whose path label in the dual is the reversal of $t[i : i-d]$, hence no node whose path label is $t[i : i-d]$ in the primal heap.

Therefore, u has path label $t[i : i-d+1]$ and has no child in the primal graph on t_{i-d}.

TABLE 30.3 Construct the position heap H and its dual D for a string $t[i:1]$. Return (H, D, y), where y is a pointer to the node with position label i.

```
FastPHBuild $(T, i)$
    If $i = 1$, return a single tree node labeled 1
    Let $(H', D', v) =$ FastPHBuild$(t, i-1)$
    Search upward from $v$ in $H'$ to find the lowest ancestor $v'$ of $v$ that has
        a child $u$ on edge labeled $t_i$ in the dual.
    Let $w$ be the penultimate node on this path.
    Let $d = d_w$ be the depth of $w$ in the heap
    Create a new child $y$ of $u$ in the position heap on edge labeled $t_{i-d}$
    Make $y$ be the child of $w$ in the dual on edge labeled $t_i$.
    Give $y$ position label $i$.
    Give $y$ depth label $d_y = d + 1$
    Return the modified position heap, the modified dual, and $y$.
```

It follows that updating the primal heap to reflect $t[i:1]$ requires adding a new child y labeled t_{i-d_2} to u in the primal heap. Since w's path label is the longest proper suffix of y's path label, w must be the parent of y in the dual. Since its depth is one greater than w's, $d_y = d + 1$. ♢

LEMMA 30.13 FastPHBuild takes $O(|t|)$ time.

Proof The inductive step takes $O(1)$ time, except for the cost of searching upward from v to find v'. Let k be the distance from v' to v and let $k' = k - 1$. The cost of searching upward is $O(k)$. The depth of the new node y is $d_{v'} + 2$, so it is $d_v - k + 2 \leq d_v + 1$. Since v is the node added just before y, the depth of each successive node added increases by at most one and decreases by $\Theta(k)$. The total increases are $O(|t|)$, so the sum of k's over all recursive calls is bounded by this, hence also $O(|t|)$. ♢

On tests we have run on several-megabyte texts, FastPHBuild is noticeably faster than PHBuild. This advantage must be weighed against the fact that the algorithm is slightly more difficult to understand, and uses more memory during construction, to store the dual edges.

By contrast, the algorithm we describe next for finding the positions of p in t in $O(|p|+k)$ time is unlikely to compete in practice with PHFind, since the worst case bound of $O(|p|^2 + k)$ for PHFind overstates the typical case. However, it is interesting from a theoretical standpoint.

Let # be a character that is not in Σ. Let $t\#t$ denote the concatenation of two copies of t with the special character # in between. To obtain the time bound for PHFind, we may build the position heap of $t\#t$ in $O(|t|)$ time using FastPHBuild. Index the positions from $|t|$ to $-|t|$ in descending order. This gives 0 as the position of the # character (see Figure 30.13).

To find the starting positions of p in t, it suffices to find only its positive starting positions in $t\#t$. Suppose that there is a path labeled p that has at most one node with a positive position number. Finding the last node v of the path takes $O(|p|)$ time, and all k positive

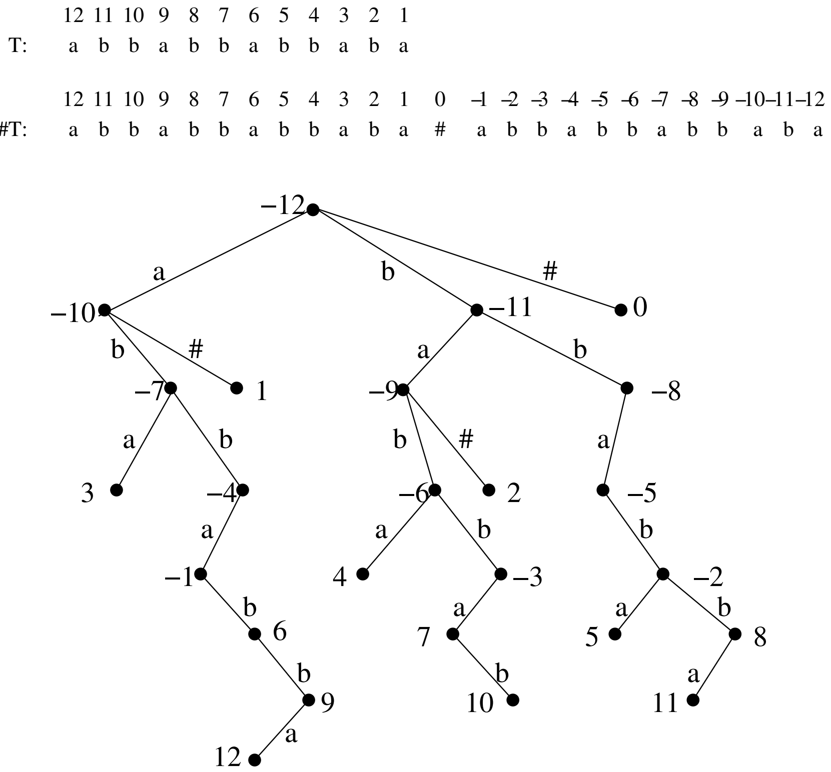


FIGURE 30.13: Finding occurrences of p in t in $O(|p| + k)$ time, using a position heap. Because of the extra memory requirements and the good expected performance of the $O(|p|^2 + k)$ approach, the algorithm is of theoretical interest only. The trick is to build the position heap of $t\#t$, indexing so that positions in the second occurrence are indexed with negative numbers. To find the occurrences of p in t, it suffices to return only its positive positions in $t\#t$. Indexing into the heap is organized so that positive positions are descendants of nodes that are indexed to. Negative occurrences, which are ancestors, do not need to be verified against the text, eliminating the $\Theta(|p|^2)$ step of the simpler algorithm.

starting positions are descendants. We can retrieve them in $O(k)$ time. Since we are not required to find negative position numbers where p occurs, we do not have the $\Theta(|p|^2)$ cost of finding which ancestors of v are actual matches. This gives an $O(|p| + k)$ bound in this case.

Otherwise, the problem can be solved by chopping p into segments $\{x_1, x_2, ..., x_k\}$ such that each x_i is the label of a path from the root in the heap that has exactly one node v_i with a positive position number, namely, the last node of the path. Every positive position of x_i is matched by a negative position, which must correspond to an ancestor of v_i. Since there are at most $|x_i|$ ancestors of v_i, v_i has at most $|x_i|$ (positive) descendants, which can be retrieved in $O(|x_i|)$ time.

To see that this implies an $O(|p|)$ time bound to return all occurrences of p in t, the reader should first note that a family $\mathcal{F}$ of k sets $X_1, X_2, ..., X_k$ of integers are represented with sorted lists, it takes $O(|X_1| + |X_2| + ...|X_k|)$ time to find their intersection. The key to this insight is that when two sets in $\mathcal{F}$ are merged, replacing them with their intersection, the sum of cardinalities of sets in $\mathcal{F}$ drops by an amount proportional to the time to perform the intersection. Therefore, the bound for all intersections is proportional to the sum of cardinalities of the initial lists. The problem of finding the occurrences of p reduces to this one as follows. Let X_i denote the positive positions of segment x_i of p. Shift these positions to the left by $|x_1x_2...x_{i-1}|$ to find the candidate positions they imply for the left endpoint of p in t. Intersecting the sets of candidates gives the locations of p in t.

To find the substrings $\{x_1, x_2, ..., x_k\}$ of p, index from the root of the position heap on the leading characters of p until a positive node is reached. The label of this path is x_1, and recursing on the remaining suffix of p gives $\{x_2, x_3, ..., x_{k-1}\}$. It doesn't give x_k, since an attempt to produce x_k in this way it may run out of characters of p before a node with a positive position number is reached. Instead, find x_k by indexing from the right end of p using the dual heap until a positive position number is reached. Therefore, $\{x_1, x_2, ..., x_{k-1}\}$ represent disjoint intervals p, while x_{k-1} and x_k can represent overlapping intervals of p. The sum of their lengths is therefore $O(|p|)$, giving an $O(|p|)$ bound to find all occurrences of p in t in this case.

References

[1] A. Blumer, J. Blumer, D. Ehrenfeucht, D. Haussler, and R McConnell. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34:578–595, 1987.

[2] B. Clift, D. Haussler, R. McConnell, T. D. Schneider, and G. D. Stormo. Sequence landscapes. *Nucleic Acids Research*, 14:141–158, 1986.

[3] E. G. Coffman and J. Eve. File structures using hashing functions. *Communications of the ACM*, 11:13–21, 1981.

[4] U. Manber and E. Myers. Suffix arrays: a new method for on-line search. *SIAM J. Comput.*, 22:935–948, 1993.

[5] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23:262–272, 1976.

[6] D. P. Mehta and S. Sahni. A data structure for circular string analysis and visualization. *IEEE Transactions on Computers*, 42:992–997, 1993.

[7] D. P. Mehta and S. Sahni. Computing display conflicts in string visualization. *IEEE Transactions on Computers*, 43:350–361, 1994.

[8] D. P. Mehta and S. Sahni. Models, techniques, and algorithms for finding, selecting and displaying patterns in strings and other discrete objects. *Journal of Systems and Software*, 39:201–221, 1997.

31

Persistent Data Structures

Haim Kaplan
Tel Aviv University

31.1 Introduction

Think of the initial configuration of a data structure as version zero, and of every subsequent update operation as generating a new version of the data structure. Then a data structure is called *persistent* if it supports access to all versions and it is called *ephemeral* otherwise. The data structure is *partially persistent* if all versions can be accessed but only the newest version can be modified. The structure is *fully persistent* if every version can be both accessed and modified. The data structure is *confluently persistent* if it is fully persistent and has an update operation which combines more than one version. Let the version graph be a directed graph where each node corresponds to a version and there is an edge from node V_1 to a node V_2 if and only of V_2 was created by an update operation to V_1. For partially persistent data structure the version graph is a path; for fully persistent data structure the version graph is a tree; and for confluently persistent data structure the version graph is a directed acyclic graph (DAG).

A notion related to persistence is that of purely functional data structures. (See Chapter 40 by Okasaki in this handbook.) A purely functional data structure is a data structure that can be implemented without using an assignment operation at all (say using just the functions CAR, CDR, and CONS, of pure lisp). Such a data structure is automatically persistent. The converse, however, is not true. There are data structures which are persistent and perform assignments.

Since the seminal paper of Driscoll, Sarnak, Sleator, and Tarjan (DSST) [18], and over the past fifteen years, there has been considerable development of *persistent* data structures. Persistent data structures have important applications in various areas such as functional programming, computational geometry and other algorithmic application areas.

The research on persistent data structures splits into two main tracks. The first track is of designing general transformations that would make any ephemeral data structure persistent while introducing low overhead in space and time. The second track is on how to make specific data structures, such as lists and search trees, persistent. The seminal work of DSST mainly addresses the question of finding a general transformation to make any data structure persistent. In addition DSST also address the special case of making search trees persistent in particular. For search trees they obtain a result which is better than what one gets by simply applying their general transformation to, say, red-black trees.

There is a *naive scheme* to make any data structure persistent. This scheme performs the operations exactly as they would have been performed in an ephemeral setting but before each update operation it makes new copies of all input versions. Then it performs the update on the new copies. This scheme is obviously inefficient as it takes time and space which is at least linear in the size of the input versions.

When designing an efficient general transformation to make a data structure persistent DSST get started with the so called *fat node method*. In this method you allow each field in the data structure to store more than one value, and you tag each value by the version which assigned it to the field. This method is easy to apply when we are interested only in a partially persistent data structure. But when the target is a fully persistent data structure, the lack of linear order on the versions already makes navigation in a naive implementation of the fat node data structure inefficient. DSST manage to limit the overhead by linearizing the version tree using a data structure of Dietz and Sleator so we can determine fast whether one version precedes another in this linear order.

Even when implemented carefully the fat node method has logarithmic (in the number of versions) time overhead to access or modify a field of a particular node in a particular version. To reduce this overhead DSST described two other methods to make data structures persistent. The simpler one is the *node copying method* which is good to obtain partially persistent data structures. For obtaining fully persistent data structures they suggest the *node splitting method.* These methods simulate the fat node method using nodes of constant size. They show that if nodes are large enough (but still of constant size) then the amount of overhead is constant per access or update of a field in the ephemeral data structure.

These general techniques suggested by DSST have some limitations. First, all these methods, including even the fat node method, fail to work when the data structure has an update operation which combines more than one version, and confluent persistence is desired. Furthermore, the node splitting and node copying methods apply only to pointer based data structures (no arrays) where each node is of constant size. Since the simulation has to add reverse pointers to the data structure the methods require nodes to be of bounded indegree as well. Last, the node coping and the node splitting techniques have $O(1)$ amortized overhead per update or access of a field in the ephemeral data structure. DSST left open the question of how to make this overhead $O(1)$ in the worst case.

These limitations of the transformations of DSST were addressed by subsequent work. Dietz and Raman [13] and Brodal [5] addressed the question of bounding the worst case overhead of an access or an update of a field. For partial persistence Brodal gives a way to implement node coping such that the overhead is $O(1)$ in the worst case. For fully persistence, the question of whether there is a transformation with $O(1)$ worst case overhead is still unresolved.

The question of making data structures that use arrays persistent with less than logarithmic overhead per step has been addressed by Dietz [12]. Dietz shows how to augment the fat node method with a data structure of van Emde Boaz, Kaas, and Zijlstra [33, 34] to make an efficient fully persistent implementation of an array. With this implementation, if we denote by m the number of updates, then each access takes $O(\log\log m)$ time, an update takes $O(\log\log m)$ expected amortized time and the space is linear in m. Since we can model the memory of a RAM by an array, this transformation of Dietz can make any data structure persistent with slowdown double logarithmic in the number of updates to memory.

The question of how to make a data structure with an operation that combines versions confluently persistent has been recently addressed by Fiat and Kaplan [19]. Fiat and Kaplan point out the fundamental difference between fully persistent and confluently persistent data structures. Consider the naive scheme described above and assume that each update operation creates constantly many new nodes. Then, as long as no update operation combines more than one version, the size of any version created by the naive scheme is linear in the number of versions. However when updates combine versions the size of a single version can be exponential in the number of versions. This happens in the simple case where we update a linked list by concatenating it to itself n times. If the initial list is of size one then the final list after n concatenations is of size 2^n.

Fiat and Kaplan prove by simple information theoretic argument that for any general reduction to make a data structure confluently persistent there is a DAG of versions which cannot be represented using only constant space per assignment. Specifically, Fiat and Kaplan define the *effective depth of the DAG* which is the logarithm of the maximum number of different paths from the root of the DAG to any particular vertex. They show that the number of bits that may be required for assignment is at least as large as the *effective depth of the DAG*. Fiat and Kaplan also give several methods to make a data structure confluently persistent. The simplest method has time and space overhead proportional to the depth of the DAG. Another method has overhead proportional to the effective depth of the DAG and degenerate to the fat node method when the DAG is a tree. The last method reduces the time overhead to be polylogarithmic in either the depth of the DAG or the effective depth of the DAG at the cost of using randomization and somewhat more space.

The work on making specific data structures persistent has started even prior to the work of DSST. Dobkin and Munro [16] considered a persistent data structure for computing the rank of an object in an ordered set of elements subject to insertions and deletions. Overmars [29] improved the time bounds of Dobkin and Munro and further reduced the storage for the case where we just want to determine whether an element is in the current set or not. Chazelle [8] considered finding the predecessor of a new element in the set. As we already mentioned DSST suggest two different ways to make search trees persistent. The more efficient of their methods has $O(\log n)$ worst case time bound and $O(1)$ worst case space bound for an update.

A considerable amount of work has been devoted to the question of how to make concatenable double ended queues (deques) confluently persistent. Without catenation, one can make deques fully persistent either by the general techniques of DSST or via real-time simulation of the deque using stacks (see [23] and the references there). Once catenation is added, the problem of making stacks or deques persistent becomes much harder, and the methods mentioned above fail. A straightforward use of balanced trees gives a representation of persistent catenable deques in which an operation on a deque or deques of total size n takes $O(\log n)$ time. Driscoll, Sleator, and Tarjan [17] combined a tree representation with several additional ideas to obtain an implementation of persistent catenable stacks in which the k^{th} operation takes $O(\log \log k)$ time. Buchsbaum and Tarjan [7] used a recursive decomposition of trees to obtain two implementations of persistent catenable deques. The first has a time bound of $2^{O(\log^* k)}$ and the second a time bound of $O(\log^* k)$ for the k^{th} operation, where $\log^* k$ is the iterated logarithm, defined by $\log^{(1)} k = \log_2 k, \log^{(i)} k = \log \log^{(i-1)} k$ for $i > 1$, and $\log^* k = \min\{i \mid \log^{(i)} k \leq 1\}$.

Finally, Kaplan and Tarjan [23] gave a real-time, purely functional (and hence confluently persistent) implementation of deques with catenation in which each operation takes $O(1)$

time in the worst case. A related structure which is simpler but not purely functional and has only amortized constant time bound on each operation has been given by Kaplan, Okasaki, and Tarjan [21]. A key ingredient in the results of Kaplan and Tarjan and the result of Kaplan, Okasaki, and Tarjan is an algorithmic technique related to the redundant digital representations devised to avoid carry propagation in binary counting [9]. If removing elements from one side of the deque is disallowed. Okasaki [28] suggested another confluently persistent implementation with $O(1)$ time bound for every operation. This technique is related to path reversal technique which is used in some union-find data structures [32].

Search trees also support catenation and split operations [31] and therefore confluently persistent implementation of search trees is natural to ask for. Search trees can be made persistent and even confluently persistent using the path copying technique [18]. In path copying you copy every node that changes while updating the search tree and its ancestors. Since updates to search trees affect only a single path, this technique results in copying at most one path and thereby costs logarithmic time and space per update. Making finger search trees confluently persistent is more of a challenge, as we want to prevent the update operation to propagate up on the leftmost and rightmost spines of the tree. This allows an update to be made at distance d from the beginning or end of the list in $O(\log d)$ time. Kaplan and Tarjan [22] used the redundant counting technique to make finger search tree confluently persistent. Using the same technique they also managed to reduce the time (and space) overhead of catenation to be $O(\log \log n)$ where n is the number of elements in the larger tree.

The structure of the rest of this chapter is as follows. Section 31.2 describes few algorithms that use persistent data structures to achieve their best time or space bounds. Section 31.3 surveys the general methods to make data structures persistent. Section 31.4 gives the highlights underlying persistent concatenable deques. We conclude in Section 31.5.

31.2 Algorithmic Applications of Persistent Data Structures

The basic concept of persistence is general and may arise in any context where one maintains a record of history for backup and recovery, or for any other purpose. However, the most remarkable consequences of persistent data structures are specific algorithms that achieve their best time or space complexities by using a persistent data structure. Most such algorithms solve geometric problems but there are also examples from other fields. In this section we describe few of these algorithms.

The most famous geometric application is the algorithm for planar point location by Sarnak and Tarjan [30] that triggered the development of the whole area. In the planar point location problem we are given a subdivision of the Euclidean plane into polygons by n line segments that intersect only at their endpoints. The goal is to preprocess these line segments and build a data structure such that given a query point we can efficiently determine which polygon contains it. As common in this kind of computational geometry problem, we measure a solution by three parameters: The space occupied by the data structure, the preprocessing time, which is the time it takes to build the data structure, and the query time.

Sarnak and Tarjan suggested the following solution (which builds upon previous ideas of Dobkin and Lipton [15] and Cole [10]). We partition the plane into vertical slabs by drawing a vertical line through each vertex (intersection of line segments) in the planar subdivision. Notice that the line segments of the subdivision intersecting a slab are totally ordered. Now it is possible to answer a query by two binary searches. One binary search locates the slab that contains the query, and another binary search locates the segment preceding the query point within the slab. If we associate with each segment within a slab, the polygon just above it, then we have located the answer to the query. If we represent the slabs by a binary search tree from left to right, and the segments within each slab by a binary search tree sorted from bottom to top, we can answer a query in $O(\log n)$ time.* However if we build a separate search tree for each slab then the worst case space requirement is $\Omega(n^2)$, when $\Omega(n)$ lines intersect $\Omega(n)$ slabs.

The key observation is that the sets of line segments intersecting adjacent slabs are similar. If we have the set of one particular slab we can obtain the set of the slab to its right by deleting segments that end at the boundary between these slabs, and inserting segments that start at that boundary. As we sweep all the slabs from left to right we get that in total there are n deletions and n insertions; one deletion and one insertion for every line segment. This observation reduces the planar point location to the problem of maintaining partially persistent search trees. Sarnak and Tarjan [30] suggested a simple implementation of partially persistent search tree where each update takes $O(\log n)$ amortized time and consumes $O(1)$ amortized space. Using these search trees they obtained a data structure for planar point location that requires $O(n)$ space, takes $O(n \log n)$ time to build, and can answer each query in $O(\log n)$ time.

The algorithm of Sarnak and Tarjan for planar point location in fact suggests a general technique for transforming a 2-dimensional geometric search problem into a persistent data structure problem. Indeed several applications of this technique have emerged since Sarnak and Tarjan published their work [3]. As another example consider the problem of 3-sided range searching in the plane. In this problem we preprocess a set of n points in the plane so given a triple (a, b, c) with $a \le b$ we can efficiently reports all points $(x, y) \in S$ such that $a \le x \le b$, and $y \le c$. The priority search tree of McCreight [26] yields a solution to this problem with $O(n)$ space, $O(n \log n)$ preprocessing time, and $O(\log n)$ time per query. Using persistent data structure, Boroujerdi and Moret [3] suggest the following alternative. Let $y_1 \le y_2 \le \cdots \le y_n$ be the y-coordinates of the points in S in sorted order. For each i, $1 \le i \le n$ we build a search tree containing all i points $(x, y) \in S$ where $y \le y_i$, and associate that tree with y_i. Given this collection of search tree we can answer a query (a, b, c) in $O(\log n)$ time by two binary searches. One search uses the y coordinate of the query point to find the largest i such that $y_i \le c$. Then we use the search tree associated with y_i to find all points (x, y) in it with $a \le x \le b$. If we use partially persistent search trees then we can build the trees using n insertions so the space requirement is $O(n)$, and the preprocessing time is $O(n \log n)$.

This technique of transforming a 2-dimensional geometric search problem into a persistent data structure problem requires only a partially persistent data structure. This is since we only need to modify the last version while doing the sweep. Applications of fully persistent data structures are less common. However, few interesting ones do exist.

One such algorithm that uses a fully persistent data structure is the algorithm of Alstrup et al. for the binary dispatching problem [1]. In object oriented languages there is a hierarchy of classes (types) and method names are overloaded (i.e., a method may have different implementations for different types of its arguments). At run time when a method is invoked, the most specific implementation which is appropriate for the arguments has to be activated. This is a critical component of execution performance in object oriented languages. Here is a more formal specification of the problem.

We model the class hierarchy by a tree T with n nodes, each representing a class. A

*Note that testing whether a point is above or below a line takes $O(1)$ time.

class A which is a descendant of B is *more specific* than B and we denote this relation by $A \leq B$ or $A < B$ if we know that $A \neq B$. In addition we have m different implementations of methods, where each such implementation is specified by a name, number of arguments, and the type of each argument. We shall assume that $m > n$, as if that is not the case we can map nodes that do not participate in any method to their closest ancestor that does participate in $O(n)$ time. A method invocation is a query of the form $s(A_1, \ldots, A_d)$ where s is a method name that has d arguments with types $A_1, \ldots, A_d$, respectively. An implementation $s(B_1, \ldots, B_d)$ is applicable for $s(A_1, \ldots, A_d)$ if $A_i \leq B_i$ for every $1 \leq i \leq d$. The *most specific* method which is applicable for $s(A_1, \ldots, A_d)$ is the method $s(B_1, \ldots, B_d)$ such that $A_i \leq B_i$ for $1 \leq i \leq d$, and for any other implementation $s(C_1, \ldots, C_d)$ which is applicable for $s(A_1, \ldots, A_d)$ we have $B_i \leq C_i$ for $1 \leq i \leq d$. Note that for $d > 1$ this may be ambiguous, i.e. we might have two applicable methods $s(B_1, \ldots, B_d)$ and $s(C_1, \ldots, C_d)$ where $B_i \neq C_i$, $B_j \neq C_j$, $B_i \leq C_i$ and $C_j \leq B_j$. The dispatching problem is to find for each invocation the most specific applicable method if it exists. If it does not exist or in case of ambiguity, "no applicable method" or "ambiguity" has to be reported, respectively. In the binary dispatching problem, $d = 2$, i.e. we assume that all implementations and invocations have two arguments.

Alstrup et al. describe a data structure for the binary dispatching problem that use $O(m)$ space, $O(m(\log \log m)^2)$ preprocessing time and $O(\log m)$ query time. They obtain this data structure by reducing the problem to what they call the *bridge color problem.* In the bridge color problem the input consists of two trees T_1 and T_2 with edges, called bridges, connecting vertices in T_1 to vertices in T_2. Each bridge is colored by a subset of colors from C. The goal is to construct a data structure which allows queries of the following form. Given a triple (v_1, v_2, c) where $v_1 \in T_1$, $v_2 \in T_2$, and $c \in C$ finds the bridge (w_1, w_2) such that

1. $v_1 \leq w_1$ in T_1, and $v_2 \leq w_2$ in T_2, and c is one of the colors associated with (w_1, w_2).
2. There is no other such bridge (w', w'') with $v_2 \leq w'' < w_2$ or $v_1 \leq w' < w_1$.

If there is no bridge satisfying the first condition the query just returns nothing and if there is a bridge satisfying the first condition but not the second we report "ambiguity". We reduce the binary dispatching problem to the bridge color problem by taking T_1 and T_2 to be copies of the class hierarchy T of the dispatching problem. The set of colors is the set of different method names. (Recall that each method name may have many implementations for different pairs of types.) We make a bridge (v_1, v_2) between $v_1 \in T_1$ and $v_2 \in T_2$ whenever there is an implementation of some method for classes v_1 and v_2. We color the bridge by all names of methods for which there is an implementation specific to the pair of type (v_1, v_2). It is easy to see now that when we invoke a method $s(A_1, A_2)$ the most specific implementation of s to activate corresponds to the bridge colored s connecting an ancestor of v_1 to an ancestor of v_2 which also satisfies Condition (2) above.

In a way which is somewhat similar to the reduction between static two dimensional problem to a dynamic one dimensional problem in the plane sweep technique above, Alstrup et al. reduce the static bridge color problem to a similar dynamic problem on a single tree which they call the *tree color problem.* In the tree color problem you are given a tree T, and a set of colors C. At any time each vertex of T has a set of colors associated with it. We want a data structure which supports the updates, color(v,c): which add the color c to the set associated with v; and uncolor(v,c) which deletes the color c from the set associated with v. The query we support is given a vertex v and a color c, find the closest ancestor of v that has color c.

The reduction between the bridge color problem and the tree color problem is as follows. For each node $v \in T_1$ we associate an instance ℓ_v of the tree color problem where the underlying tree is T_2 and the set of colors C is the same as for the bridge color problem. The label of a node $w \in T_2$ in ℓ_v contains color c if w is an endpoint of a bridge with color c whose endpoint in T_1 is an ancestor of v. For each pair (w, c) where $w \in T_2$ and c is a color associated with w in ℓ_v we also keep the closest ancestor v' to v in T_1 such that there is a bridge (v', w) colored c. We can use a large (sparse) array indexed by pairs (w, c) to map each such pair to its associated vertex. We denote this additional data structure associated with v by a_v. Similarly for each vertex $u \in T_2$ we define an instance ℓ_u of the tree color problem when the underlying tree is T_1, and the associated array a_u.

We can answer a query (v_1, v_2, c) to the bridge color data structure as follows. We query the data structure ℓ_{v_1} with v_2 to see if there is an ancestor of v_2 colored c in the coloring of T_2 defined by ℓ_{v_1}. If so we use the array a_{v_1} to find the bridge (w_1, w_2) colored c where $v_1 \leq w_1$ and $v_2 \leq w_2$, and w_1 is as close as possible to v_1. Similarly we use the data structures ℓ_{v_2} and a_{v_2} to find the bridge (w_1, w_2) colored c where $v_1 \leq w_1$ and $v_2 \leq w_2$, and w_2 is as close as possible to v_2, if it exists. Finally if both bridges are identical then we have the answer to the query (v_1, v_2, c) to the bridge color data structure. Otherwise, either there is no such bridge or there is an ambiguity (when the two bridges are different).

The problem of this reduction is its large space requirement if we represent each data structure ℓ_v, and a_v for $v \in T_1 \cup T_2$ independently.[†] The crucial observation though is that these data structures are strongly related. Thus if we use a dynamic data structure for the tree color problem we can obtain the data structure corresponding to w from the data structure corresponding to its parent using a small number of modifications. Specifically, suppose we have generated the data structures ℓ_v and a_v for some $v \in T_1$. Let w be a child of v in T_1. We can construct ℓ_w by traversing all bridges whose one endpoint is w. For each such bridge (w, u) colored c, we perform color(u,c), and update the entry of (u, c) in a_v to contain w.

So if we were using fully persistent arrays and a fully persistent data structure for the tree color problem we can construct all data structures mentioned above while doing only $O(m)$ updates to these persistent data structures. Alstrup et al. [1] describe a data structure for the tree color problem where each update takes $O(\log\log m)$ expected time and query time is $O(\log m/\log\log m)$. The space is linear in the sum of the sizes of the color-sets of the vertices. To make it persistent without consuming too much space Alstrup et al. [1] suggest how to modify the data structure so that each update makes $O(1)$ memory modifications in the worst case (while using somewhat more space). Then by applying the technique of Dietz [12] (see also Section 31.3.3) to this data structure we can make it fully persistent. The time bounds for updates and queries increase by a factor of $O(\log\log m)$, and the total space is $O(|C|m)$. Similarly, we can make the associated arrays a_v fully persistent. The resulting solution to the binary dispatching problem takes $O(m(\log\log m)^2)$ time to construct, requires $O(|C|m)$ space and support a query in $O(\log m)$ time. Since the number of memory modifications while constructing the data structure is only $O(m)$ Alstrup et al. also suggest that the space can be further reduces to $O(m)$ by maintaining the entire memory as a dynamic perfect hashing data structure.

Fully persistent lists proved useful in reducing the space requirements of few three dimensional geometric algorithms based on the sweep line technique, where the items on the sweep line have secondary lists associated with them. Kitsios and Tsakalidis [25] considered hidden line elimination and hidden surface removal. The input is a collection of (non

[†] We can compress the sparse arrays using hashing but even if we do that the space requirement may be quadratic in m.

intersecting) polygons in three dimensions. The hidden line problem asks for the parts of the edges of the polygons that are visible from a given viewing position. The hidden surface removal problem asks to compute the parts of the polygons that are visible from the viewing position.

An algorithm of Nurmi [27] solves these problems by projecting all polygons into a collection of possible intersecting polygons in the plane and then sweeping this plane, stopping at any vertex of a projected polygon, or crossing point of a pair of projected edges. When the sweep stops at such point, the visibility status of its incident edges is determined. The algorithm maintain a binary balanced tree which stores the edges cut by the sweep line in sorted order along the sweep line. With each such edge it also maintains another balanced binary tree over the faces that cover the interval between the edge and its successor edge on the sweep line. These faces are ordered in increasing depth order along the line of sight. An active edge is visible if the topmost face in its list is different from the topmost face in the list of its predecessor. If n is the number of vertices of the input polygons and I is the number of intersections of edges on the projection plane then the sweep line stops at $n + I$ points. Looking more carefully at the updates one has to perform during the sweep, we observe that a constant number of update operations on balanced binary search trees has to be performed non destructively at each point. Thus, using fully persistent balanced search trees one can implement the algorithm in $O((n + I) \log n)$ time and $O(n + I)$ space. Kitsios and Tsakalidis also show that by rebuilding the data structure from scratch every $O(n)$ updates we can reduce the space requirement to $O(n)$ while retaining the same asymptotic running time.

Similar technique has been used by Bozanis et al. [4] to reduce the space requirement of an algorithm of Gupta et al. [20] for the rectangular enclosure reporting problem. In this problem the input is a set S of n rectangles in the plane whose sides are parallel to the axes. The algorithm has to report all pairs (R, R') of rectangles where $R, R' \in S$ and R encloses R'. The algorithm uses the equivalence between the rectangle enclosure reporting problem and the 4-dimensional dominance problem. In the 4-dimensional dominance problem the input is a set of n points P in four dimensional space. A point $p = (p_1, p_2, p_3, p_4)$ dominates $p' = (p'_1, p'_2, p'_3, p'_4)$ if and only if $p_i \geq p'_i$ for $i = 1, 2, 3, 4$. We ask for an algorithm to report all *dominating pairs* of points, (p, p'), where $p, p' \in P$, and p dominates p'. The algorithm of Gupta et al. first sorts the points by all coordinates and translates the coordinates to ranks so that they become points in U^4 where $U = \{0, 1, 2, \ldots, n\}$. It then divides the sets into two equal halves R and B according to the forth coordinate (R contains the points with smaller forth coordinate). Using recurrence on B and on R it finds all dominating pairs (p, p') where p and p' are either both in B or both in R. Finally it finds all dominating pairs (r, b) where $r \in R$ and $b \in B$ by iterating a plane sweeping algorithm on the three dimensional projections of the points in R and B. During the sweep, for each point in B, a list of points that it dominates in R is maintained. The size of these lists may potentially be as large as the output size which in turn may be quadratic. Bozanis et al. suggest to reduce the space by making these lists fully persistent, which are periodically being rebuilt.

31.3 General Techniques for Making Data Structures Persistent

We start in Section 31.3.1 describing the fat node simulation. This simulation allows us to obtain fully persistent data structures and has an optimal space expansion but time slowdown logarithmic in the number of versions. Section 31.3.2 describes the node copying and the node splitting methods that reduce the time slowdown to be constant while increasing the space expansion only by a constant factor. In Section 31.3.3 we address the question of making arrays persistent. Finally in Section 31.3.4 we describe simulation that makes data structures confluently persistent.

31.3.1 The Fat Node Method

DSST first considered the *fat node method*. The fat node method works by allowing a field in a node of the data structure to contain a list of values. In a partial persistent setting we associate field value x with version number i, if x was assigned to the field in the update operation that created version i.[‡] We keep this list of values sorted by increasing version number in a search tree. In this method simulating an assignment takes $O(1)$ space, and $O(1)$ time if we maintain a pointer to the end of the list. An access step takes $O(\log m)$ time where m is the number of versions.

The difficulty with making the fat node method work in a fully persistent setting is the lack of total order on the versions. To eliminate this difficulty, DSST impose a total order on the versions consistent with the partial order defined by the version tree. They call this total order the *version list*. When a version i is created it is inserted into the version list immediately after its parent (in the version tree). This implies that the version list defines a preorder on the version tree where for any version i, the descendants of i in the version tree occur consecutively in the version list, starting with i.

The version list is maintained in a data structure that given two versions x and y allows to determine efficiently whether x precedes y. Such a data structure has been suggested by Dietz and Sleator [11]. (See also a simpler related data structure by [2].) The main idea underlying these data structures is to assign an integer label to each version so that these labels monotonically increase as we go along the list. Some difficulty arises since in order to use integers from a polynomial range we occasionally have to relabel some versions. For efficient implementation we need to control the amount of relabeling being done. We denote such a data structure that maintains a linear order subject to the operation $insert(x, y)$ which inserts x after y, and $order(x, y)$ which returns "yes" if x precedes y, an *Order Maintenance* (OM) data structure.

As in the partial persistence case we keep a list of version-value pairs in each field. This list contains a pair for each value assigned to the field in any version. These pairs are ordered according to the total order imposed on the versions as described above. We maintain these lists such that the value corresponding to field f in version i is the value associated with the largest version in the list of f that is not larger than i. We can find this version by carrying out a binary search on the list associated with the field using the OM data structure to do comparisons.

To maintain these lists such that the value corresponding to field f in version i is the value

[‡] If the update operation that created version i assigned to a particular field more than once we keep only the value that was assigned last.

associated with the largest version in the list of f that is not larger than i, the simulation of an update in the fully persistent setting differ slightly from what happens in the partially persistent case. Assume we assign a value x to field f in an update that creates version i. (Assume for simplicity that this is the only assignment to f during this update.) First we add the pair (i, x) to the list of pairs associated with field f. Let i' be the version following i in the version list (i.e. in the total order of all versions) and let i'' be the version following i in the list associated with f. (If there is no version following i in one of these lists we are done.) If $i'' > i'$ then the addition of the pair (i, x) to the list of pairs associated with f may change the value of f in all versions between i' and the version preceding i'' in the version list, to be x. To fix that we add another pair (i', y) to the list associated with f, where y is the value of f before the assignment of x to f. The overhead of the fat node method in a fully persistent settings is $O(\log m)$ time and $O(1)$ space per assignment, and $O(\log m)$ time per access step, where m is the number of versions. Next, DSST suggested two methods to reduce the logarithmic time overhead of the fat node method. The simpler one obtains a partially persistent data structure and is called *node copying*. To obtain a fully persistent data structure DSST suggested the *node splitting* method.

31.3.2 Node Copying and Node Splitting

The node-copying and the node splitting methods simulate the fat node method using nodes of constant size. Here we assume that the data structure is a pointer based data structure where each node contains a constant number of fields. For reasons that will become clear shortly we also assume that the nodes are of constant bounded in-degree, i.e. the number of pointer fields that contains the address of any particular node is bounded by a constant.

In the node copying method we allow nodes in the persistent data structure to hold only a fixed number of field values. When we run out of space in a node, we create a new copy of the node, containing only the newest value of each field. Let d be the number of pointer fields in an ephemeral node and let p be the maximum in-degree of an ephemeral node. Each persistent node contains d fields which corresponds to the fields in the ephemeral node, p *predecessor fields*, e *extra fields*, where e is a sufficiently large constant that we specify later, and one field for a *copy pointer*.

All persistent nodes which correspond to the same ephemeral node are linked together in a single linked list using the copy pointer. Each field in a persistent node has a version stamp associated with it. As we go along the chain of persistent nodes corresponding to one ephemeral node then the version stamps of the fields in one node are no smaller than version stamps of the fields in the preceding nodes. The last persistent node in the chain is called *live*. This is the persistent node representing the ephemeral node in the most recent version which we can still update. In each live node we maintain *predecessor pointers*. If x is a live node and node z points to x then we maintain in x a pointer to z.

We update field f in node v, while simulating the update operation creating version i, as follows.[§] Let x be the *live* persistent node corresponding to v in the data structure. If there is an empty extra field in x then we assign the new value to this extra field, stamp it with version i, and mark it as a value associated with original field f. If f is a pointer field which now points to a node z, we update the corresponding predecessor pointer in z to point to x. In case all extra fields in x are used (and none of them is stamped with version

[§] We assume that each field has only one value in any particular version. When we update a field in version i that already has a value stamped with version i then we overwrite its previous value.

i) we copy x as follows.

We create a new persistent node y, make the copy pointer of x point to y, store in each original field in y the most recent value assigned to it, and stamp these values with version stamp i. In particular, field f in node y stores its new value marked with version i. For each pointer field in y we also update the corresponding predecessor pointer to point to y rather than to x.

Then we have to update each field pointing to x in version $i-1$ to point to y in version i. We follow, in turn, each predecessor pointer in x. Let z be a node pointed to by such a predecessor pointer. We identify the field pointing to x in z and update its value in version i to be y. We also update a predecessor pointer in y to point to z. If the old value of the pointer to x in z is not tagged with version i (in particular this means that z has not been copied) then we try to use an extra field to store the new version-value pair. If there is no free extra pointer in z we copy z as above. Then we update the field that points to x to point to y in the new copy of z. This sequence of node copying may cascade, but since each node is copied at most once, the simulation of the update step must terminate. In version i, y is the *live* node corresponding to v.

A simple analysis shows that if we use at least as many extra fields as predecessor fields at each node (i.e. $e \geq p$) then the amortized number of nodes that are copied due to a single update is constant. Intuitively, each time we copy a node we gain e empty extra fields in the live version that "pay" for the assignments that had to be made to redirect pointers to the new copy.

A similar simulation called the *node splitting method* makes a data structure fully persistent with $O(1)$ amortized overhead in time and space. The details however are somewhat more involved so we only sketch the main ideas. Here, since we need predecessor pointers for any version[¶] it is convenient to think of the predecessor pointers as part of the ephemeral data structure, and to apply the simulation to the so called *augmented* ephemeral data structure.

We represent each fat node by a list of persistent nodes each of constant size, with twice as many extra pointers as original fields in the corresponding node of the augmented ephemeral data structure. The values in the fields of the persistent nodes are ordered by the version list. Thus each persistent node x is associated with an interval of versions in the version lists, called the *valid interval* of x, and it stores all values of its fields that fall within this interval. The first among these values is stored in an original field and the following ones occupy extra fields.

The key idea underlying this simulation is to maintain the pointers in the persistent structure *consistent* such that when we traverse a pointer valid in version i we arrive at a persistent node whose valid interval contains version i. More precisely, a value c of a pointer field must indicate a persistent node whose valid interval contains the valid interval of c.

We simulate an update step to field f, while creating version i from version $p(i)$, as follows. If there is already a persistent node x containing f stamped with version i then we merely change the value of f in x. Otherwise, let x be the persistent node whose valid interval contains version i. Let $i+$ be the version following i in the version list. Assume the node following x does not have version stamp of $i+$. We create two new persistent node x', and x'', and insert them into the list of persistent nodes of x, such that x' follows x, and x'' follows x'. We give node x' version stamp of i and fill all its original fields with their values at version i. The extra fields in x' are left empty. We give x'' version stamp of $i+$. We fill

[¶] So we cannot simply overwrite a value in a predecessor pointer.

the original fields of x'' with their values at version $i+$. We move from the extra fields of x all values with version stamps following $i+$ in the version list to x''. In case the node which follows x in its list has version stamp $i+$ then x'' is not needed.

After this first stage of the update step, values of pointer fields previously indicating x may be inconsistent. The simulation then continues to restore consistency. We locate all nodes containing inconsistent values and insert them into a set S. Then we pull out one node at the time from S and fix its values. To fix a value we may have to replace it with two or more values each valid in a subinterval of the valid interval of the original value. This increases the number of values that has to be stored at the node so we may have to split the node. This splitting may cause more values to become inconsistent. So node splitting and consistency fixing cascades until consistency is completely restored. The analysis is based on the fact that each node splitting produce a node with sufficiently many empty extra fields. For further details see [18].

31.3.3 Handling Arrays

Dietz [12] describes a general technique for making arrays persistent. In his method, it takes $O(\log\log m)$ time to access the array and $O(\log\log m)$ expected amortized time to change the content of an entry, where m is the total number of updates. The space is linear in m. We denote the size of the array by n and assume that $n < m$.

Dietz essentially suggests to think of the array as one big fat node with n fields. The list of versions-values pairs describing the assignments to each entry of the array is represented in a data structure of van Emde Boas et al. [33, 34]. This data structure is made to consume space linear in the number of items using dynamic perfect hashing [14]. Each version is encoded in this data structure by its label in the associated Order Maintenance (OM) data structure. (See Section 31.3.1.)

A problem arises with the solution above since we refer to the labels not solely via order queries on pairs of versions. Therefore when a label of a version changes by the OM data structure the old label has to be deleted from the corresponding van Emde Boaz data structure and the new label has to be inserted instead. We recall that any one of the known OM data structures consists of two levels. The versions are partitioned into sublists of size $O(\log m)$. Each sublist gets a label and each version within a sublist gets a label. The final label of a version is the concatenation of these two labels. Now this data structure supports an insertion in $O(1)$ time. However, this insertion may change the labels of a constant number of sublists and thereby implicitly change the labels of $O(\log m)$ versions. Reinserting all these labels into the van Emde Boaz structures containing them may take $\Omega(\log m \log\log m)$ time

Dietz suggests to solve this problem by bucketizing the van Emde Boaz data structure. Consider a list of versions stored in such a data structure. We split the list into buckets of size $O(\log m)$. We maintain the versions in each bucket in a regular balanced search tree and we maintain the smallest version from each bucket in a van Emde Boaz data structure. This way we need to delete and reinsert a label of a version into the van Emde Boaz data structure only when the minimum label in a bucket gets relabeled.

Although there are only $O(m/\log m)$ elements now in the van Emde Boaz data structures, it could still be the case that we relabel these particular elements too often. This can happen if sublists that get split in the OM data structure contains a particular large number of buckets' minima. To prevent that from happening we modify slightly the OM data structure as follows.

We associate a potential to each version which equals 1 if the version is currently not a minimum in its bucket of its van Emde Boaz data structure and equals $\log\log m$ if it is a minimum in its bucket. Notice that since there are only $O(m/\log m)$ buckets' minima the total potential assigned to all versions throughout the process is $O(m)$. We partition the versions into sublists according to their potentials where the sum of the potentials of the elements in each sublist is $O(\log m)$. We assign labels to the sublists and within each sublists as in the original OM data structure. When we have to split a sublist the work associated with the split, including the required updates on the associated van Emde Boaz data structures, is proportional to the increase in the potential of this sublist since it had last split.

Since we can model the memory of a Random Access Machine (RAM) as a large array. This technique of Dietz is in fact general enough to make any data structure on a RAM persistent with double logarithmic overhead on each access or update to memory.

31.3.4 Making Data Structures Confluently Persistent

Finding a general simulation to make a pointer based data structure confluently persistent is a considerably harder task. In a fully persistent setting we can construct any version by carrying out a particular sequence of updates ephemerally. This seemingly innocent fact is already problematic in a confluently persistent setting. In a confluently persistent setting when an update applies to two versions, one has to produce these two versions to perform the update. Note that these two versions may originate from the same ancestral version so we need some form of persistence even to produce a single version. In particular, methods that achieve persistence typically create versions that share nodes. Semantically however, when an update applied to versions that share nodes we would like the result to be as if we perform the update on two completely independent copies of the input versions.

In a fully persistent setting if each operation takes time polynomial in the number of versions then the size of each version is also polynomial in the number of versions. This breaks down in a confluently persistent setting where even when each operation takes constant time the size of a single version could be exponential in the number of versions. Recall the example of the linked list mentioned in Section 31.1. It is initialized to contain a single node and then concatenated with itself n time. The size of the last versions is 2^n. It follows that any polynomial simulation of a data structure to make it confluently persistent must in some cases represent versions is a compressed form.

Consider the naive scheme to make a data structure persistent which copies the input versions before each update. This method is polynomial in a fully persistent setting when we know that each update operation allocates a polynomial (in the number of versions) number of new nodes. This is not true in a confluently persistent setting as the linked list example given above shows. Thus there is no easy polynomial method to obtain confluently persistence at all.

What precisely causes this difficulty in obtaining a confluently persistent simulation ? Lets assume first a fully persistent setting and the naive scheme mentioned above. Consider a single node x created during the update that constructed version v. Node x exists in version v and copies of it may also exist in descendant versions of v. Notice however that each version derived from v contains only a *single* node which is either x or a copy of it. In contrast if we are in a confluently persistent setting a descendant version of v may contain more than a single copy of x. For example, consider the linked list being concatenated to itself as described above. Let x be the node allocated when creating the first version. Then after one catenation we obtain a version which contains two copies of x, after 2 catenations we obtain a version containing 4 copies of x, and in version n we have 2^n copies of x.

Now, if we get back to the fat node method, then we can observe that it identifies a node in a specific version using a pointer to a fat node and a version number. This works since in each version there is only one copy of any node, and thus breaks down in the confluently persistent setting. In a confluently persistent setting we need more than a version number and an address of a fat node to identify a particular node in a particular version.

To address this identification problem Fiat and Kaplan [19] used the notion of *pedigree*. To define pedigree we need the following notation. We denote the version DAG by D, and the version corresponding to vertex $v \in D$ by D_v. Consider the naive scheme defined above. Let w be some node in the data structure D_v. We say that node w in version v was *derived from* node y in version u if version u was one of the versions on which the update producing v had been performed, and furthermore node $w \in D_v$ was formed by a (possibly empty) set of assignments to a copy of node $y \in D_u$.

Let w be a node in some version D_u where D_u is produced by the naive scheme. We associate a *pedigree* with w, and denote it by $p(w)$. The pedigree, $p(w)$, is a path $p = \langle v_0, v_1, \ldots, v_k = u\rangle$ in the version DAG such that there exist nodes w_0, w_1, …, w_{k-1}, $w_k = w$, where w_i is a node of D_{v_i}, w_0 was allocated in v_0, and w_i is derived from w_{i-1} for $1 \leq i \leq k$. We also call w_0 the *seminal node* of w, and denote it by $s(w)$. Note that $p(w)$ and $s(w)$ uniquely identify w among all nodes of the naive scheme.

As an example consider Figure 31.1. We see that version v_4 has three nodes (the 1st, 3rd, and 5th nodes of the linked list) with the same seminal node w_0'. The pedigree of the 1st node in D_{v_4} is $\langle v_0, v_1, v_3, v_4\rangle$. The pedigree of the 2nd node in D_{v_4} is also $\langle v_0, v_1, v_3, v_4\rangle$ but its seminal node is w_0. Similarly, we can see that the pedigrees of the 3rd, and the 5th nodes of D_{v_4} are $\langle v_0, v_2, v_3, v_4\rangle$ and $\langle v_0, v_2, v_4\rangle$, respectively.

The basic simulation of Fiat and Kaplan is called the *full path method* and it works as follows. The data structure consists of a collection of *fat nodes*. Each fat node corresponds to an explicit allocation of a node by an update operation or in another words, to some seminal node of the naive scheme. For example, the update operations of Figure 31.1 performs 3 allocations (3 seminal nodes) labeled w_0, w_0', and w_0'', so our data structure will have 3 fat nodes, $f(w_0)$, $f(w_0')$ and $f(w_0'')$. The full path method represents a node w of the naive scheme by a pointer to the fat node representing $s(w)$, together with the pedigree $p(w)$. Thus a single fat node f represents all nodes sharing the same seminal node. We denote this set of nodes by $N(f)$. Note that $N(f)$ may contain nodes that co-exist within the same version and nodes that exist in different versions. A *fat node* contains the same fields as the corresponding seminal node. Each of these fields, however, rather than storing a single value as in the original node stores a dynamic table of field values in the fat node. The simulation will be able to find the correct value in node $w \in N(f)$ using $p(w)$. To specify the representation of a set of values we need the following definition of an *assignment pedigree*.

Let $p = \langle v_0, \ldots, v_k = u\rangle$ be the pedigree of a node $w \in D_u$. Let $w_k = w, w_{k-1}, \ldots, w_1$, $w_i \in D_{v_i}$ be the sequence of nodes such that $w_i \in D_{v_i}$ is derived from $w_{i-1} \in D_{v_{i-1}}$. This sequence exists by the definition of node's pedigree. Let A be a field in w and let j be the maximum such that there has been an assignment to field A in w_j during the update that created v_j. We define the *assignment pedigree of a field* A in node w, denoted by $p(A, w)$, to be the pedigree of w_j, i.e. $p(A, w) = \langle v_0, v_1, \ldots, v_j\rangle$.

In the example of Figure 31.1 the nodes contain one pointer field (named *next*) and one data field (named x). The assignment pedigree of x in the 1st node of D_{v_4} is simply $\langle v_0\rangle$, the assignment pedigree of x in the 2nd node of D_{v_4} is likewise $\langle v_0\rangle$, the assignment pedigree of x in the 3rd node of D_{v_4} is $\langle v_0, v_2, v_3\rangle$. Pointer fields also have assignment pedigrees. The assignment pedigree of the pointer field in the 1st node of D_{v_4} is $\langle v_0, v_1\rangle$, the assignment pedigree of the pointer field in the 2nd node of D_{v_4} is $\langle v_0, v_1, v_3\rangle$, the assignment pedigree of the pointer field of the 3rd node of D_{v_4} is $\langle v_0, v_2\rangle$, finally, the assignment pedigree of the pointer field of the 4th node of D_{v_4} is $\langle v_2, v_3, v_4\rangle$.

We call the set $\{p(A, w) \mid w \in N(f)\}$ *the set of all assignment pedigrees for field* A *in a*

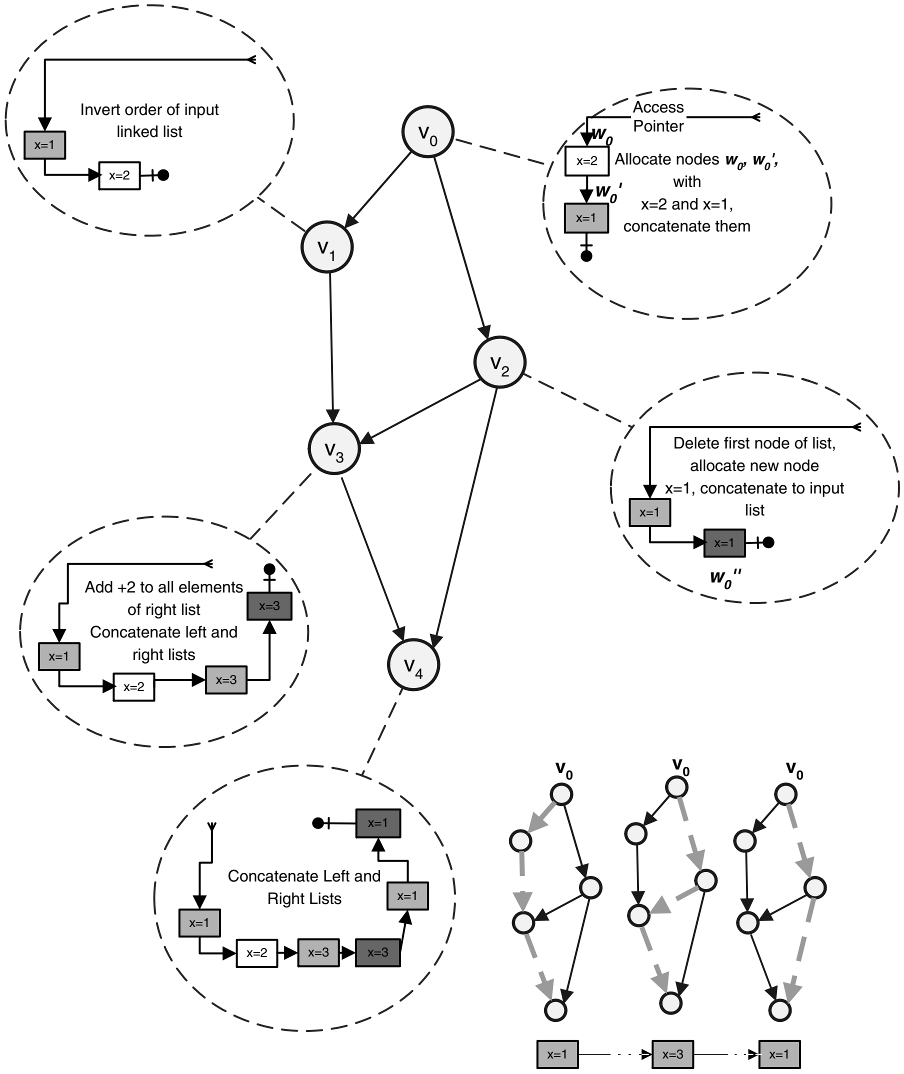
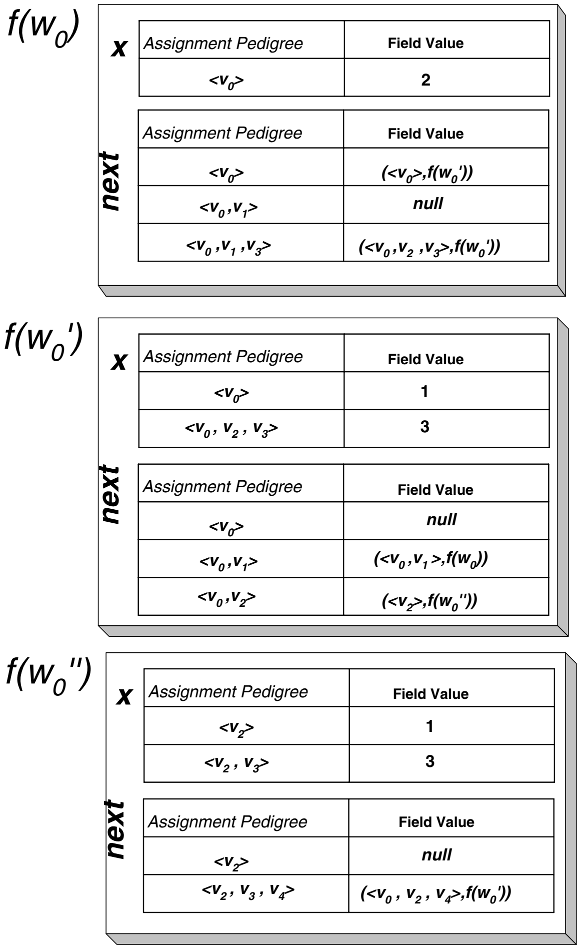

Invert order of input linked list
x=1
x=2
v0
v1
v2
v3
v4
Access Pointer
w0
x=2
Allocate nodes w0, w0', with x=2 and x=1, concatenate them
w0'
x=1
Delete first node of list, allocate new node x=1, concatenate to input list
x=1
x=1
w0''
Add +2 to all elements of right list Concatenate left and right lists
x=1
x=2
x=3
x=3
Concatenate Left and Right Lists
x=1
x=2
x=3
x=3
x=1
x=1
v0
v0
v0
x=1
x=3
x=1

FIGURE 31.1: A DAG of five versions. In each circle we show the corresponding update operation and the resulting version. Nodes with the same color originate from the same seminal node. The three gray nodes in version D_{v_4} all have the same seminal node (w'_0), and are distinguished by their pedigrees $\langle v_0, v_1, v_3, v_4\rangle$, $\langle v_0, v_2, v_3, v_4\rangle$, and $\langle v_0, v_2, v_4\rangle$.

fat note f, and denote it by $P(A, f)$. The table that represents field A in fat node f contains an entry for each assignment pedigree in $P(A, f)$. The value of a table entry, indexed by an assignment pedigree $p = \langle v_0, v_1, \ldots, v_j\rangle$, depends on the type of the field as follows. If A is a data field then the value stored is the value assigned to A in the node $w_j \in D_{v_j}$ whose pedigree is p. If A is a pointer field then let w be the node pointed to by field A after the assignment to A in w_j. We store the pedigree of w and the address of the fat node that represents the seminal node of w.

An access pointer to a node w in version v is represented by a pointer to the fat node representing the seminal node of w and the pedigree of w.

In Figure 31.2 we give the fat nodes of the persistent data structure given in Figure 31.1.



$f(w_0)$

x

Assignment Pedigree	**Field Value**
<v_0>	2

next

Assignment Pedigree	**Field Value**
<v_0>	(<v_0>,$f(w_0')$)
<v_0,v_1>	null
<v_0,v_1,v_3>	(<v_0,v_2,v_3>,$f(w_0')$)

$f(w_0')$

x

Assignment Pedigree	**Field Value**
<v_0>	1
<v_0, v_2, v_3>	3

next

Assignment Pedigree	**Field Value**
<v_0>	null
<v_0,v_1>	(<v_0,v_1>,$f(w_0)$)
<v_0,v_2>	(<v_2>,$f(w_0'')$)

$f(w_0'')$

x

Assignment Pedigree	**Field Value**
<v_2>	1
<v_2, v_3>	3

next

Assignment Pedigree	**Field Value**
<v_2>	null
<v_2, v_3, v_4>	(<v_0, v_2, v_4>,$f(w_0')$)

FIGURE 31.2: The fat nodes for the example of Figure 31.1.

For example, the field next has three assignments in nodes of $N(f(w'_0))$. Thus, there are three assignment pedigrees in $P(next, f(w'_0))$:

1. $\langle v_0 \rangle$ — allocation of w'_0 in version D_{v_0} and default assignment of null to next.
2. $\langle v_0, v_1 \rangle$ — inverting the order of the linked list in version D_{v_1} and thus assigning next a new value. The pointer is to a node whose pedigree is $\langle v_0, v_1 \rangle$ and whose seminal node is w_0. So we associate the value $(\langle v_0, v_1 \rangle, f(w_0))$ with $\langle v_0, v_1 \rangle$.
3. $\langle v_0, v_2 \rangle$ — allocating a new node, w''_0, in version D_{v_2}, and assigning next to point to this new node. The pedigree of w''_0 is $\langle v_2 \rangle$ so we associate the value $(\langle v_2 \rangle, f(w''_0))$ with $\langle v_0, v_2 \rangle$.

You can see all three entries in the table for next in the fat node $f(w'_0)$ (Figure 31.2). Similarly, we give the table for field x in $f(w'_0)$ as well as the tables for both fields in fat nodes $f(w_0)$ and $f(w''_0)$.

When we traverse the data structure we are pointing to some fat node f and hold a pedigree q of some node w whose seminal node corresponds to f and we would like to retrieve the value of field A in node w from the table representing field A in f. We do that as follows. First we identify the assignment pedigree $p(A, w)$ of field A in node w. This is the longest pedigree which is a prefix of q and has an entry in this table. In case A is a data field, the value we are after is simply the value associated with $p(A, w)$. However if A is a pointer field then the value stored with $p(A, w)$ may not be the value of A in w. This value identifies a node in the version where the assignment occurred, whereas we are interested in a node in the version of w that this pointer field points to.

Let $q = \langle q_0, \ldots, q_k \rangle$ and let $p(A, w) = \langle q_0, q_1, \ldots, q_j \rangle$. Let the value of $p(A, w)$ be (t, f), where t is the pedigree of the target node in D_{q_j} and f is the fat node representing the seminal node of this target node. The nodes identified by the pedigrees $p(A, w)$ and t were copied in versions $q_{j+1}, \ldots, q_k$ without any assignment made to field A in the nodes derived from the node whose pedigree is $p(A, w)$. Thus the pedigree of the target node of field A of node w in D_{q_k} is $t \| \langle q_{j+1}, \ldots, q_k \rangle$, where $\|$ represents concatenation.

It follows that we need representations for pedigrees and the tables representing field values that support an efficient implementation of the followings.

1. Given a pedigree q find the longest prefix of q stored in a table.
2. Given a pedigree q, replace a prefix of q with another pedigree p.
3. To facilitate updates we also need to be able to add a pedigree to a table representing some field with a corresponding value.

In their simplest simulation Fiat and Kaplan suggested to represent pedigrees as linked lists of version numbers, and to represent tables with field values as tries. Each assignment pedigree contained in the table is represented by a path in the corresponding trie. The last node of the path stores the associated value. Nodes in the trie can have large degrees so for efficiency we represent the children of each node in a trie by a splay tree.

Let U be the total number of assignments the simulation performs and consider the update creating version v. Then with this implementation each assignment performed during this update requires $O(d(v))$ words of size $O(\log U)$ bits, and takes $O(d(v) + \log U)$ time, where $d(v)$ is the depth of v in the DAG. Field retrieval also takes $O(d(v) + \log U)$ time.

The second method suggested by Fiat and Kaplan is the *compressed path method.* The essence of the compressed path method is a particular partition of our DAG into disjoint trees. This partition is defined such that every path enters and leaves any specific tree at most once. The compressed path method encodes paths in the DAG as a sequence of pairs of versions. Each such pair contains a version where the path enters a tree T and the version where the path leaves the tree T. The length of each such representation is $O(e(D))$.[‖] Each value of a field in a fat node is now associated with the compressed representation of the path of the node in $N(f)$ in which the corresponding assignment occurred. A key property of these compressed path representations is that they allow easy implementation of the operations we need to perform on pedigree, like replacing a prefix of a pedigree with another pedigree when traversing a pointer. With the compressed path method each assignment requires up to $O(e(D))$ words each of $O(\log \mathcal{U})$ bits. Searching or updating the trie representing all values of a field in a fat node requires $O(e(D) + \log \mathcal{U})$ time. For the case where the DAG is a tree this method degenerates to the fat node simulation of [18].

Fiat and Kaplan also suggested how to use randomization to speed up their two basic methods at the expense of (slightly) larger space expansion and polynomially small error probability. The basic idea is encode each path (or compressed path) in the DAG by an integer. We assign to each version a random integer, and the encoding of a path p is simply

[‖] Recall that $e(D)$ is the logarithm of the maximum number of different paths from the root of the DAG to any particular version.

the sum of the integers that correspond to the versions on p. Each value of a field in a fat node is now associated with the integer encoding the path of the node in $N(f)$ in which the corresponding assignment occurred. To index the values of each field we use a hash table storing all the integers corresponding to these values.

To deal with values of pointer fields we have to combine this encoding with a representation of paths in the DAG (or compressed paths) as balanced search trees, whose leaves (in left to right order) contain the random integers associated with the vertices along the path (or compressed path). This representation allows us to perform certain operations on these paths in logarithmic (or poly-logarithmic) time whereas the same operations required linear time using the simpler representation of paths in the non-randomized methods.

31.4 Making Specific Data Structures More Efficient

The purely functional deques of Kaplan and Tarjan [23], the confluently persistent deques of Kaplan, Okasaki, and Tarjan [21], the purely functional heaps of Brodal and Okasaki [6], and the purely functional finger search trees of Kaplan and Tarjan [22], are all based on a simple and useful mechanism called redundant counters, which to the best of our knowledge first appeared in lecture notes by Clancy and Knuth [9]. In this section we describe what redundant counters are, and demonstrate how they are used in simple persistent deques data structure.

A persistent implementation of deques support the following operations:

$q' = push(x, q)$: Inserts an element x to the beginning of the deque q returning a new deque q' in which x is the first element followed by the elements of q.

$(x, q') = pop(q)$: Returns a pair where x is the first element of q and q' is a deque containing all elements of q but x.

$q' = Inject(x, q)$: Inserts an element x to the end of the deque q returning a new deque q' in which x is the last element preceded by the elements of q.

$(x, q') = eject(q)$: Returns a pair where x is the last element of q and q' is a deque containing all elements of q but x.

A *stack* supports only push and pop, a *queue* supports only push and eject. Catenable deques also support the operation

$q = catenate(q_1, q_2)$: Returns a queue q containing all the elements of q_1 followed by the elements of q_2.

Although queues, and in particular catenable queues, are not trivial to make persistent, stacks are easy. The regular representation of a stack by a singly linked list of nodes, each containing an element, ordered from first to last, is in fact purely functional. To push an element onto a stack, we create a new node containing the new element and a pointer to the node containing the previously first element on the stack. To pop a stack, we retrieve the first element and a pointer to the node containing the previously second element.

Direct ways to make queues persistent simulate queues by stacks. One stack holds elements from the beginning of the queue and the other holds elements from its end. If we are interested in fully persistence this simulation should be real time and its details are not trivial. For a detailed discussion see Kaplan and Tarjan [23] and the references there.

Kaplan and Tarjan [23] described a new way to do a simulation of a deque with stacks. They suggest to represent a deque by a recursive structure that is built from bounded-size deques called *buffers*. Buffers are of two kinds: *prefixes* and *suffixes*. A non-empty deque q over a set A is represented by an ordered triple consisting of a *prefix*, $prefix(q)$, of elements of A, a *child deque*, $child(q)$, whose elements are ordered pairs of elements of A, and a *suffix*, $suffix(q)$, of elements of A. The order of elements within q is the one consistent with the orders of all of its component parts. The child deque *child*(q), if non-empty, is represented in the same way. Thus the structure is recursive and unwinds linearly. We define the descendants $\{child^i(\ q)\}$ of deque d in the standard way, namely $child^0(q) = q$ and $child^{i+1}(q) = child(child^i(q))$ for $i \geq 0$ if $child^i(q)$ is non-empty.

Observe that the elements of q are just elements of A, the elements of *child*(q) are pairs of elements of A, the elements of *child*(*child*(q)) are pairs of pairs of elements of A, and so on. One can think of each element of $child^i(q)$ as being a complete binary tree of depth i, with elements of A at its 2^i leaves. One can also think of the entire structure representing q as a stack (of q and its descendants), each element of which is prefix-suffix pair. All the elements of q are stored in the prefixes and suffixes at the various levels of this structure, grouped into binary trees of the appropriate depths: level i contains the prefix and suffix of $child^i(q)$. See Figure 31.3.

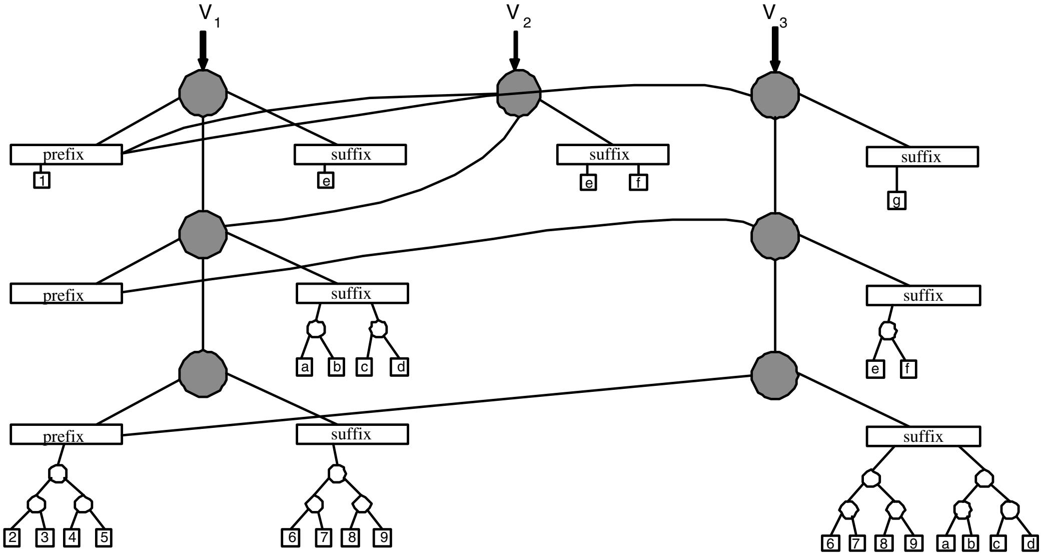


FIGURE 31.3: Representation of a deque of elements over A. Each circle denotes a deque and each rectangle denotes a buffer. Squares correspond to elements from A which we denote by numbers and letters. Each buffer contains 0, 1, or 2 elements. Three versions are shown V_1, V_2, and V_3. Version V_2 was obtained from V_1 by injecting the element f. Version V_3 obtained from version V_2 by injecting the element g. The latter inject triggered two recursive injects into the child and grandchild deques of V_2. Note that identical binary trees and elements are represented only once but we draw them multiple times to avoid cluttering of the figure.

Because of the pairing, we can bring *two* elements up to level i by doing *one pop* or *eject* at level $i + 1$. Similarly, we can move two elements down from level i by doing one *push* or *inject* at level $i + 1$. This two-for-one payoff leads to real-time performance.

Assume that each prefix or suffix is allowed to hold 0, 1, or 2 elements, from the beginning or end of the queue, respectively. We can implement $q' = push(x, q)$ as follows. If the prefix of q contains 0 or 1 elements we allocate a new node to represent q' make its child deque and its suffix identical to the child and suffix of q, respectively. The prefix of q' is a newly allocated prefix containing x and the element in the prefix of q, if the prefix of q contained one element. We return a pointer the new node which represents q'. For an example consider version V_2 shown in Figure 31.3 that was obtained from version V_1 by a case of inject symmetric to the case of the push just described.

The hard case of the push is when the prefix of q already contains two elements. In this case we make a pair containing these two elements and push this pair recursively into $child(q)$. Then we allocate a new node to represent q', make its suffix identical to the suffix of q, make the deque returned by the recursive push to $child(q)$ the child of q', and make the prefix of q' be a newly allocated prefix containing x. For an example consider version V_3 shown in Figure 31.3 that was obtained from version V_2 by a recursive case of inject symmetric to the recursive case of the push just described. The implementations of pop and eject is symmetric.

This implementation is clearly purely functional and therefore fully persistent. However the time and space bounds per operation are $O(\log n)$. The same bounds as one gets by using search trees to represent the deques with the path copying technique. These logarithmic time bounds are by far off from the ephemeral $O(1)$ time and space bounds.

Notice that there is a clear correspondence between this data structure and binary counters. If we think of a buffer with two elements as the digit 1, and of any other buffer as the digit 0, then the implementation of $push(q)$ is similar to adding one to the binary number defined by the prefixes of the queues $child^i(q)$. It follows that if we are only interested in partially persistent deques then this implementation has $O(1)$ amortized time bound per operation (see the discussion of binary counters in the next section). To make this simulation efficient in a fully persistent setting and even in the worst case, Kaplan and Tarjan suggested to use redundant counters.

31.4.1 Redundant Binary Counters

To simplify the presentation we describe redundant binary counters, but the ideas carry over to any basis. Consider first the regular binary representation of an integer i. To obtain from this representation the representation of $i+1$ we first flip the rightmost digit. If we flipped a 1 to 0 then we repeat the process on the next digit to the left. Obviously, this process can be long for some integers. But it is straightforward to show that if we carry out a sequence of such increments starting from zero then on average only a constant number of digits change per increment.[**] Redundant binary representations (or counters as we will call them) address the problem of how to represent i so we can obtain a representation of $i+1$ while changing only a constant number of digits in the worst case.

A *redundant binary representation*, d, of a non-negative integer x is a sequence of digits $d_n, \ldots, d_0$, with $d_i \in \{0, 1, \ldots, 2\}$, such that $x = \sum_{i=0}^{n} d_i 2^i$. We call d *regular* if, between any two digits equal to 2, there is a 0, and there is a 0 between the rightmost 2 and the least significant digit. Notice that the traditional binary representation of each integer (which does not use the digit 2) is *regular*. In the sequel when we refer to a regular representation we mean a regular redundant binary representation, unless we explicitly state otherwise.

Suppose we have a regular representation of i. We can obtain a regular representation of $i+1$ as follows. First we increment the rightmost digit. Note that since the representation of i is regular, its rightmost digit is either 0 or 1. So after the increment the rightmost digit

[**] The rightmost digit changes every increment, the digit to it left changes every other operation, and so on.

is either 1 or 2 and we still have a redundant binary representation for $i+1$. Our concern is that this representation of $i+1$ may not be regular. However, since the representation of i we started out with was regular the only violation to regularity that we may have in the representation of $i+1$ is lacking a 0 between the rightmost 2 and the least significant digit. It is easy to check that between any two digits equal to 2, there still is a 0, by the regularity of i.

We can change the representation of $i+1$ to a representation which is guaranteed to be regular by a simple *fix* operation. A *fix* operation on a digit $d_i = 2$ increments d_{i+1} by 1 and sets d_i to 0, producing a new regular representation d' representing the same number as d.[††] If after incrementing the rightmost digit we perform a fix on the rightmost 2 then we switch to another representation of $i+1$ that must be regular. We omit the proof here which is straightforward.

It is clear that the increment together with the fix that may follow change at most three digits. Therefore redundant binary representations allow to perform an increment while changing constantly many digits. However notice that in any application of this numbering system we will also need a representation that allows to get to the digits which we need to fix efficiently. We show one such representation in the next section.

These redundant representations can be extended so we can also decrement it while changing only a constant number of digits, or even more generally so that we can increment or decrement any digit (add or subtract 2^i) while changing a constant number of other digits. These additional properties of the counters were exploited by other applications (see e.g. [22, 24]).

31.4.2 Persistent Deques

Kaplan and Tarjan use this redundant binary system to improve the deque implementation we described above as follows. We allow each of the prefixes and suffixes to contain between 0 and 5 elements. We label each buffer, and each deque, by one of the digits 0, 1, and 2. We label a buffer 0 if it has two or three elements, we label it 1 if it has one or four elements, and we label it 2 if it has zero or five elements. Observe that we can add one element to or delete one element from a buffer labeled 0 or 1 without violating its size constraint: A buffer labeled 0 may change its label to 1, and a buffer labeled 1 may change its label to 2. (In fact a 1 can also be changed to 0 but this may not violate regularity.) The label of a deque is the larger among the labels of its buffers, unless its child and one of its buffers are empty, in which case the label of the deque is identical to the label of its nonempty buffer.

This coloring of the deques maps each deque to a redundant binary representation. The least significant digit of this representation is the digit of q, the next significant digit is the digit of $child(q)$, and, in general, the i^{th} significant digit is the digit corresponding to $child^i(q)$ if the latter is not empty. We impose an additional constraint on the deques and require that the redundant binary representation of any top-level deque is regular.

A regular top-level deque is labeled 0 or 1 which implies that both its prefix and its suffix are labeled 0 or 1. This means that any deque operation can be performed by operating on the appropriate top-level buffer. Suppose that the operation is either a push or a pop, the case of inject and eject is symmetric. We can construct the resulting queue q' by setting $child(q') = child(q)$ and $suffix(q') = suffix(q)$. The prefix of q' is a newly allocated buffer that contains the elements in $prefix(q)$ together with the new element in case of push or

[††] We use the *fix* only when we know that d_{i+1} is either 0 or 1.

without the first element in case of pop. Clearly all these manipulations take constant time.

The label of q', however, may be one larger than the label of q. So the redundant binary representation corresponding to q' is either the same as the redundant binary representation of q in which case it is regular, or it is obtained from the redundant binary representation of q by incrementing the least significant digit. (The least significant digit can also decrease in which case regularity is also preserved.) This corresponds to the first step in the increment procedure for redundant regular representations described in the previous section.

To make the redundant binary representation of q' regular we may have to apply a fix operation. Let i be the minimum such that $child^i(q')$ is labeled 2. If for all $j < i$, $child^j(q')$ is labeled 1 then the fix has to change the label of $child^i(q')$ to 0 and increment the label of $child^{i+1}(q')$.

Fortunately, we have an appropriate interpretation for such a fix. Assume $child^{i+1}(q')$ have a non-empty child. (We omit the discussion of the case where $child^{i+1}(q')$ have an empty child which is similar.) We know that the label of $child^{i+1}(q')$ is either 0 or 1 so neither of its buffers is empty or full. If the prefix of $child^i(q')$ has at least four elements we eject 2 of these elements and push them as a single pair to the prefix of $child^{i+1}(q')$. If the prefix of $child^i(q')$ has at most one element we pop a pair from the prefix of $child^{i+1}(q')$ and inject the components of the pair into the prefix of $child^i(q')$. This makes the prefix of $child^i(q')$ containing either two or three elements. Similarly by popping a pair from or pushing a pair to the suffix of $child^i(q')$, and injecting a pair to or ejecting a pair from the suffix of $child^{i+1}(q')$ we make the suffix of $child^i(q')$ containing two or three elements. As a result the label of $child^i(q')$ and its two buffers becomes 0 while possibly increasing the label of one or both buffers of $child^{i+1}(q')$ and thereby the label of $child^{i+1}(q')$ as well.

There is one missing piece for this simulation to work efficiently. The topmost deque labeled 2 may be arbitrarily deep in the recursive structure of q', since it can be separated from the top level by many deques labeled 1. To implement the fix efficiently we have to be able to find this deque fast and change it in a purely functional way by copying the deques that change without having to copy all their ancestors deques.

For this reason we do not represent a deque in the obvious way, as a stack of prefix-suffix pairs. Instead, we break this stack up into substacks. There is one substack for the top-level deque and one for each descendant deque labeled 0 or 2 not at the top level. Each substack consists of a top-level, or a deque labeled 0, or a deque labeled 2 and all consecutive proper descendant deques labeled 1. We represent the entire deque by a stack of substacks of prefix-suffix pairs using this partition into substacks. This can be realized with four pointers per each node representing a deque at some level. Two of the pointers are to the prefix and suffix of the deque. One pointer is to the node for the child deque if this deque is non-empty and labeled 1. One pointer is to the node of the nearest proper descendant deque not labeled 1, if such a deque exists and q itself is not labeled 1 or top-level. See Figure 4.2.

A single deque operation will require access to at most the top three substacks, and to at most the top two elements in any such substack. The label changes caused by a deque operation produce only minor changes to the stack partition into substacks, changes that can be made in constant time. In particular, changing the label of the top-level deque does not affect the partition into substacks. Changing the label of the topmost deque which is labeled 2 to 0 and the label of its child from 1 to 2 splits one substack into its first element, now a new substack, and the rest. This is just a substack pop operation. Changing the label of the topmost deque which is labeled 2 to 0 and the label of its child from 0 to 1 merges a singleton substack with the substack under it. This is just a substack push operation.

To add catenation, Kaplan and Tarjan had to change the definition of the data structure and allow deques to be stored as components of elements of recursive deques. The redundant

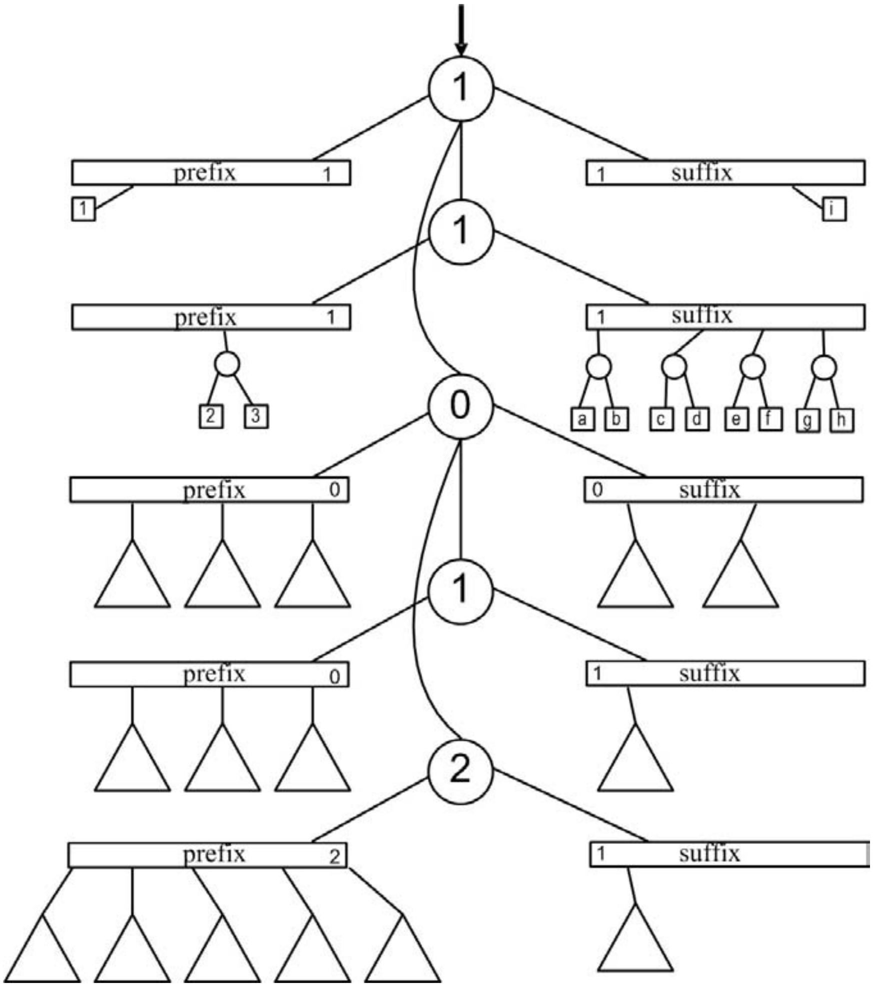


FIGURE 31.4: Pointer representation of stack of substacks structure. Each circle corresponds to a deque and it is marked by its label. Each buffer is a rectangle which is marked by its label. Triangles denote complete binary trees of elements whose depths depend on the level. This particular queue is represented by a stack of three substacks.

binary numbering system, however, still plays a key role. To represent a catenable deque, Kaplan and Tarjan use noncatenable deques as the basic building blocks. They define a *triple* over a set A recursively as a prefix of elements of A, a possibly empty deque of triples over A, and a suffix of elements of A, where each prefix or suffix is a noncatenable deque. Then, they represent a catenable deque of elements from A by either one or two triples over A. The underlying skeleton of this structure is a binary tree (or two binary trees) of triples. The redundant binary number system is extended so that it can distribute work along these trees by adding an extra digit.

Kaplan, Okasaki, and Tarjan [21] simplified these data structures at the expense of making the time bounds amortized rather than worst case and using assignment, thus obtaining a confluently persistent data structure which is not purely functional. The key idea underlying their data structure is to relax the rigid constraint of maintaining regularity. Instead, we "improve" the representation of a deque q with full or empty prefix when we try to push or pop an element from it. Similarly, with full or empty suffix. This improvement in the representation of q is visible to all deques that contain q as a subdeque at some level and prevents from pushing into deques with full prefixes or popping from deques with empty prefixes from happening too often.

More specifically, assume that we push into a deque q with full prefix. First, we eject two elements from this prefix, make a pair containing them, and push the pair recursively into $child(q)$. Let the result of the recursive push be $child'(q)$. Then we change the representation of q so that it has a new prefix which contains all the elements in the prefix of q but the two which we ejected, and its child deque is $child'(q)$. The suffix of q does not change. Finally we perform the push into q by creating a new queue q' that has the same suffix and child deque as q, but has a new prefix that contains the elements in the prefix of q together with the new element. A careful but simple analysis shows that each operation in this implementation takes $O(1)$ amortized time. By extending this idea, Kaplan, Okasaki, and Tarjan managed to construct catenable deques using only constant size buffers as the basic building blocks.

31.5 Concluding Remarks and Open Questions

Much progress has been made on persistent data structures since the seminal paper of Driscoll et al. [18]. This progress has three folds: In developing general techniques to make any data structure persistent, in making specific data structures persistent, and in emerging algorithmic applications. Techniques developed to address these challenges sometimes proved useful for other applications as well.

This algorithmic field still comprise intriguing challenges. In developing general techniques to make data structures persistent, a notable challenge is to find a way to make the time slowdown of the node splitting method worst case. Another interesting research track is how to restrict the operations that combine versions in a confluently persistent setting so that better time bounds, or simpler simulations, are possible. We also believe that the techniques and data structures developed in this field would prove useful for numerous forthcoming applications.

Acknowledgment

This work was supported, in part, by the Israel Science Foundation (ISF) under grant 548/00.

References

[1] S. Alstrup, G. S. Brodal, I. L. Gørtz, and T. Rauhe. Time and space efficient multi-method dispatching. In *Proc. 8th Scandinavian Workshop on Algorithm Theory*, volume 2368 of *Lecture Notes in Computer Science*, pages 20–29. Springer, 2002.

[2] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *Proc. 10th Annual European Symposium on Algorithms (ESA 2002)*, volume 2461 of *Lecture Notes in Computer Science*, pages 152–164. Springer, 2002.

[3] A. Boroujerdi and B. Moret. Persistence in computational geometry. In *Proc. 7th Canadian Conf. Comp. Geometry (CCCG 95)*, pages 241–246, 1995.

[4] P. Bozanis, N. Kitsios, C. Makris, and A. Tsakalidis. The space-optimal version of

a known rectangle enclosure reporting algorithm. *Information Processing Letters*, 61(1):37–41, 1997.

[5] G. S. Brodal. Partially persistent data structures of bounded degree with constant update time. *Nordic Journal of Computing*, 3(3):238–255, 1996.

[6] G. S. Brodal and C. Okasaki. Optimal purely functional priority queues. *Journal of Functional Programming*, 6(6):839–858, 1996.

[7] A. L. Buchsbaum and R. E. Tarjan. Confluently persistent deques via data structural bootstrapping. *J. of Algorithms*, 18:513–547, 1995.

[8] B. Chazelle. How to search in history. *Information and control*, 64:77–99, 1985.

[9] M. J. Clancy and D. E. Knuth. A programming and problem-solving seminar. Technical Report STAN-CS-77-606, Department of Computer Science, Stanford University, Palo Alto, 1977.

[10] R. Cole. Searching and storing similar lists. *J. of Algorithms*, 7:202–220, 1986.

[11] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 19th Annual ACM Symposium on Theory of Computing*, pages 365–372, 1987.

[12] Paul F. Dietz. Fully persistent arrays. In *Proc. 1st Worksh. Algorithms and Data Structures (WADS 1989)*, volume 382 of *Lecture Notes in Computer Science*, pages 67–74. Springer, 1989.

[13] Paul F. Dietz and Rajeev Raman. Persistence, amortization and randomization. In *Proc. 2nd annual ACM-SIAM symposium on Discrete algorithms*, pages 78–88. Society for Industrial and Applied Mathematics, 1991.

[14] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, M. Meyer auf der Heide, H. Rohnert, and R.E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. on Computing*, 23(4):738–761, 1994.

[15] D. P. Dobkin and R. J. Lipton. Multidimensional searching problems. *SIAM J. on Computing*, 5(2):181–186, 1976.

[16] D. P. Dobkin and J. I. Munro. Efficient uses of the past. *J. of Algorithms*, 6:455–465, 1985.

[17] J. Driscoll, D. Sleator, and R. Tarjan. Fully persistent lists with catenation. *Journal of the ACM*, 41(5):943–959, 1994.

[18] J. R. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan. Making data structures persistent. *J. of Computer and System Science*, 38:86–124, 1989.

[19] Amos Fiat and Haim Kaplan. Making data structures confluently persistent. *Journal of Algorithms*, 48(1):16–58, 2003. Symposium on Discrete Algorithms.

[20] P. Gupta, R. Janardan, H. M. Smid, and B. DasGupta. The rectangle enclosure and point-dominance problems revisited. *International Journal of Computational Geometry and Applications*, 7(5):437–455, 1997.

[21] H. Kaplan, C. Okasaki, and R. E. Tarjan. Simple confluently persistent catenable lists. *Siam J. on Computing*, 30(3):965–977, 2000.

[22] H. Kaplan and R. E. Tarjan. Purely functional representations of catenable sorted lists. In *Proc. 28th Annual ACM Symposium on Theory of Computing*, pages 202–211. ACM Press, 1996.

[23] H. Kaplan and R. E. Tarjan. Purely functional, real-time deques with catenation. *Journal of the ACM*, 46(5):577–603, 1999.

[24] Haim Kaplan, Nira Shafrir, and Robert Endre Tarjan. Meldable heaps and boolean union-find. In *Proc. 34th Annual ACM Symposium on Theory of Computing (STOC)*, pages 573–582, 2002.

[25] N. Kitsios and A. Tsakalidis. Space reduction and an extension for a hidden line elimination algorithm. *Computational Geometry*, 6(6):397–404, 1996.

[26] E. M. McCreight. Priority search trees. *Siam J. on Computing*, 14:257–276, 1985.

[27] O. Nurmi. A fast line sweep algorithm for hidden line elimination. *BIT*, 25(3):466–472, 1985.

[28] C. Okasaki. Amortization, lazy evaluation, and persistence: Lists with catenation via lazy linking. In *IEEE Symposium on Foundations of Computer Science*, pages 646–654, 1995.

[29] M. H. Overmars. Searching in the past, I. Technical Report RUU-CS-81-7, Department of Computer Science, University of Utrecht, Utrecht, The Netherlands, 1981.

[30] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, 1986.

[31] R. E. Tarjan. *Data Structures and Network algorithms.* SIAM, Philadelphia, 1982.

[32] R. E. Tarjan and J. Van Leeuwen. Worst case analysis of set union algorithms. *Journal of the ACM*, 31:245–281, 1984.

[33] P. van Emde Boaz. Preserving order in a forest in less than logarithmic time. *Information Processing Letters*, 6(3):80–82, 1977.

[34] P. van Emde Boaz, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

32

PQ Trees, PC Trees, and Planar Graphs

Wen-Lian Hsu
Academia Sinica

Ross M. McConnell
Colorado State University

32.1 Introduction

A graph is *planar* if it is possible to draw it on a plane so that no edges intersect, except at endpoints. Such a drawing is called a *planar embedding*.

Not all graphs are planar: Figure 32.1 gives examples of two graphs that are not planar. They are known as K_5 , the complete graph on five vertices, and $K_{3,3}$, the complete bipartite graph on two sets of size 3. No matter what kind of convoluted curves are chosen to represent the edges, the attempt to embed them always fails when the last of the edges cannot be inserted without crossing over some other edge, as illustrated in the figure.

There is considerable practical interest in algorithms for finding planar embeddings of planar graphs. An example of an application of this problem is where an engineer wishes to embed a network of components on a chip. The components are represented by wires, and no two wires may cross without creating a short circuit. This problem can be solved by treating the network as a graph and finding a planar embedding of it. Planar graphs play a central role in geographic information systems, and in many problems in computational geometry.

The study of planar graphs dates to Euler. The *faces* of an embedding are connected regions of the plane that are separated from each other by cycles of G. Euler showed that for any planar embedding, if V is the set of vertices, E the set of edges, F the set of faces (regions of the plane that are connected in the embedding), and C the set of connected components of the graph, then $|V| + |F| = |E| + |C| + 1$. Many other results about planar

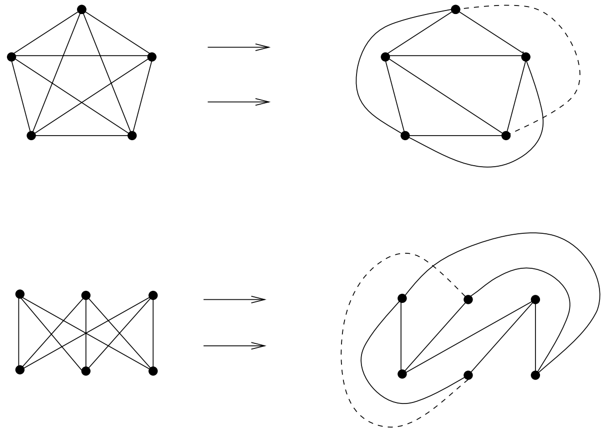


FIGURE 32.1: Two non-planar graphs. The first is the K_5, the complete graph on five vertices, and the second is the $K_{3,3}$, the complete bipartite class on two sets of three vertices each. Any attempt to embed them in the plane fails when a final edge cannot be inserted without crossing the boundary between two faces.

graphs can be proven using this formula. For instance, using the formula, it is easily proven with counting arguments that K_5 and $K_{3,3}$ are non-planar [12].

The famous 4-color theorem states that the vertices of a planar graph can always be partitioned into four independent sets; an equivalent statement is that a mapmaker never needs to use more than four colors to color countries on a map so that adjacent countries are of different colors. It remained open in the literature for almost 100 years and was finally proven with the aid of a computer program in 1976 [1, 2].

A *subdivision* of an edge xy of a graph is obtained by creating a new node z, and replacing xy with new edges xz and zy. The inverse of this operation is the *contraction* of z, and only operates on vertices of degree 2. A subdivision of a graph is any graph that can be obtained from it by a sequence of subdivision operations. Since K_5 and $K_{3,3}$ are non-planar, it is obvious that subdivisions of these graphs are also non-planar. Therefore, a graph that has a subgraph that is a subdivision of K_5 or $K_{3,3}$ as a subgraph must be non-planar. Such a subgraph is said to be *homeomorphic* to a $K_{3,3}$ or a K_5.

A famous results in graph theory is the theorem of Kuratowski [21], which states that the absence of a subdivision of a K_5 or a $K_{3,3}$ is also sufficient for a graph to be planar. That is, a graph is planar if and only if it has no subgraph that is a subdivision of $K_{3,3}$ or K_5. Such a subdivision is known as a *Kuratowski subgraph.*

A *certifying algorithm* for a decision problem is one that produces an accompanying piece of evidence, or *certificate* that proves that its answer is correct. The certificate should be simple to check, or *authenticate.* Certifying algorithms are highly desirable in practice, where the possibility must be considered that an implementation has a bug and a simple yes or no answer cannot be entirely trusted unless it is accompanied by a certificate. The issue is discussed at length in [20]. Below, we describe a certifying algorithm for recognizing planar graphs. The algorithm produces either a planar embedding of the graph, proving that the graph is planar, or points out a Kuratowski subgraph, proving that it is not.

Next, let us consider a problem that is seemingly unrelated to that of finding a planar embedding of a graph, but which can be solved with similar data structures. Given a set S of intervals of a line, let their *interval graph* be the graph that has one vertex for each of the intervals in S, and an edge between two vertices if their intervals intersect. That

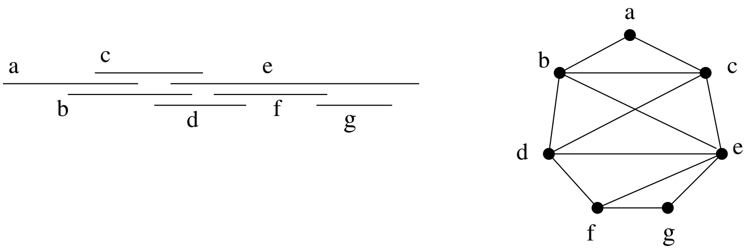


FIGURE 32.2: An interval graph is the intersection graph of a set of intervals on a line. There is one vertex for each of the intervals, and two vertices are adjacent if and only if the corresponding intervals intersect.

is, a graph is an interval graph if it is the *intersection graph* of a set of intervals on a line. Figure 32.2 gives an illustration.

Interval graphs also come up in a variety of other applications, such as scheduling jobs that conflict if they must be carried out during overlapping time intervals. If an interval representation is given, otherwise NP-complete problems, such as finding a maximum independent set, can be solved in linear time [9].

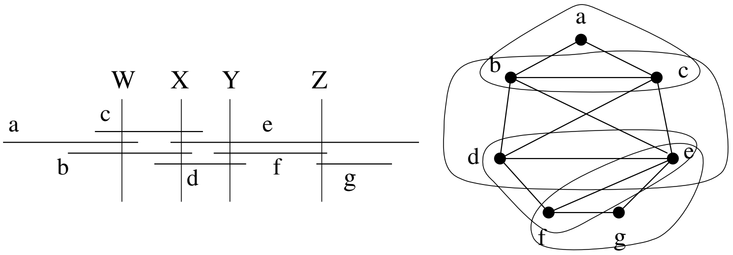
Given the intervals, it is trivial to construct their interval graph. However, we are interested in the inverse problem, where, given a graph G, one must find a set of intervals that have G as their interval graph or else determine that G is not an interval graph.

Interest in this problem began in the late 1950's when the noted biologist Seymour Benzer used them to establish that genetic information is stored inside a biological structure that has a linear topology [3]; this topology arises from the now-familiar structure of DNA. To do this, he developed methods of inducing mutations using X-ray photons, which could be assumed to reflect damage to a contiguous region, and for testing whether two of these mutations had common effects that indicated that the the damaged regions intersect. This gave rise naturally to a graph where each mutation is a vertex and where two vertices have an edge between them if they intersect. He got the result by showing that this graph is an interval graph.

Let us say that such a set S is a *realizer* of the interval graph G if G is S's interval graph. Benzer's result initiated considerable interest in efficient algorithms to finding realizers of interval graphs, since they give possible linear orderings of DNA fragments, or *clones*, given data about which fragments intersect [5, 10, 11, 16, 17, 19, 26–28]. A linear-time bound for the problem was first given by Booth and Lueker in [5]. Though the existence of forbidden subgraphs of interval graphs has long been well-known [22], the first linear-time certifying algorithm for recognizing interval graphs has only been given recently [20]; the certificate of acceptance is an interval realizer and the certificate of rejection is a forbidden subgraph.

When the ordering of intervals is unique except for trivial details, such as the lengths of the intervals and the relative placement of endpoints that intersect, this solves the *physical mapping problem* on DNA clones: it tells how the clones are arranged on the genome. Efficient algorithms for solving certain variations of this problem played a role in the assembling the genomes of organisms, and continue to play a significant role in genetic research [39]. For input data containing errors, Lu and Hsu [23] give an error-tolerant algorithm for the clone assembly problem.

A graph is a *circular-arc graph* if it is the intersection graph of a set of arcs on a circle. Booth conjectured that recognizing whether a graph is a circular-arc graph would turn out to be NP complete [4], but Tucker later found a polynomial-time algorithm [38]. McConnell has recently found a linear-time algorithm [29]. The problem of finding a certifying algorithm



$$\begin{array}{ccccc} & W & X & Y & Z \\ a & 1 & 0 & 0 & 0 \\ g & 0 & 0 & 0 & 1 \\ c & 1 & 1 & 0 & 0 \\ e & 0 & 1 & 1 & 1 \\ d & 0 & 1 & 1 & 0 \\ f & 0 & 0 & 1 & 1 \\ b & 1 & 1 & 0 & 0 \end{array}$$

FIGURE 32.3: The clique matrix of a graph has one row for each vertex, one column for each maximal clique, and a 1 in row i column j iff vertex i is contained in clique j. The maximal cliques of an interval graph correspond to points of maximal overlap in an interval representation. Ordering the columns of a clique matrix in the order in which they appear in an interval representation gives a consecutive-ones ordering of the clique matrix.

for the problem remains open.

Finding the maximal cliques of an arbitrary graph is hard: in fact it is NP complete to find whether a graph has a clique of a given size k. However, if a graph is *chordal* it is possible to list out its maximal cliques in linear time [32], and interval graphs are chordal. (A chordal graph is one that has no simple cycle on four or more vertices as an induced subgraph.) We may therefore create a *clique matrix*, which has one row for each vertex of the graph, one column for each maximal clique, and a 1 in row i, column j iff clique j contains vertex i.

THEOREM 32.1 *A chordal graph is an interval graph iff there is a way to order the columns of its clique matrix so that, in every row, the 1's are consecutive.*

To see this, suppose G is an interval graph and S is a realizer. Then, for each maximal clique C, a *clique point* on the line can be selected that intersects the intervals that correspond to elements of C and no others. (See Figure 32.3.) Ordering the columns of the clique matrix according to the left-to-right order of the corresponding clique points ensures that the 1's in each row will be consecutive. Conversely, given a consecutive-ones ordering, the 1's in each row occupy an interval on the sequence of columns. It is easy to see that these intervals constitute a realizer of G, since two vertices are adjacent iff they are members of a common maximal clique.

Such an ordering of the columns of a 0-1 matrix is known as a *consecutive-ones ordering*, and a 0-1 matrix has the *consecutive-ones property* if there exists a consecutive-ones ordering of it. The main thrust of Booth and Lueker's algorithm consists of an algorithm for determining whether there exists a a consecutive-ones ordering of the columns of a 0-1 matrix. Their algorithm operates on a sparse representation of the matrix, and solves this in time linear in the number of 1's in the matrix. To test for the consecutive-ones property, they developed a representation, called a *PQ tree*, of *all* the consecutive-ones orderings of the columns. The tree consists of *P nodes* and *Q nodes.* The leaves of the tree are columns of the matrix, and the left-to-right leaf order of the tree gives a consecutive-ones ordering, just as it does when the order of children of a node are reversed, or when the order of children of a P node are permuted arbitrarily (see Figure 32.4). All consecutive-ones orderings of the columns can be obtained by a sequence of such rearrangements.

The PQ tree helps with keeping track of possible consecutive-ones orientations as they work by induction on the number of rows of the matrix. Each interval realizer of G is given

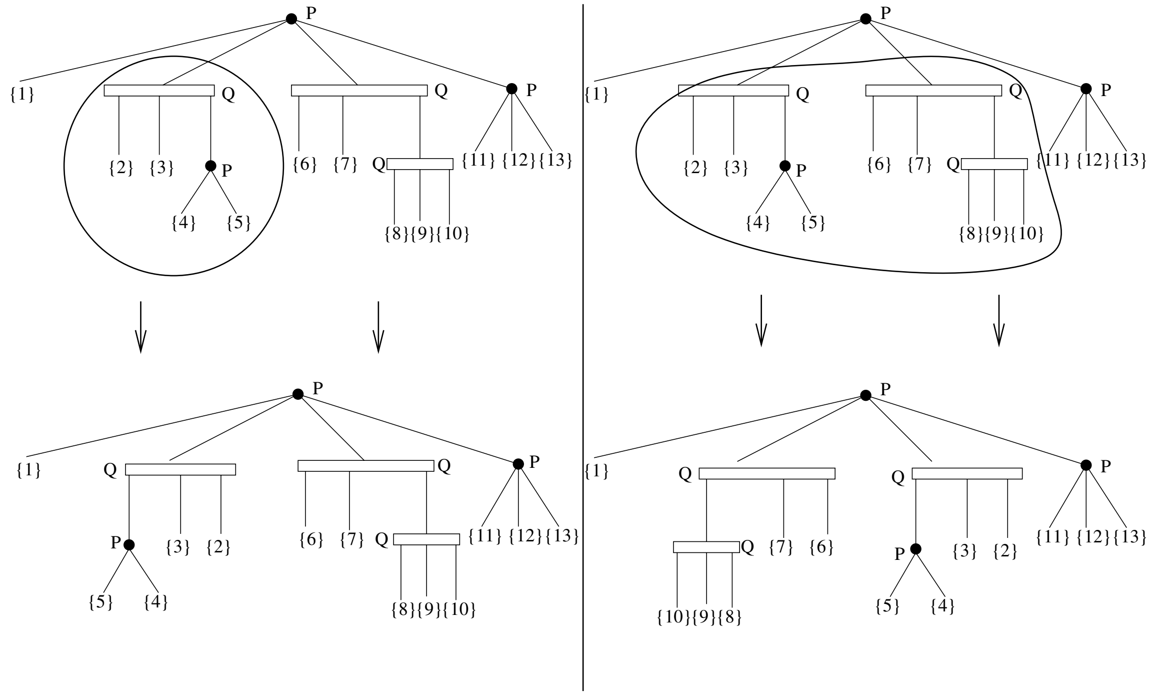


FIGURE 32.4: The leaves of a PQ tree are the columns of a consecutive-ones matrix. The left-to-right order of the leaves gives a consecutive-ones arrangement of the columns. So does the result of reversing the leaf descendants of a node. The order of leaves of a consecutive set of children of a P node can also be reversed to obtain a new consecutive-ones ordering. All consecutive-ones orderings can be obtained by a sequence of these reversals.

by a consecutive-ones ordering, except for minor details that do not affect the order of clique points.

The literature on problems related to PQ trees is quite extensive. Korte and Möhring [19] considered a modified PQ tree and a simpler incremental update of the tree. Klein and Reif [18] constructed efficient parallel algorithms for manipulating PQ trees. Hsu gave a simple test that is not based on PQ trees [15].

McConnell gives a generalization of the PQ tree to arbitrary 0-1 matrices, gives a linear-time algorithm for producing it, and a linear-time certifying algorithm for recognizing the consecutive-ones property [25].

The PQ tree play an important role in the linear-time algorithm of Lempel, Even, and Cederbaum for finding a planar embedding of planar graphs [24]. The algorithm takes advantage of the PQ tree's rich ability to represent families of linear orderings in order to keep track of possible arrangements of edges in an embedding of G.

Booth and Lueker's algorithm for constructing the PQ tree has a reputation for being difficult to understand and to program, and the many algorithms that have appeared since reflect an effort to address this concern. Their algorithm builds the tree by induction on the number of rows of the matrix. For each row, it must perform a second induction from the leaves toward the root. At each node encountered during this second induction, it uses one of nine *templates* for determining how the tree must change in the vicinity of the node. Recognizing which template must be used is quite challenging. Each template is actually a representative of a larger set of cases that must be dealt with explicitly by a program.

These templates carry over into the use of the PQ tree in planar graph embedding.

The *PC tree* is an alternative introduced by Shih and Hsu [34] to address these difficulties. It is essentially the result of "unrooting" the PQ tree to obtain a free tree that awards no special status to any root node, and where notions of "up" and "down" in the tree have no meaning. This introduces a symmetry to the problem that is otherwise broken by the choice of the root, and once it is introduced, the various templates collapse down to a single case.

This suggests that the cases that must be considered in the templates are an artifact of an arbitrary choice of a root in the tree. The reason that this was not recognized earlier may have more to do with the fact that rooted trees are ubiquitous as data structures, whereas free trees are not commonly used as data structures. The need to root such a data may simply have been an assumption that people failed to scrutinize.

A matrix has the *circular-ones* property if the columns can be ordered so that, in every row, either the zeros are consecutive or the ones are consecutive. That is, it has the circular-ones property if the ones are consecutive when the matrix is wrapped around a vertical cylinder, which has the effect of eliminating any special status to any column, such as being the leftmost column.

Hsu [14] gives an algorithm using PC trees for solving the consecutive-ones problem. Hsu and McConnell [17] have shown that that both the PQ tree and the PC tree have remarkably simple definitions as mathematical objects. They are each precisely given by previously-known theorems on set families that had not previously been applied in this domain. Moreover, we show that the PC tree gives a representation of all circular-ones orderings of a matrix just as the PQ tree gives a representation of all consecutive-ones orderings.

Figure 32.5 illustrates how the PC tree represents the circular-ones orderings. The leaves are the columns of the matrix, and are arrayed around the large circle, which represents the circular ordering. The C nodes (double circles) have a cyclic order on their edges that can be reversed. We could think of them as coins with edges attached at discrete points around the sides, and that can be turned heads-up or tails-up, an operation that we will call a *flip*. The P nodes (black internal nodes) have no cyclic ordering. The circular-ones orderings of the columns of the matrix are just those that result from planar embeddings of this gadget that put the leaves on the outer circle. This description makes it obvious what family of circular orderings is represented: you can select an edge and reverse the order of all leaves that lie on one side of the edge, or you can reverse the order of a consecutive set of leaves if they are the leaves of a subset of the trees in the forest that would result from the removal of a P node.

Booth and Lueker showed that testing for the circular-ones property reduces in linear time to testing for the consecutive-ones property. It appears to be more natural to perform the reduction in the opposite direction. That is, to solve the consecutive-ones problem reduce it to the circular-ones problem, which can be solved with the PC tree instead of with the PQ tree. To do this, just add the zero vector as a new column of the matrix, compute the PC tree for the new matrix, and then pick it up by the leaf corresponding to the new column to root it. (See Figure 32.6.) In [17], it is shown that the subtree rooted at its child is the PQ tree for the original matrix.

32.2 The Consecutive-Ones Problem

In this section, we give an algorithm that is related to Booth and Lueker's algorithm, except that it uses the PC tree in place of the PQ tree.

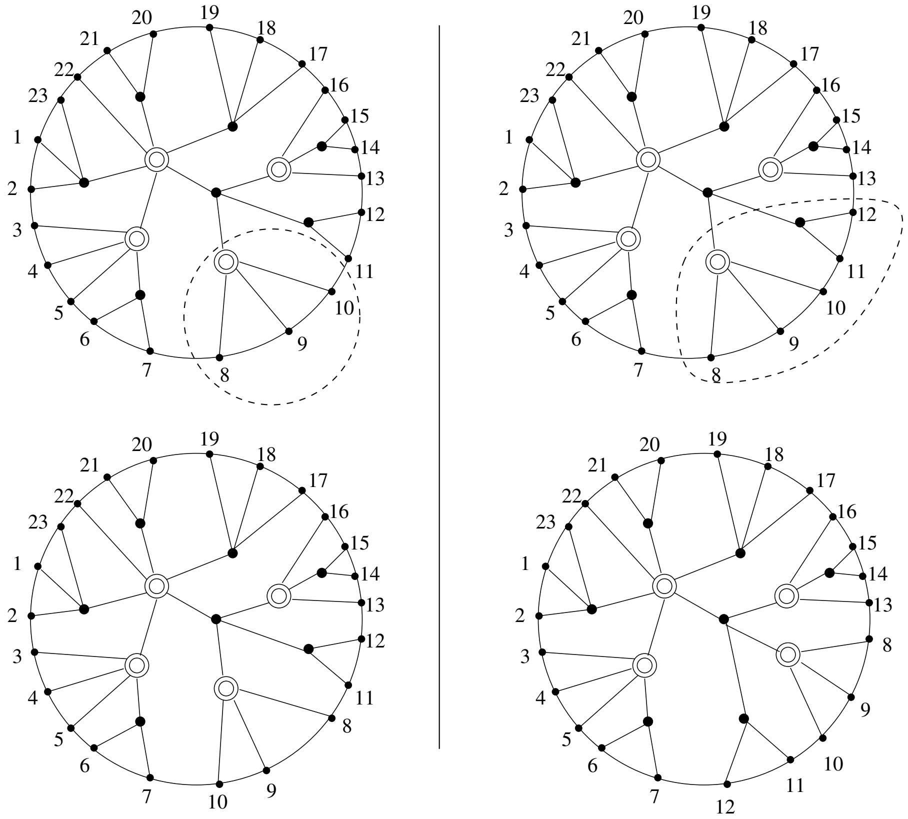


FIGURE 32.5: The PC tree can be viewed as a gadget for generating the circular-ones orderings of the columns. The C nodes are represented by double circles and the P nodes are represented by black dots. The subtree lying at one side of an edge can be flipped over to reverse the order of its leaves. The order of leaves of a consecutive set of subtrees that would result from the removal of a P node can also be reversed. All circular-ones orderings can be obtained by a sequence of such reversals.

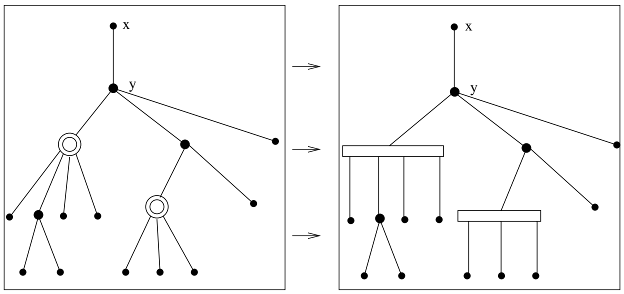


FIGURE 32.6: Assigning a new zero column x to a matrix, computing the PC tree for it, and then picking the PC tree up at x to root it, gives the PQ tree for the matrix, rooted at y, when the C nodes are reinterpreted as Q nodes.

Let us say that two subsets X and Y of a domain V *strongly overlap* if $X \cap Y$, $X - Y \cup Y - X$, and $V - X - Y$ are all nonempty. We view the columns of a 0-1 matrix as a set V, and each row of the matrix as a subset of V consisting of those columns where there is a 1 in the row.

A set X is an *edge module* if it is the union of leaves in one of the two subtrees that results when an edge is removed. It is a *P module* if it is not an edge module but the union of leaves in a subset of the trees formed when a P node is removed. An edge or P module is an *unrooted module.* The key to understanding the construction of the PC tree is the fact that the unrooted modules are precisely those nonempty proper subsets of V that do not strongly overlap any row of the matrix.

We construct the PC tree by induction on the number of rows of a matrix. The i^{th} step of the algorithm modifies the PC tree so that it is correct for the submatrix consisting of the first i rows of the matrix. As a base case, after the first step, the PC tree consists of two adjacent P nodes, with one of them adjacent to the leaves that correspond to ones in the first row and the other adjacent to the leaves that correspond to the zeros.

During the i^{th} step, no new unrooted modules are created by adding a row, but some unrooted modules in the first $i-1$ rows may become defunct as unrooted modules once the i^{th} row is considered. It is necessary to modify the tree so that it no longer represents these sets as unrooted modules.

Let the *full leaves* denote the leaves that correspond to ones in row i, and let the *empty leaves* denote those that correspond to zeros in row i. If an edge module X becomes defunct in the i^{th} step, then X and $\overline{X}$ each contain both empty leaves and full leaves. Then X corresponds to an edge whose removal separates the PC tree into two trees, each of which has both full and empty leaves. Let us call such an edge a *terminal edge.* Terminal edges must be removed from the tree, since they correspond to defunct edge modules. If M has the circular ones property, these terminal edges form a path (See Figure 32.7). Let us call this path the *terminal path.* The *terminal nodes* are the nodes that lie at the ends of the terminal path. All nodes and edges that must be altered in Step i lie on the terminal path. When there is a unique node of the PC tree that has both full and empty neighbors, we consider it to be a terminal path of length 0; this node assumes the role of both terminal nodes.

Algorithm 32.2 Constructing the PC Tree

The initial PC tree is a P node that is adjacent to all leaves, which allows all $(n-1)!$ circular orderings.

At each row:

- *Find the terminal path, and then perform flips of C nodes and modify the cyclic order of edges incident to P nodes so that all ones lie on one side of the path (see Figure 32.8.)*
- *Split each node on the path into two nodes, one adjacent to the edges to full leaves and one adjacent to the edges to empty leaves.*
- *Delete the edges of the path and replace them with a new C node x whose cyclic order preserves the order of the nodes on this path.*
- *Contract all edges from x to C-node neighbors, and any node that has only two neighbors.*

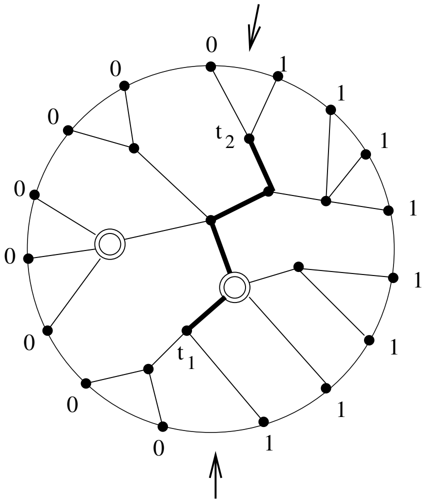


FIGURE 32.7: The edges that must be modified when a new row is added are those that represent two sets that have a mixture of zeros and ones, as these sets fail the criterion for being unrooted modules in the new row. If the matrix has the circular-ones property, these edges lie on a path, called the *terminal path*. The *terminal nodes* are the nodes t_1 and t_2 that lie at the ends of the terminal path.

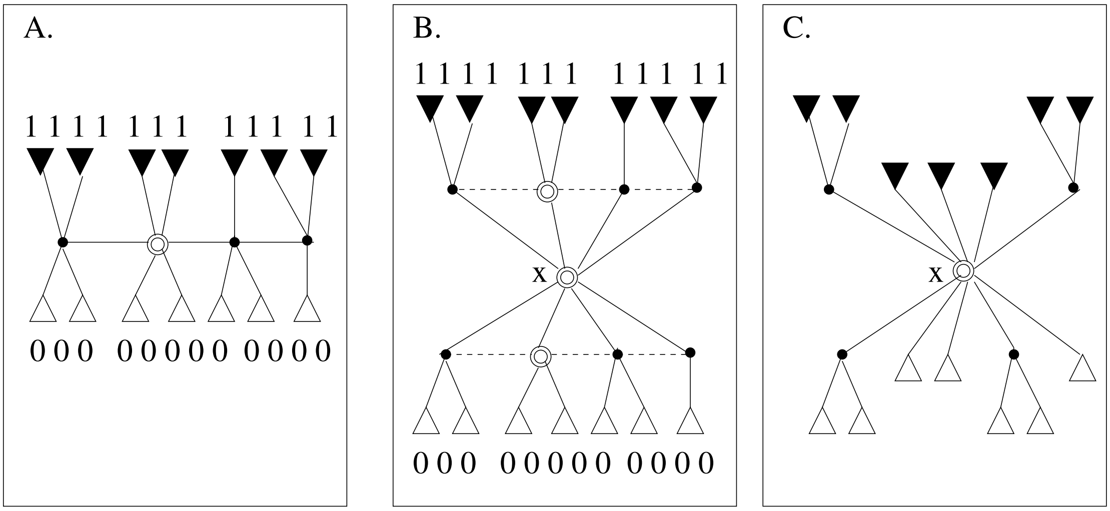


FIGURE 32.8: To update the PC tree when a new row is added to the matrix, flip the C nodes and order the P nodes on the terminal path so that the edges that go to trees whose leaves are zeros in the row lie on one side (white) and those that go to trees whose leaves are ones in the row lie on the other side (black). This is always possible if the new matrix has the circular-ones property (Figure A). Then divide each node on the terminal path into two parts, one that is adjacent to the black trees and one that is adjacent to the white trees (Figure B). Replace the edges of these two paths with a new C node, x, whose cyclic order reflects the order of nodes these two paths. Finally, contract each edge from x to a C-node neighbor, and contract each internal node of degree two (Figure C).

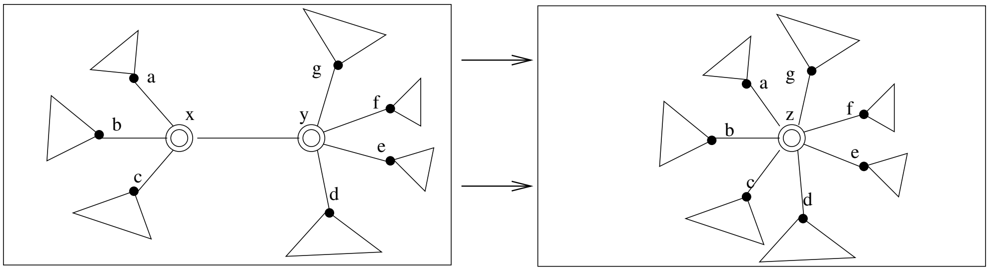


FIGURE 32.9: An order-preserving contraction of an edge xy. The neighbors of x and y are cyclically ordered. The edge is removed and x and y are identified, so that the cyclic order of neighbors of x and y about the edge is preserved.

32.2.1 A Data Structure for Representing the PC Tree

For the implementation, we pair up the $n-1$ edges with $n-1$ of the nodes of the tree, so that each edge is paired with one of the nodes that it is incident to. This can be accomplished by rooting the tree at an arbitrary node in order to define a parent function, and then pairing each non-root node with its parent edge. It is worth noting that in contrast to the rooting of the PC tree, which serves to give a distinguished role to the root, the sole purpose of this is to make low-level operations more efficient. An example of such an operation is the problem of finding out whether two nodes of an unrooted tree are adjacent, which can be determined in $O(1)$ time if it is rooted, by examining the parent pointers of the two nodes.

An undirected graph is a special case of a directed graph where every arc (u, v) has a *twin arc* (v, u). Thus, we may speak of the directed *arcs* of the PC tree, not just its edges.

DEFINITION 32.1 The data structure for the PC tree is the following. Each P node carries a pointer to the parent edge. Each edge uv is implemented with two oppositely directed *twin arcs* (u, v) and (v, u). Each arc (x, y) has a pointer to its two neighbors in the cyclic order about y, a pointer to its twin, and a *parent bit* label that indicates whether y is the parent of x. In addition, if y is a P node, then (x, y) has a pointer to y. There is no explicit representation of a C node; its existence is implicit in the doubly-linked circular list of its incident edges that gives their cyclic order. No two C nodes are adjacent, so each of these edges has one end that identifies a neighbor of the C node, and another end that indicates that the end is incident to a C node, without identifying the C node.

If (x, y) is an arc directed into y and y is a C node, then y is not represented by an explicit record. We can find y's parent edge by cycling through the the records for arcs that are directed into y in either cyclic direction about y, until we reach an arc with the required parent bit. Thus, finding a C node's parent edge is not an $O(1)$ operation.

The data structure makes no distinction between the two directions in which a list can be traversed; this distinction is made only at the time when a traversal is begun. One must keep track of both the current and previous element. To move to the next element, one must retrieve both neighbors of the current element, and select the one that is different from the previous element.

Since we will deal with unrooted trees whose internal nodes can be cyclically ordered, it will be useful to define the cyclic order of edges incident to z after the contraction. An *order-preserving contraction* is the one depicted in Figure 32.9, where the neighbors x and y are each consecutive and preserve their original adjacencies in the circular order of z's neighbors.

Using this data structure, it takes $O(1)$ time to remove or insert a section of a list, given pointers to the endpoints. Since the list draws no distinction between forward and backward, a section of a list can be inserted in either order in $O(1)$ time. It therefore supports an order-preserving contraction of an edge between two adjacent C nodes x and y in $O(1)$ time, given a pointer to (x, y), in addition to allowing insertion or removal of an edge from a node's circular adjacency list or reversal of a section of a circular list in $O(1)$ time

32.2.2 Finding the Terminal Path Efficiently

Finding the terminal path is the only step where pointers to parent edges are required in the i^{th} iteration.

Recall that we have defined full and empty leaves of the PC tree. For the internal nodes, let us say that a node is *full* if it is possible to root the tree so that all leaves in the subtree rooted at the node are full, and *empty* if it is possible to root the tree so that all leaves in its subtree are empty. Since each node has degree at least three, at most one of these designations applies at a node.

We use the following *full-partial labeling algorithm* to mark the full nodes. The efficiency of the algorithm is due to the fact that it avoids touching some of the empty nodes; it leaves them unmarked. We label a leaf as full if it corresponds to a column with a one in the i^{th} row. We label an internal node as full if all of its neighbors except one have been labeled full. We label an internal node as *partial* if at least one of its neighbors has been labeled full. Whenever we label a node as full, we increment a counter in its non-full neighbor x that records how many full neighbors x has, labeling x as full if this counter rises to one less than the degree of x. However, if x is a C node, then since it is given implicitly by a circular list of neighbors, we do not keep an explicit counter at x. Nevertheless, it is easy to detect when all of its neighbors except one is full. Recall that no two C nodes are adjacent.

To perform the labeling in the presence of C nodes, we use an unrooted variant of the pointer borrowing strategy of [5]. We maintain *block-spanning pointers* from the first to last vertex and from the last to the first vertex in each consecutive block of full neighbors around the cycle that makes up x. Each time a new y neighbor of x becomes full, either y becomes a one-element block, a block is extended by appending or prepending y, or two blocks and y merged, by appending y to one of the blocks and prepending it to the other. Each of these operations gives access in $O(1)$ time to the first or last vertex in each affected block, so it is trivial to update the block-spanning pointers in $O(1)$ time. A test of whether the first and last vertices in the resulting block share a non-full neighbor z on the cycle takes $O(1)$ time. If x passes this test, it is full, and the full-neighbor counter of z is incremented.

Since every node of the PC tree has degree at least three, the number of full leaves is at least as great as the number of full internal nodes, and there are at most k full leaves. Assigning full labels takes $O(k)$ time. The number of partial nodes is at most as great as the number of full nodes, since each full node has at most one partial neighbor. Assigning partial labels takes $O(k)$ time also.

Henceforth, let us call a node *full* or *partial* according to the final label assigned to it by the full-partial labeling algorithm.

The key insight for finding the terminal edges is the observation that an edge is terminal if and only if it lies on a path in the tree between two partial nodes.

In the special case where there are no terminal edges and the terminal path has length 0, it is the unique node that has both full and unmarked neighbors. It is easy to see that since there are no other such nodes, its unmarked neighbors are empty. This node is easy to find, given the marking of the nodes.

Otherwise, let the *apex* be the least common ancestor of the partial nodes. We find the terminal edges by starting at each partial node and extending a path up through its ancestors, marking edges on the path. We do this in parallel at all partial nodes, extending the paths at the same rate. When a path runs into another partial node, we stop extending that path. If a path extends above the apex, we may or may not detect this right away. Eventually, we will be extending only one path, at which point, the marked edges form a connected subtree. The apex is the first point below the highest point of this subtree that is either partial or has two paths entering it. Unmarking the edges from the marked edge down to the apex leaves edges marked iff they are terminal edges.

If x is a node on the terminal path other than the apex, then the parent of x is also on the terminal path. If x is a P node, this takes $O(1)$ time, since it has a pointer to its parent.

If x is a partial C node that has a full neighbor, we may assume that we have a pointer to the edge to this neighbor, since this is provided by the full-partial labeling algorithm. This is always the case at a terminal node, which has a full neighbor and an empty neighbor. As we climb the terminal path toward the apex, when reaching a C node y from its child x on the terminal path, we obtain a pointer to the edge (x, y), since x is a P node and (x, y) is its parent edge. We obtain pointer even if y has no full neighbor.

The key to bounding the cost of finding x's parent when x is a C node on the terminal path is the observation that if it has any full neighbors, the full neighbors are consecutive, and the edges to its neighbors on the terminal path are adjacent to the full neighbors in the cyclic order. Thus, if it has no full neighbor, we can look at the two neighbors of the child edge in the cyclic order, and one of them must be the parent edge. This takes $O(1)$ time. If x has a full neighbor, then we can cycle clockwise and counterclockwise through the edges to full neighbors. Of the first edges clockwise and counter-clockwise that are not to full neighbors, one of these is the parent. In this case, the cost of finding the parent is proportional to the number of full neighbors x.

If x does not lie on the terminal path, then this procedure may fail, in which case we detect that it is not on the terminal path, or it may succeed, in which case we can bound the cost of finding the parent in the same way.

If the union of all paths traversed has p' nodes and there are k ones in row i, then the total cost is $O(p' + k)$. However, the number of nodes in these paths that are not on the terminal path is at most the number of nodes that are on the terminal path, because of the way the paths are extended in parallel. The following summarizes these observations.

LEMMA 32.1 If the terminal edges form a path and the full and empty neighbors can be flipped to opposite sides of it, then finding the terminal path in the i^{th} step takes $O(p + k)$ time, where p is the length of the terminal path and k is the number of full nodes.

If the conditions of the lemma are not satisfied, the matrix does not have the circular-ones property, and is rejected.

32.2.3 Performing the Update Step on the Terminal Path Efficiently

The number of full neighbors of nodes on the terminal path is bounded by the number k of ones in the i^{th} row. Before splitting a node, we record its neighbors on the terminal path, and then delete the edges to these neighbors, in $O(1)$ time. We then split the node by splicing out the full neighbors, and forming a new node with them. The remainder of the old node serves as the other half of the split. This takes time proportional to the number of full neighbors. Since the number of full neighbors of nodes on the terminal path is bounded by the number of ones in the i^{th} row, the total time for this is $O(p+k)$. Creating the new C node x and installing edges to the split nodes in the required cyclic order then takes $O(p)$ time.

Since our data structure includes a parent function, we must assign the parent bits to the new edges. Let y be the copy of the apex that retains its parent edge after the split of the apex. Except in the case of the apex, the parent edge of every vertex on the terminal path is an edge of the terminal path, and these edges have been deleted. Thus, none of these nodes have a parent edge. We make x the new parent of these nodes, and we let y be the parent of x. This takes $O(p)$ time by setting the appropriate bits in the $O(p)$ edges incident to x.

The operation of deleting a C node z from x's neighborhood and replacing it with the neighbors of z is just a contraction of the edge xz, depicted in Figure 32.9, which takes $O(1)$ time. These observations can be summarized as follows:

LEMMA 32.2 Updating the tree during the i^{th} step takes $O(p+k)$ time, where p is the length of the terminal path and k is the number of full nodes.

32.2.4 The Linear Time Bound

Assume that the matrix is given in sparse form, where, for each row, the set of columns where the row has a one is listed. Let us assume that every row and every column has at least one nonzero entry, since it can otherwise be eliminated in a preprocessing step. A linear time bound is one that is proportional to the number of nonzero entries in the matrix.

The algorithm processes one row at a time of the matrix M. Let T denote the current state of the PC tree after the first i rows have been processed. That is, T is the PC tree for the submatrix induced by the first i rows. Let C_i be the set of C nodes and let P_i be the set of P_i nodes in T, and let u_i be the number of ones in the rows of the matrix that have not yet been processed. If x is a node of T, let $deg(x)$ denote the *degree*, or number of neighbors of x in T.

If, when processing a row, we could update the PC tree in time proportional to the number of ones in the row, the linear time bound would be immediate. Unfortunately, the update step does not conform to this bound. Therefore, we use a technique called *amortized analysis* [9], which uses an accounting scheme whereby updates that exceed this bound borrow credits from updates that were completed with time to spare. The analysis shows that, even though there is variability in the time required by the updates, the aggregate cost of all updates is linear.

We adopt a budget where we keep an account that must have a number of credits $\phi(M,i)$ in reserve, where $\phi(N,i)$ is given by the following:

- $\phi(M,i) = 2u_i + |C_i| + \sum_{x \in P_i}(deg(x) - 1)$.

Such a function is often called a *potential function*. Each credit can pay an $O(1)$ operation. Any operation that reduces ϕ allows us to withdraw credits from the account to pay for some of the operations. Since $\phi(M,0)$ is $\Theta(m)$, we must pre-pay the cost of later operations by making a deposit of $\Theta(m)$ credits to the account. If we can maintain this reserve as we progress and still pay for all operations with withdrawals, then, since ϕ never runs the account to zero, the running time of the algorithm is $O(m)$.

To cover the costs in row i, we must be able to withdraw $k+p$ credits, by Lemmas 32.1 and 32.2, where k is the number of full nodes. Since every full node has at least two full neighbors, the number of marked nodes is at most two times the number of 1's in row i, so k credits are freed up by the decrease from $2u_i$ to $2u_{i+1}$.

Each C node on the terminal path is first split, but then both of these copies are contracted. This decreases the number of C nodes by one without changing the degrees of any P nodes, so spending a credit for each C node on the path is within the budget. Suppose a P node does not lie at the end of the terminal path. If it is split, the sum of degrees of the two parts is the same as the degree of the original, after the terminal-path edges are deleted and replaced with an edge to the new C node. However, in calculating ϕ, subtracting one from the degrees of two P nodes instead of one frees a credit. If the P node has only empty neighbors or only full neighbors, it is not split. In this case its degree decreases by one when its two incident terminal-path edges are deleted and replaced by a single edge to the new C node. A P node at the end of the terminal path fails to free up a credit, but there are only $O(1)$ of these. Contractions of nodes of degree two free up a credit, whether they are C nodes or P nodes. $\Theta(k+p)$ credits for row i can be paid out while adhering to the budget.

32.3 Planar Graphs

In this section, we describe an algorithm due to Shih and Hsu [34] that uses PC trees to recognize whether a graph H is planar. If it is, the algorithm returns a planar embedding, and if it is not, it points out a subgraph that is a subdivision of a $K_{3,3}$ or a K_5.

Linear time planarity test was first established by Hopcroft and Tarjan [13] based on a *path addition approach*, which finds a path in the graph and uses it to break the problem down recursively. A *vertex addition approach*, originally developed by Lempel, Even and Cederbaum [24], was later improved by Booth and Lueker [5] (hereafter, referred to as the *B&L algorithm*) to run in linear time using PQ trees. This approach adds one vertex at a time, updating the PQ tree to keep track of possible embeddings of the subgraph induced by vertices added so far. Both of these approaches are quite complex. Furthermore, both approaches use separate algorithms for recognition and embedding (Chiba et al [8]). Several other approaches have also been developed for simplifying the planarity test (see for example [7, 31, 35, 36]) and the embedding algorithm [6, 30]. Shih and Hsu [33] developed a linear time test, which has been referred to as the simplest linear time planarity test by Thomas in his lecture notes [37]. Independently, Boyer and Myrvold discovered a similar algorithm to the PC tree approach [6]. Later, in [34], they implemented the algorithm based on PC trees, which will be referred to as the *S&H algorithm*. When the given graph is not planar, the algorithm immediately produces explicit Kuratowski subgraphs. Furthermore, the recognition and embedding are done simultaneously in the algorithm.

32.3.1 Preliminaries

To represent a planar embedding, it suffices to find, for each vertex, the clockwise circular ordering induced by the planar embedding on its incident edges. Given these circular orderings, there are algorithms that can assign spatial coordinates to the nodes. Here, we deal only with the problem of finding these circular orderings, and refer to them collectively as the *embedding* of the graph.

As the algorithm progresses, more of the embedding becomes known. In particular, S&H comes to know the cyclic order of edges incident to a subset of the vertices. When the cyclic order of a vertex is known, it does not know whether this order should be clockwise or counterclockwise in the embedding, so it uses a C node to represent its cyclic order.

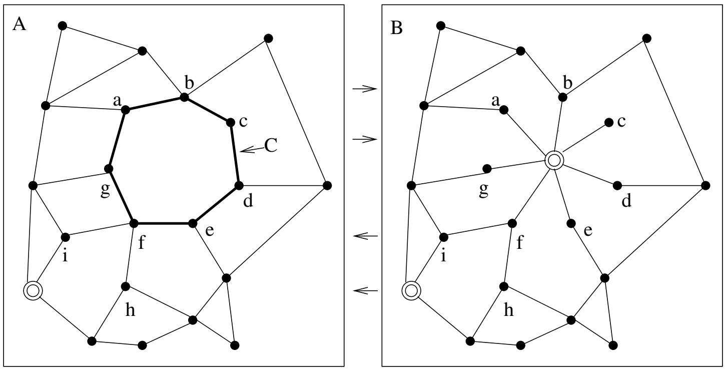


FIGURE 32.10: A *cycle replacement* consists of selecting a cycle in a graph, adding a new C node whose neighbors ordering gives the ordering of nodes on the cycle, and deleting the edges of the cycle. As we illustrate in parts C and D of Figure 32.11, below, it is applied to a graph that arises partway into the induction step. It's inverse operation is a `C-node replacement`.

Collectively, the C nodes represent the partial embedding known so far. Let us call such a graph with C nodes a *constrained graph.* The data structure for implementing the C node is the same as the one given in Definition 32.1; the algorithm continues to ensure that no two C nodes are adjacent.

If T is a depth-first spanning tree (DFS tree) of an undirected graph G, all edges of G are *tree edges* (edges of T) or *back edges* (edges between a descendant and an ancestor in T [9]. A vertex is a *back vertex* if it has an incident back edge from one of its descendants. Since there are $n-1$ edges in T, the number of back edges is $m-n+1$. We may therefore refer to the number of back edges without reference to any particular DFS tree.

A *cut set* is a set of vertices of a connected graph whose removal from the graph disconnects it. An *articulation vertex* in a graph is a vertex whose deletion disconnects the graph. A graph is *biconnected* if it has no articulation vertex [9]. A *biconnected component* of a graph is a maximal biconnected subgraph. The articulation vertices can be found in linear time [9], so it suffices to embed each biconnected component separately, and then connect them by their adjoining articulation vertices. Henceforth, therefore, we may assume that G is biconnected.

Given a planar embedding of a constrained graph G, and a cycle C in G that is a cut set, let us say that C's *subembedding* is C and everything internal to it in the embedding.

A *C-node replacement* is the following operation (See Figure 32.10): If $(y_1, y_2, ..., y_k, y_1)$ is the cyclic order of neighbors of x, install an edge $y_i y_{(i+1) mod k}$ for each i from 1 to k, then delete x. This replaces x with the cycle $(y_1, y_2, ..., y_k, y_1)$. A *cycle replacement* is the inverse of this operation: select a cycle C, and insert a new C node x whose cyclic ordering of neighbors gives the vertices of C in order, and then delete the edges of C.

32.3.2 The Strategy

The algorithm of S&H can be described as a recursive algorithm, `Embed`. The graph passed to the initial call has no C nodes, but graphs passed to lower calls will have them. A DFS spanning tree is also passed to the call, and since the DFS tree may contain C nodes, it is a PC tree.

For now, assume that the initial graph is planar; later, we show how to modify the

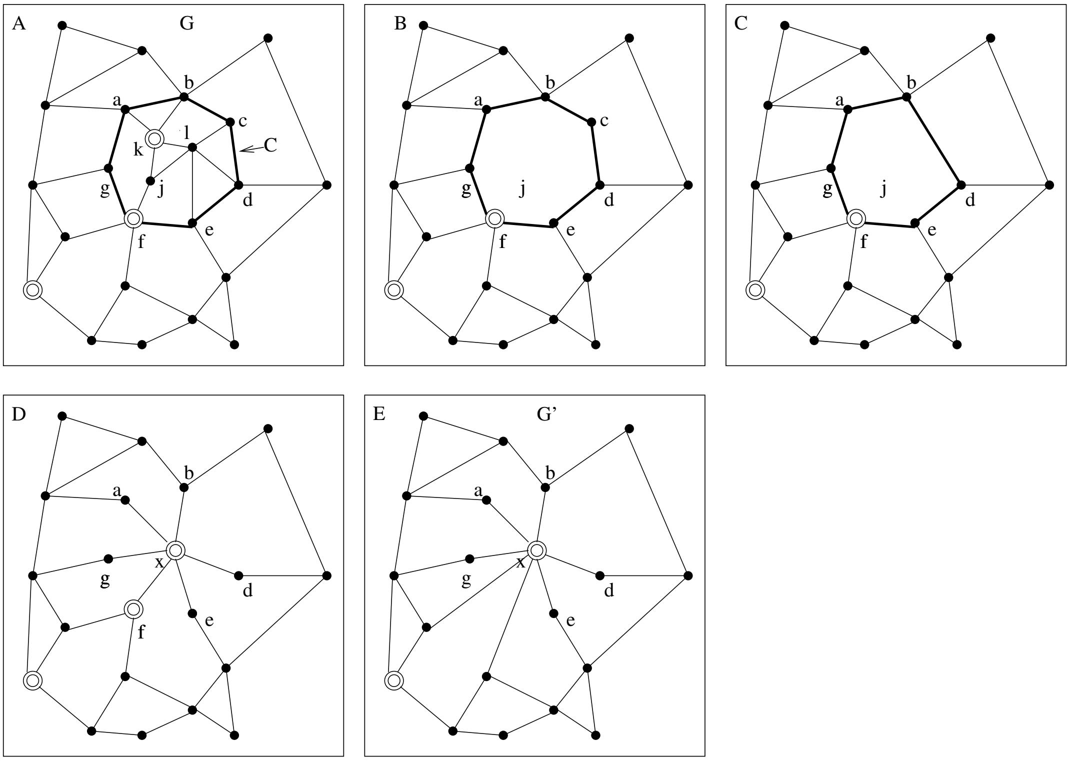


FIGURE 32.11: The `Embed` operation. `Embed` finds a cycle C and its subembedding A in an unknown planar embedding of G. `Embed` removes elements of A that are internal to C, contracting resulting vertices of degree 2 on C, and performs a cycle replacement on C, inserting a new C node x. It then performs contractions to eliminate C-node neighbors of x. The result is the graph G'. By induction on $m - n + 1$, a recursive call can be used to find an embedding of G'. Performing a C-node replacement of x gives a planar embedding where C is a face; inverting the foregoing operations inserts the planar embedding of A inside of it.

algorithm so that it returns a Kuratowski subgraph when this is not the case. The input graph to each recursive call is also biconnected. `Embed` works by induction on the number $m - n + 1$ of back edges (see Figure 32.11):

1. Choose a cycle C that is a cut set and return its subembedding A in some unknown planar embedding of G (Figure 32.11, part A).
2. Remove elements internal to A to obtain graph G_2, which has a planar embedding where C is a face (Figure 32.11, part B).
3. Contract nodes on C that now have degree 2 to obtain a cycle C' (Figure 32.11, part C).
4. Perform a cycle replacement on C' to obtain a constrained graph G_3 (Figure 32.11, part D).
5. Perform edge contractions between adjacent C nodes in G_3 to obtain a a biconnected constrained graph G' where no two C nodes are adjacent (Figure 32.11, part E).
6. By induction on the number $m{-}n{+}1$ of back edges, we may call `Embed` recursively

on G' to obtain a planar embedding of it.

7. On the planar embedding of G', perform the inverses of steps 6, 5, 4 and 3 to obtain a planar embedding of G.

The reason for contracting nodes on C of degree two in Step 3 is to ensure that, like G, G' is biconnected. Failure to do so would result in pendant nodes, such as node c in Figure 32.10.

Because of the way C is selected, the base case will be a biconnected constrained graph G with a vertex n such that $G - n$ is a PC tree. $G - n$ is trivial to embed, and since this embedding has only one face, it is trivial to add n and its incident edges to this embedding to obtain an embedding of G.

32.3.3 Implementing the Recursive Step

Let us name the vertices according to their postorder numbering on a DFS tree T of G. If i is a vertex, we let T_i denote the subtree of T rooted at i. Let us say that a vertex j is *earlier* than vertex i if $j < i$.

The following are the inputs to `Embed`.

I1: A biconnected constrained graph G.

I2: The earliest back vertex i in T;

I3: A DFS tree T of G where all C nodes in the tree are earlier than i. The DFS tree is implemented as in Definition 32.1, except that the circular lists of edges incident to a C node can include non-tree edges. The parent bits of Definition 32.1 are required only on tree edges, and are consistent with the rooting of the DFS tree.

I4: An ordered list of i and all later vertices, ordered in postorder on the DFS tree.

The Terminal Path

Let i be the earliest back vertex, let r be a child of i in the DFS tree whose subtree T_r in the DFS tree has a back edge to i. Since r is earlier than i, T_r is an induced subgraph of the constrained graph G that has no back edges, hence it is a tree.

A trivial case occurs when i is the root n of the DFS tree. Since G is biconnected, n is not an articulation vertex, so T_r is unique, and $T_r = G - n$. T_r is trivial to embed, and since this embedding has only one face, it is also trivial to add n and its incident edges to this embedding. This is the base case referred to in Section 32.3.2.

Otherwise, $i < n$. For ease of presentation, let us imagine, but not explicitly create, an *unrooted* tree T'_r as follows. For each edge (x, j) from a node x of T_r to a node $j \geq i$, add an edge (x, j_x) to T_r. Note that this applies to the tree edge (r, i), yielding (r, i_r). The result is a PC tree, but there may be multiple copies of each back vertex j, one for each edge from a node of T_r to j.

By analogy to Section 32.2.2, let us consider a leaf j_x of T'_r to be *full* if $j = i$ and *empty* otherwise. Since G is biconnected, every leaf of T_r has a neighbor greater than or equal to i; otherwise, its neighbor in T_r would be an articulation vertex. Therefore, every node of T_r is an internal node in T'_r. In addition, since $i < n$ and i is not an articulation vertex, T_r has edges both to i and to proper ancestors of i, so T'_r has both empty and full leaves. Finally, since i has an incident tree edge and an incident back edge from T_r, T'_r has at least two full leaves.

As in Section 32.2.2, let us consider an internal node x to be *full* if there is a rooting of T'_r where x's subtree only has full leaves, and *empty* if there exists a rooting where its subtree

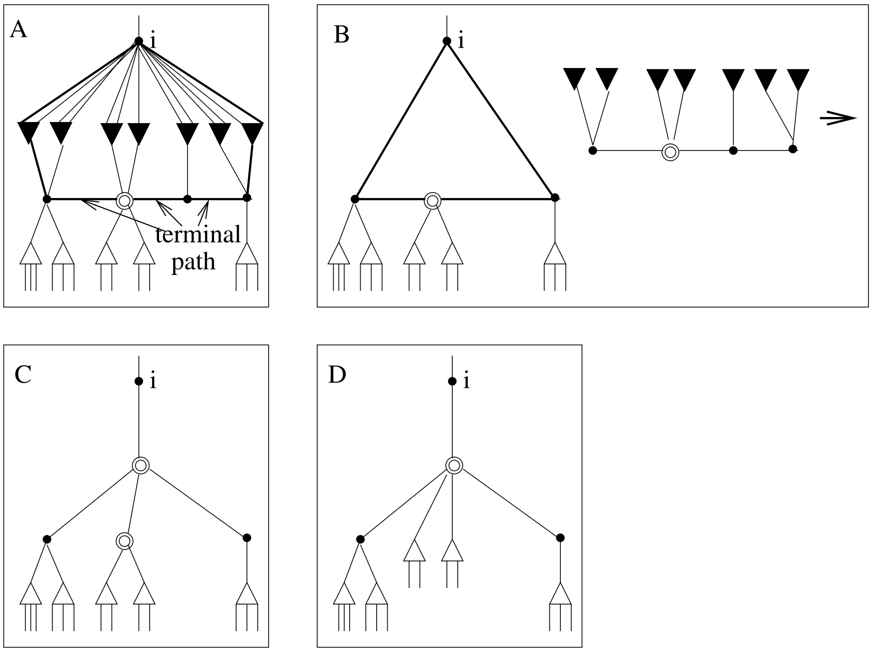


FIGURE 32.12: The induction step of `Embed`. The cycle C selected by `Embed` consists of i, the terminal path, and the leftmost and rightmost paths to i from the terminal nodes after full and empty subtrees have been flipped to opposite sides of the terminal path (A). The full nodes and their back edges are trivial to embed inside C, since all of their back edges go to i. The nodes internal to C are to be removed. If they are removed, however, the nodes on the paths from i to the terminal nodes will have degree 2 so they can contracted out of the cycle. The net effect is to remove all full nodes from G, and leave a cycle consisting of i and the terminal path. This is accomplished by splitting the terminal path, as in the consecutive-ones problem, removing the full side of the split, and inserting an edge from i to each terminal node (B). A cycle replacement is performed on this cycle (C), and, as in the consecutive-ones problem, an order-preserving contraction is performed to remove C-node neighbors of of the new C node (D). This yields G'; performing a recursive call on G' and inverting the steps from A to D on the resulting embedding gives a planar embedding of G.

only has empty leaves. It is a *terminal edge* if neither of its endpoints is full or empty. If the terminal edges form a path, this is the *terminal path.* If there are terminal edges but they do not form a path, then T'_r has no terminal path. As before, if there are no terminal edges, the terminal path has length 0 and consists of a single node.

We give a proof of the following in Section 32.3.5:

LEMMA 32.3 If the constrained graph G has a planar embedding, it has a terminal path, and nodes on this path can be flipped so that all full subtrees lie on one side and all empty subtrees lie on the other, without violating the cyclic order of any C node.

This gives a planar embedding of T'_r in which all copies of i can be joined without crossing any back edges, as illustrated in Figure 32.12 (A). By the definition of T_r, there must be a back edge to i, and since G is biconnected, there must be a back edge from T_r to a proper ancestor of i; otherwise i would be an articulation vertex, since we are in an induction step where $i \neq n$. The full subtrees, the terminal path, and i form an induced subgraph with a bounding cycle in this embedding. This bounding cycle is the cycle C referred to in the overview of the algorithm in Section 32.3.2.

As described in the overview, only the cycle, minus its nodes of degree two, is retained for the recursive call (Figure 32.12, part B), and it is replaced by a C node (Figure 32.12, part C). This results in adjacent C nodes whenever there is a C node on the terminal path, which is remedied with an $O(1)$-time contraction is performed on them, as depicted in Figure 32.9, with a result illustrated in Figure 32.12, part D.

A recursive call on the resulting graph provides an embedding of it. Inverting the operations depicted in parts D through B of Figure 32.12 yields a face into which the already-known embedding of the full subtrees can be inserted to yield a planar embedding of the original constrained graph.

Finding the Terminal Path

We use the full-partial labeling algorithm of Section 32.2.2 to label the full internal nodes of T'_r. This does not require creating T'_r explicitly, since T_r and its back edges represent T'_r implicitly, and the full-partial labeling algorithm can be run on this representation by considering i to be full and its ancestors to be empty.

However, we must confront an annoying detail that we didn't have in Section 32.2.2, which is that internal nodes can have degree 2. This allows the possibility that an internal node can be both empty and full. This can happen as follows. Let x be a full node, let y be an empty neighbor of degree 2, and let z be y's other neighbor. Rooting T'_r at z gives y a subtree whose leaves are all full, and rooting it at x gives y a subtree whose leaves are all empty. Therefore, y is *ambiguous.*

If it is run without modification, the full-partial labeling algorithm of Section 32.2.2 will label ambiguous nodes as full. However, we can detect the first time it labels an ambiguous node y. In this case, x is a full neighbor that has just notified y that it is full. If x has degree 2, then it is also ambiguous, contradicting our choice of y. If it has degree 1, it is the only full leaf, contradicting the fact that T'_r has at least two full leaves. Therefore, x has degree greater than two, and full leaves can be reached from y only by going through x.

This situation is easily detected: when it is time for x to notify y that it has become full, x is the only node that has been labeled full but not yet notified its non-full neighbor, and y has degree 2. In this case, we halt the full-partial labeling algorithm early, and select x to be the only node in a terminal path of length 0. Aside from this detail, the full-label algorithm is the same as in Section 32.2.2.

By the analysis of Lemma 32.1, the running time is proportional to the number of terminal edges plus the number of full nodes if the conditions of Lemma 32.3 are met. If they are not met, the graph can be rejected. However, in this case, we still want to know the terminal edges since we will use them to produce a Kuratowski subgraph. The algorithm for finding them is easily implemented to run in time proportional to the size of G in this case, as it requires finding only the subtree of edges that lie on paths between partial nodes. They can be labeled by rooting the tree at one of the partial nodes and working upward from the other partial nodes.

The Linear Time Bound

The analysis of the complexity of the full-partial algorithm of Section 32.2.2 made use of the fact that there are no nodes of degree two. We have not ensured that this is true of T'_r. The main problem that this causes is that the number of full nodes is not asymptotically bounded by the number of full leaves, so we can no longer claim that the running time of the full-partial labeling algorithm is bounded by the number of full leaves.

Instead, S&H makes use of the observation that all full nodes are deleted from the recursive call. We may therefore use a potential function that charges the costs of the full-partial labeling algorithm to nodes and edges that are removed from the recursive call, at $O(1)$ cost per node or edge.

The potential function is similar to the one for the consecutive-ones property, except that u_i denotes the number of nodes and edges of the graph:

$$\phi(M, i) = 2u_i + |C_i| + \sum_{x \in P_i} (deg(x) - 1)$$

We show that the value of the potential function for the recursive call drops by an amount proportional to the set of operations performed outside of the recursive call, such as running the full-partial labeling algorithm.

The analysis of the cost of the $O(p)$ operations is unaffected. Since the full nodes are deleted, the $\Omega(k)$ drop in the potential function pays for the remaining $O(k)$ operations. The remaining operations of finding the terminal path are analyzed as in the consecutive-ones problem.

Finding and splitting the terminal path is performed just as it is in the consecutive-ones problem. By Lemmas 32.1 and 32.2, this takes time proportional to the number k of full nodes plus the length p of the terminal path, and covers the cost of removing the nodes internal to C illustrated in Figure 32.12 (B).

We must now analyze the cost of meeting the preconditions I1-I4 in each recursive call. I1 through I4 can easily be met in $O(n + m)$ time for the initial call, where all nodes are P nodes, using standard techniques from [9].

Given I1 - I4 for G, we describe how to modify them so that they can be met in the recursive call on G'.

LEMMA 32.4 A rooted spanning tree of an undirected graph is a DFS tree if and only if all non-tree edges are back edges.

The necessity of this condition is common knowledge [9]. For the sufficiency, observe that if T is a rooted spanning tree such that every non-tree edge is a back edge, then ordering the adjacency lists so that tree edges appear first and calling DFS at the root of T will force it to adhere to T as the DFS tree.

For Input I4, let T' be the DFS tree passed to the recursive call. Since all differences between T and T' occur in subtree T_i, the postorder of elements later than i in T is also the same as in T'. The input is met by searching forward in the preorder list for the next back vertex, and discarding the traversed elements from the front of the list.

For Input I3, let us make the new C node x inserted inside C be a child of i, hence a parent of its other neighbors in the DFS tree that is passed to the recursive calls. This requires us to label the parent-bit labels of edges incident to the new C node before contractions, as in Figure 32.12 (C). We have already bounded the cost of touching these edges, so this does not affect the asymptotic running time. Any remaining C nodes that are removed by $O(1)$-time contractions, as in Figure 32.12 (D) have the contracted edge as their parent edge, so the new node becomes the parent of their empty subtrees, without requiring any further relabeling of parent bits.

Since all back edges in T go from descendants to ancestors, it is easy to see that this is also true of T'. T' is a depth-first spanning tree of G', by Lemma 32.4, so the conditions of

I3 are satisfied by T'.

For Input I1, Step 3 of the description of Section 32.3.2 ensures that every node on the cycle has an incident edge to an empty node in T'_r. Therefore, the remaining graph is biconnected, since for every node in T_r, there are two paths to a proper ancestor of i: one of them by traveling upward on tree edges, and one of them by traveling downward through an empty tree and then to the ancestor on a back edge.

For Input I2, if i ceases to be a back vertex in G', then vertices are examined in ascending order in the list I4, starting at i until the next back vertex, i', is encountered. Since these searches are monotonically increasing, the total costs of them over all recursive calls is linear.

32.3.4 Differences between the Original PQ-Tree and the New PC-Tree Approaches

Whenever a biconnected subgraph is created, the algorithm uses a subset of vertices in its bounding cycle as representatives to be used for future embedding. The embedding of each biconnected component is temporarily stored so that, at base case, when the graph is declared planar, a final embedding can be constructed by tracing back and pasting the internal embedding of each biconnected component inside its bounding cycle.

The way S&H adopted PC trees in their planarity test is entirely different from that of B&L's application of PQ trees in Lempel, Even and Cederbaum's planarity test [24]. B&L used PQ trees to test the consecutive ones property of all nodes adjacent to the incoming node in their vertex addition algorithm. The leaves of their PQ trees are exactly those nodes adjacent to the incoming node. Internal nodes of the PQ trees are not the original nodes of the graph. They are there only to keep track of feasible permutations, whereas in S&H's approach, a P node is also a vertex of the original graph H, a C node denotes a biconnected subgraph, and nodes adjacent to the new node can be scattered anywhere, both as internal nodes and as leaves in the PC tree. Thus, S&H's PC tree faithfully represents a partial planar embedding of the given graph and is a more natural representation. Another difference is that in order to apply PQ trees in Lempel, Even, and Cederbaum's approach, there is a preprocessing step of computing the "s-t numbering" besides the depth-first search tree. This step could create a problem when one tries to apply PQ trees to find maximal planar subgraphs of an arbitrary graph [13].

Other aspects of handling the PC tree are adapted from B&L's approach, such as the handling of of Q nodes (or C nodes) during execution of the full-partial labeling algorithm.

32.3.5 Returning a Kuratowski Subgraph when G is Non-Planar

In this section, let H denote the unconstrained graph that is passed into the initial call, let G denote the constrained graph that is passed into a lower recursive call, and let G' denote the constrained graph that is passed into the next recursive call down from G.

The graph H passed in at the highest-level call has no C nodes. In any recursive call on a constrained graph G, all P nodes are vertices of H. Therefore, all neighbors of a C node are vertices of H.

LEMMA 32.5 A unique cycle C of H that has the following properties can be constructed from a C node x of G. Let $(x_1, x_2, ..., x_k)$ be the cyclic order of x's neighbors. Then $(x_1, x_2, ..., x_k)$ appear in that order on C, and the remaining nodes of C were contracted in higher-level recursive calls by applications of Step 3 of Section 32.3.2.

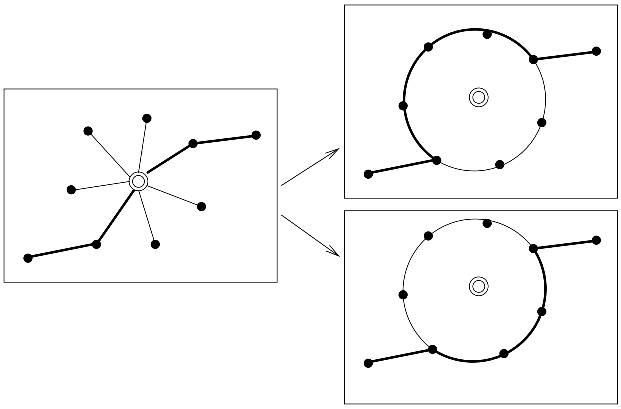


FIGURE 32.13: A C node x of G represents a cycle in H, so a path of G through x represents two possible paths in H.

Before proving this, let us examine what it implies about the relationship paths G and paths in H. A path in G through a C node x corresponds to two possible paths in H, one that proceeds in one direction about the cycle represented by x and one that proceeds in the other direction (Figure 32.13).

Proof We give the construction of the lemma by induction on the depth of a call. The initial call is the base case, where there are no C nodes and the claim is vacuously true. For the induction step, let x be the new C node created in the step, and let C be the separating cycle bounding the full nodes of G that is found in the step. Note that, as in Figure 32.12, C may itself contain C nodes. Let y be such a C node on C, and let a, b be its neighbors on C. By induction, the lemma applies to y, so y represents a cycle C_y of H. The portion (a, y, b) of C represents two possible paths in H: one, P_1, that travels one way around C_y avoiding empty neighbors of y in G (other than possibly a or b), and another, P_2, that travels the other way, avoiding full neighbors of y in G (other than possibly a or b). When constructing the cycle C_x in H represented by x, splice in P_2 in place of (a, y, b). This ensures that the neighbors of x in G' will be on C_x in H.

After application of the contraction step 3, some additional nodes of the cycle bounding the full subtrees are contracted out before they have a chance to become neighbors of the new C node, but these are those that the lemma allows to be removed. Therefore, the induction hypothesis to apply x and C_x.

The constructive proof of the lemma shows how the cycle represented by a C node can be found in H, by undoing the contractions while returning up through the recursive calls to H.

We now show how to return a subdivision of a $K_{3,3}$ or a K_5 when the conditions of Lemma 32.3 are not met.

Suppose first that the terminal edges of T'_r do not form a path. Recall that a terminal edge is defined to be an edge whose removal from T'_r separates it into two subtrees that each have both full and empty nodes. Clearly, the terminal edges are connected, so they form a subtree of T'_r with at least three leaves, z_1, z_2, and z_3. Let w be the meeting point of the paths of terminal edges that connect them, as illustrated in Figure 32.14. For $z_k \in \{z_1, z_2, z_3\}$, there is a path of non-terminal edges through full nodes, ending at a copy

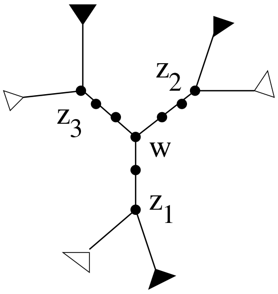


FIGURE 32.14: When the terminal edges of T_r' do not form a path, they form a subtree of T_C with at least three leaves, z_1, z_2, and z_3, each of which is adjacent to a full subtree that has a copy of i and an empty subtree that has a copy of a proper ancestor of i. Let w be the node of the tree from which paths to z_1, z_2, and z_3 branch.

of i, and a disjoint path of non-terminal edges through empty nodes, ending at a copy of a proper ancestor t_k of i. Collectively, these paths correspond to paths in G that are disjoint, except at their endpoints. (The multiple copies of i are identified in G, and $\{t_1, t_2, t_3\}$ are not necessarily distinct.) There exists $t \in \{t_1, t_2, t_3\}$ of median height. The paths from z_1, z_2, and z_3 to t_1, t_2, and t_3 can be extended via edges of the DFS tree to paths to t that are disjoint except at t (Figure 32.15). These paths and the terminal edges joining them to w define a subdivision of a $K_{3,3}$ of G with bipartition $\{\{z_1, z_2, z_3\}, \{w, i, t\}\}$.

If w is a P node, this $K_{3,3}$ of G expands to to a subdivision of $K_{3,3}$ in H by Lemma 32.5. If w is a C node, but at least one of z_1, z_2, and z_3 fails to be a neighbor of w, then we can reduce this case to the previous one by taking into account that w represents a cycle in H by Lemma 32.5, and finding a P-node neighbor w' of w to serve in place of w, as illustrated in Figure 32.16. (Since no two C nodes are adjacent, w' is a P node.)

Suppose w is a C node and each of z_1, z_2, and z_3 is a neighbor of w. Without loss of generality, suppose t_2 is a minimal element of the not-necessarily distinct elements t_1, t_2, and t_3. If $t_2 = t_1$ or $t_2 = t_3$, then S&H returns a K_5; otherwise, the algorithm returns a $K_{3,3}$, as illustrated in Figure 32.17.

Finally, let us consider the case when there are at most two terminal nodes, but it is not possible to flip the full subtrees to one side of the terminal path and the empty trees to the other, due to constraints imposed some C node x that lies on the terminal path.

For each neighbor y of x, let T_y be the neighboring subtree reachable from x through y. That is, it is the component of the PC tree that contains y when x is deleted, and, conceptually, its "leaves" are the leaves of this subtree if it is then rooted at y. If y lies on the terminal path, at least one of T_y's leaves is a copy of i and at least one is a copy of a proper ancestor of y. Otherwise, all of its leaves are copies of i or all are copies of proper ancestors, depending on whether T_y is full or empty.

The cyclic order of neighbors of x blocks flipping the full and empty subtrees to opposite sides of the terminal path iff x has four neighbors a, b, c, d whose cyclic order about x is (a, b, c, d), and and where T_a and T_c each contain a full leaf and T_b and T_d each contain an empty leaf. (A neighbor on the terminal path can be selected for either of these two categories.)

Suppose that this is the case. In T_r', there are disjoint paths from a and c through T_a and T_c to copies of i and from b and d through T_b and T_d to copies of proper ancestors t_b, t_d of i. These all correspond to paths of G that are disjoint except at their endpoints. If $t_b = t_d$, let $t = t_b = t_d$. Otherwise, let t be the lower of the two. The path to the other of the two can be extended by DFS tree edges to t that is still disjoint from the other paths, except

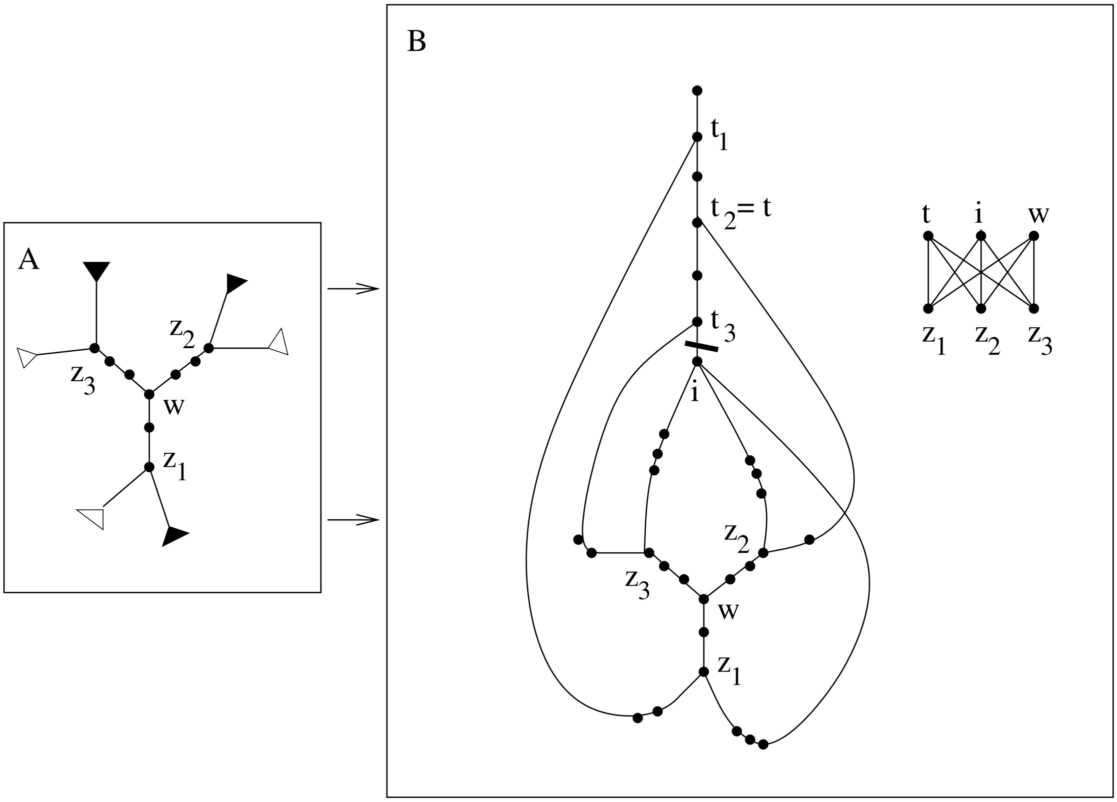


FIGURE 32.15: If w is a P node, S&H gets a $K_{3,3}$ with bipartition $\{\{z_1, z_2, z_3\}, \{w, i, t\}\}$. This $K_{3,3}$ is the contraction of a $K_{3,3}$ of the original input graph H.

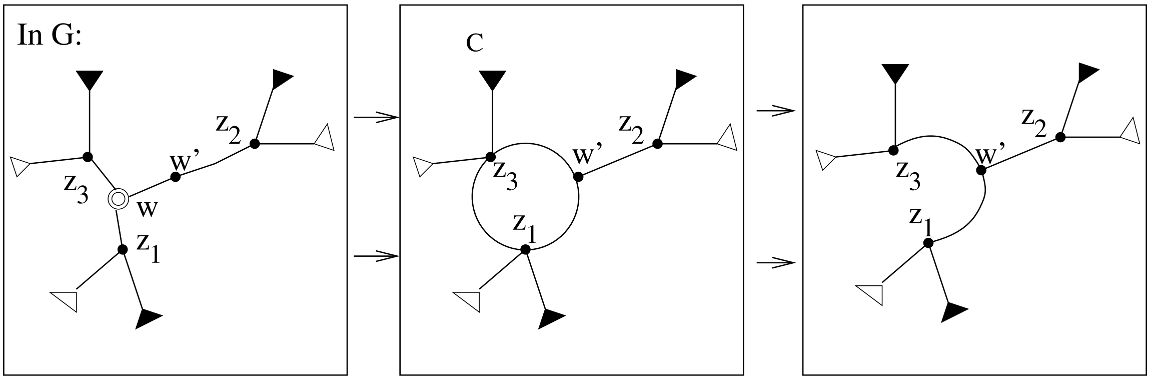


FIGURE 32.16: If w is a C node but at least one of z_1, z_2, and z_3 is a non-neighbor of w, the case can be reduced to that of Figure 32.15 by selecting a neighbor w' to serve in the role of w.

at t. These paths, the cycle represented by x, and the DFS tree edges between t and i give rise to a $K_{3,3}$ of H by Lemma 32.5, as in Figure 32.18. In addition to giving an algorithm for returning a subdivision of a $K_{3,3}$ or a K_5 when the embedding algorithm fails, we have just proven Lemma 32.3, since no planar graph contains such a subdivision.

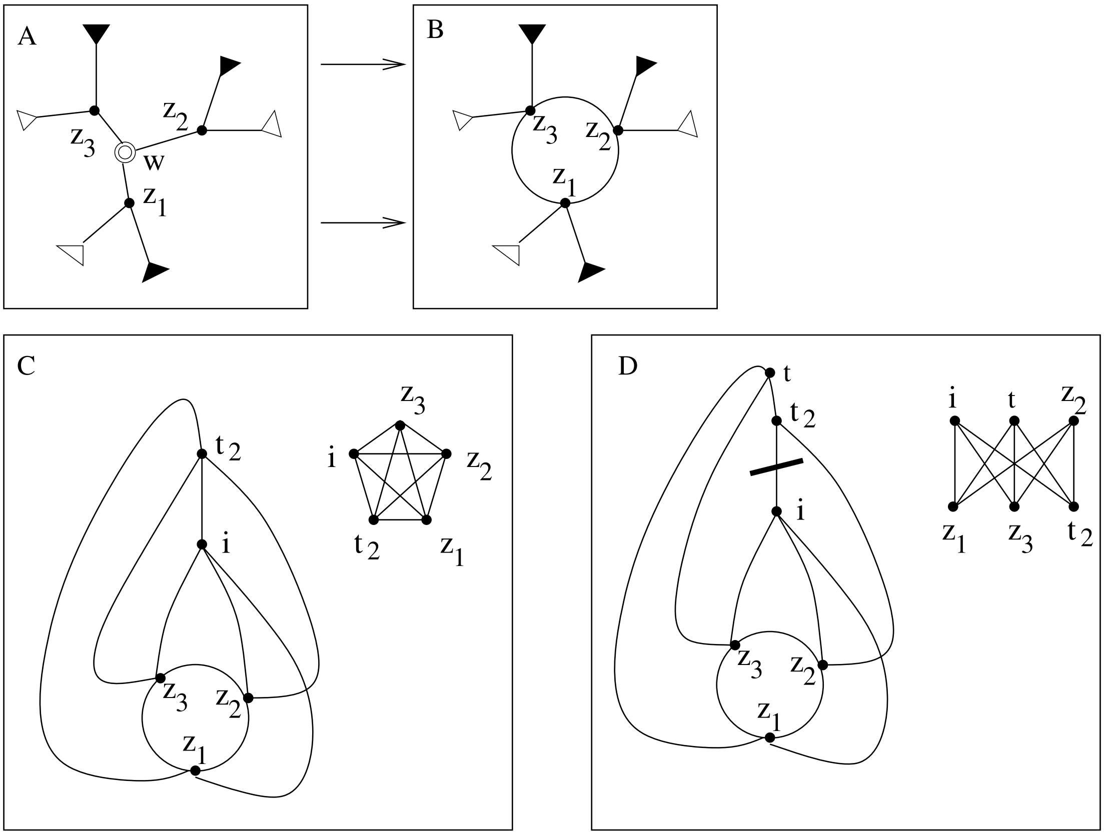


FIGURE 32.17: Each of z_1, z_2 and z_3 is a neighbor of w, which is a C node. Without loss of generality, suppose t_2 is a minimal element of t_1, t_2, and t_3. If one of the others, say, t_3, is equal to t_2, then the algorithm returns a K_5. Otherwise, it returns a $K_{3,3}$.

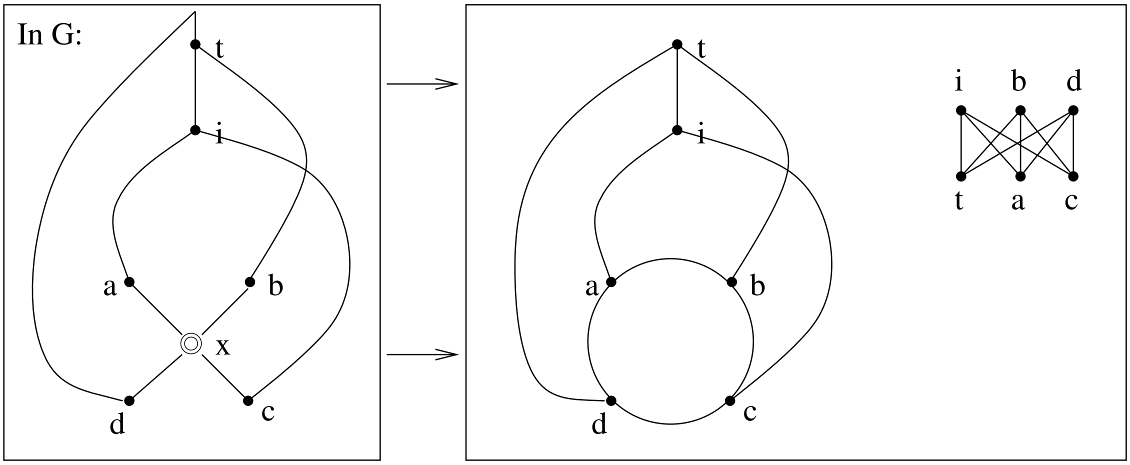


FIGURE 32.18: If a C node x has four neighbors (a, b, c, d) in that order, such that there are disjoint paths from a and c to i and b and d to an ancestor of i, then the cycle that x represents, together with these paths, gives a $K_{3,3}$.

32.4 Acknowledgment

We would like to thank the National Science Council for their generous support under Grant NSC 91-2213-E-001-011.

References

[1] K. Appel and W. Haken. Every planar map is four colorable. part i. discharging. *Illinois J. Math.*, 21:429–490, 1977.

[2] K. Appel and W. Haken. Every planar map is four colorable. part ii. reducibility. *Illinois J. Math.*, 21:491–567, 1977.

[3] S. Benzer. On the topology of the genetic fine structure. *Proc. Nat. Acad. Sci. U.S.A.*, 45:1607–1620, 1959.

[4] K. S. Booth. *PQ-Tree Algorithms.* PhD thesis, Department of Computer Science, U.C. Berkeley, 1975.

[5] S. Booth and S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *J. Comput. Syst. Sci.*, 13:335–379, 1976.

[6] J. Boyer and W. Myrvold. Stop minding your P's and Q's: A simplified $o(n)$ embedding algorithm. *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 10:140–149, 1999.

[7] J. Cai, X. Han, and R. E. Tarjan. An $o(mlogn)$-time algorithm for the maximal planar subgraph problem. *SIAM J. Comput.*, 22:1142–1162, 1993.

[8] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa. A linear algorithm for embedding planar graphs using pq-trees. *J. Comput. Syst. Sci.*, 30:54–76, 1985.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* McGraw Hill, Boston, 2001.

[10] D. R. Fulkerson and O. Gross. Incidence matrices and interval graphs. *Pacific J. Math.*, 15:835–855, 1965.

[11] M. Habib, R. M. McConnell, C. Paul, and L. Viennot. Lex-bfs and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing. *Theoretical Computer Science*, 234:59–84, 2000.

[12] F. Harary. *Graph Theory.* Addison-Wesley, Reading, Massachusetts, 1969.

[13] J. E. Hopcroft and R. E. Tarjan. Efficient planarity testing. *J. Assoc. Comput. Mach.*, 21:549–568, 1974.

[14] W. L. Hsu. PC-trees vs. PQ-trees. *Lecture Notes in Computer Science*, 2108:207–217, 2001.

[15] W. L. Hsu. A simple test for the consecutive ones property. *Journal of Algorithms*, 42:1–16, 2002.

[16] W. L. Hsu and T. Ma. Fast and simple algorithms for recognizing chordal comparability graphs and interval graphs. *SIAM J. Comput.*, 28:1004–1020, 1999.

[17] W. L. Hsu and R. M. McConnell. PC trees and circular-ones arrangements. *Theoretical Computer Science*, 296:59–74, 2003.

[18] P. N. Klein and J. H. Reif. An efficient parallel algorithm for planarity. *Journal of Computer and System Science*, 18:190–246, 1988.

[19] N. Korte and R. H. Moehring. An incremental linear-time algorithm for recognizing interval graphs. *SIAM J. Comput.*, pages 68–81, 1989.

[20] D. Kratsch, R. M. McConnell, K. Mehlhorn, and J. P. Spinrad. Certifying algorithms for recognizing interval graphs and permutation graphs. *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 866–875, 2003.

[21] C. Kuratowski. Sur le problème des courbes gauches en topologie. *Fundamata Mathematicae*, pages 271–283, 1930.

[22] C. Lekkerker and D. Boland. Representation of finite graphs by a set of intervals on the real line. *Fund. Math.*, 51:45–64, 1962.

[23] W. F. Lu and W.-L. Hsu. A test for interval graphs on noisy data. *Lecture Notes in Computer Science*, 2467:196-208, 2003.

[24] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In P. Rosenstiehl, editor, *Theory of Graphs*, pages 215–232. Gordon and Breach, New York, 1967.

[25] R. M. McConnell. A certifying algorithm for the consecutive-ones property. *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA04)*, 15:761–770, 2004.

[26] R. M. McConnell and J. P. Spinrad. Linear-time modular decomposition and efficient transitive orientation of comparability graphs. *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, 5:536–545, 1994.

[27] R. M. McConnell and J. P. Spinrad. Linear-time transitive orientation. *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, 8:19–25, 1997.

[28] R. M. McConnell and J. P. Spinrad. Modular decomposition and transitive orientation. *Discrete Mathematics*, 201(1-3):189–241, 1999.

[29] R. M. McConnell. Linear-time recognition of circular-arc graphs. *Algorithmica*, 37:93–147, 2003.

[30] K. Mehlhorn. Graph algorithms and NP-completeness. *Data structures and algorithms*, 2:93–122, 1984.

[31] K. Mehlhorn and P. Mutzel. On the embedding phase of the hopcroft and tarjan planarity testing algorithm. *Algorithmica*, 16:233–242, 1996.

[32] D. Rose, R. E. Tarjan, and G. S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput.*, 5:266–283, 1976.

[33] W. K. Shih and W.-L. Hsu. A simple test for planar graphs. *Proceedings of the International Workshop on Discrete Math. and Algorithms, University of Hong Kong*, pages 110–122, 1993.

[34] W. K. Shih and W. L. Hsu. A new planarity test. *Theoretical Computer Science*, 223:179–191, 1999.

[35] J. Small. A unified approach to testing, embedding and drawing planar graphs. *Proc. ALCOM International Workshop on Graph Drawing, Sevre, France*, 1993.

[36] H. Stamm-Wilbrandt. A simple linear-time algorithm for embedding maximal planar graphs. *Proc. ALCOM International Workshop on Graph Drawing, Sevre, France.*, 1993.

[37] R. Thomas. Planarity in linear time – lecture notes, georgia institute of technology. *Available at www.math.gatech.edu/~thomas/*, 1997.

[38] A. Tucker. An efficient test for circular-arc graphs. *SIAM Journal on Computing*, 9:1–24, 1980.

[39] M. S. Waterman. *Introduction to Computational Biology: Maps, Sequences and Genomes.* Chapman and Hall, New York, 1995.

33

Data Structures for Sets

Rajeev Raman
University of Leicester

33.1 Introduction

Sets are a fundamental concept in computer science: the study of algorithms and data structures for maintaining them were among the earliest developments in data structures. Our focus will be on problems that involve maintaining a family $\mathcal{F}$ of sets, where all sets are drawn from a *universe* U of elements. We do not assume that there is a particular natural order among the elements of U, and in particular do not focus on data structuring problems on sets where the operations explicitly assume such an order (e.g. priority queues). A *base* repertoire of actions is as follows:

$$\left.\begin{array}{ll}
\textsf{create}() & \text{Create a new empty set, add it to } \mathcal{F} \text{ and return the name of the set.} \\
\textsf{destroy}(A) & \text{If } A = \emptyset \text{ then remove } A \text{ from } \mathcal{F}. \text{ If } A \neq \emptyset \text{, flag an error.} \\
\textsf{insert}(x, A) & \text{Set } A \leftarrow A \cup \{x\}. \\
\textsf{delete}(x, A) & \text{Set } A \leftarrow A - \{x\}.
\end{array}\right\} \tag{33.1}$$

The following operation is fundamental for data structuring problems involving sets in general, but plays only an auxiliary role in this chapter:

$$\textsf{member}(x, A) \quad \text{Returns `true' if } x \in A \text{ and `false' otherwise.} \tag{33.2}$$

If we take only insert, delete and member, we get the *dictionary* problem, covered in Part III. The base repertoire plus member is essentially no more difficult, as it represents the problem of maintaining a collection of independent dictionaries over a common universe. In this chapter, we focus on adding operations to this base repertoire that take two or more sets as arguments. For example, we could consider the standard set-theoretic operations on two sets A and B:

$$A \text{ op } B, \text{op} \in \{\cup, \cap, -\}.$$

A data structure may support only an *enumerative* form of these operations, whereby the result of A op B is, in essence, some kind of enumeration of the set A op B. This result may not be in the same representation as A or B, and so one may not be able operate on it (e.g. $((A \text{ op } B) \text{ op } C)$ may not involve just two executions of op). The complexity of the algorithms will generally be measured in terms of the following parameters:

$$\begin{array}{rcl} n &=& \sum_{A \in \mathcal{F}} |A| \text{ (the total size of all sets in the family)} \\ m &=& |U| \text{ (the size of the universe)} \\ k &=& |\mathcal{F}| \text{ (the number of sets in the family)} \end{array} \tag{33.3}$$

33.1.1 Models of Computation

The problems that we consider in this chapter have been studied on a variety of different computation models. The primary models for proving upper bounds are the *pointer machine* model and the *random-access machine (RAM)* model. The pointer machine [22, 35, 40] postulates a storage consisting of an unbounded collection of registers (or records) connected by pointers. Each register can contain an arbitrary amount of additional information, and no arithmetic is allowed to compute the address of a register. The processor has a constant number of (data and pointer) registers that it can manipulate directly, but all other temporary results must be held in the storage records. In particular, this means that to answer a query, the processor must either start from a pointer p into the data structure provided by the 'user' or from one of the constant number of pointers it itself holds, and explore the data structure by following pointers starting from p.

The RAM model we use is a standard variant of the original RAM model [1], the *word RAM* model [17]. Briefly, the word RAM model uses the unit-cost criterion, and a word size of $\Theta(\log n)$ bits, where $n = \sum_{S \in \mathcal{F}} |S|$.* Clearly the word size should be at least $\log n + O(1)$ bits—otherwise one could not even address the amount of memory required to store $\mathcal{F}$. Nevertheless, there are instances where the solutions that result from the use of this model could be viewed as "cheating". For example, we could have $n = 2^{\Theta(|U|)}$, in which case the word size would be $\Theta(|U|)$ bits, which would allow most set operations to be done in $O(1)$ time by bitwise operations on a single word. The solutions that we discuss, however, do not exploit this fact. In the related *cell probe* model, the storage of the computer is modeled by cells numbered $0, 1, 2, \ldots$, each of which can store a number of $O(\log n)$ bits. The running time of an algorithm in this model is measured as just the number of words (cells) accessed during an operation. All other computations are free.

The *arithmetic* model, used primarily for proving lower bounds, was proposed by Fredman [14] and Yao [43]. We now give a somewhat simplified description of this model which conveys the essential aspects: the interested reader is referred to [25, section 7.2.3] for further details. In many cases of interest, it is useful to assume that the data structure operates on values from a set $\mathcal{M}$, which is a *monoid*. This means that $\mathcal{M} = (\mathcal{M}, +, 0)$ is augmented with an associative and commutative operator $+$ such that $\mathcal{M}$ is closed under $+$ and 0 is the identity element for $+$. The data structure is modeled as an collection of

*To some readers, the idea that the wordsize of a machine can change as we update the data structure may appear a little strange, but it is merely a formal device to ensure reasonable usage.

variables $v_0, v_1, \ldots$, each of which can hold a value from $\mathcal{M}$ and initially contains 0. After receiving the input to each operation, the algorithm executes a sequence of operations of the form $v_i \leftarrow$ INPUT, $v_i \leftarrow v_j + v_k$ or OUTPUT $\leftarrow v_i$. The algorithm must be correct for all choices of $\mathcal{M}$, thus in particular it cannot assume that the operator + is invertible. The cost of an algorithm in processing a sequence of operations is the total number of such instructions executed by it. The restriction that $\mathcal{M}$ is a monoid (rather than say, a group) is partly for ease of proving lower bounds. However, known algorithms in this framework do not gain significant asymptotic speedups by exploiting stronger models (e.g. by using the fact that + is invertible in groups).

33.2 Simple Randomized Set Representations

In this section we cover a simple, but general-purpose set representation due to [33, 34]. In addition to the base repertoire (33.1), we wish to support:

$$A \ \mathsf{op}\ B, \mathsf{op} \in \{\cup, \cap, -\},$$

as well as the following boolean operations:

$$\mathsf{equal}(A, B) \qquad \text{Returns 'true' if } A = B \text{ and 'false' otherwise.} \tag{33.4}$$

$$\mathsf{subset}(A, B) \qquad \text{Returns 'true' if } A \subseteq B \text{ and 'false' otherwise.} \tag{33.5}$$

This representation touches upon a topic of interest in its own right, that of the *unique* representation of sets, which can be defined as follows. We consider a class of representations that suffice to represent all subsets of a given universe U. However, we require that each set should have a a *unique* representation within that class. In particular the representation should be independent of the sequence of operations that created the set (for example, a red-black tree representation is not unique). Unique representations have the desirable property that it is then possible to ensure that all instances of the same set within a family (including sets that are created as intermediate results in a computation) are represented by the same object within the data structure. This allows constant-time equality tests of two sets: just check if they are the same object! The difficulty, of course, is to combine unique representations with rapid updates.

This definition also applies to *randomized* algorithms, which may access a source of random bits while executing a query or update, and the uniqueness requirement here means that the choice of representation for a set depends solely upon the sequence of random bits that are output by the source. (If, as in practice, one uses a pseudo-random generator, then the representation for a given set depends only upon the seed used by the pseudo-random number generator.)

33.2.1 The Hash Trie

We first weaken the notion of a unique representation, and speak about the unique representation of sets from a *labeled* universe. Here, we assume that the elements of the universe U are labeled with (not necessarily unique) b-bit strings, and for $x \in U$ we denote its label by $\ell(x)$. In addition, we also have the notion of a *labeled set* A_y, where y is a sequence of $\leq b$ bits. Any labeled set A_y satisfies the property that for all $x \in A_y$, y is a prefix of $\ell(x)$.

Given a set from a labeled universe, one can of course use the well-known binary trie [21] to represent it. For the sake of precision we now define the binary trie, when used to represent a (labeled) set S_y:

- if $|S_y| = 0$ then the trie is empty.
- if $|y| = b$ then the trie is a leaf node that contains a pointer to a linked list containing all elements of S_y.
- otherwise, the trie comprises an internal node with an edge labeled 0 pointing to a binary trie for the set

$$S_{y0} = \{x \in S_y \mid y0 \text{ is a prefix of } \ell(x)\}$$

and an edge labeled 1 pointing to a binary trie for the labeled set

$$S_{y1} = \{x \in S_y \mid y1 \text{ is a prefix of } \ell(x)\}$$

A set $S \subseteq U$ is represented as the labeled set S_Λ, where Λ denotes the empty string.

Consider a family $\mathcal{F}$ of sets from a universe U and let each node in the resulting collection of tries represent a labeled set in the natural way. Then if two nodes represent labeled sets A_z and B_z, such that $A_z = B_z$ (note that the label is the same) then the subtrees rooted at these nodes have an identical structure. One can then save space by ensuring that all sets in $\mathcal{F}$ whose representations have instances of the set A_z point to a single instance of A_z. By consistently applying this principle, we will ensure that two sets S, T, $S = T$ point to a single instance of S_Λ.

We now give an example. Let $U = \{a, b, c, d, e, f\}$ and $\mathcal{F}$ contain the sets $X = \{a, c, f\}$, $Y = \{c, d, e\}$ and $Z = \{a, b, c, f\}$. Suppose that the elements of U are labeled as follows:

$$\ell(a) = 001, \ell(b) = 011, \ell(c) = 010, \ell(d) = 101, \ell(e) = 110, \ell(f) = 010$$

Without exploiting the unique representation property, we would get the representation in Figure 33.1(i), and by exploiting it and storing subtrees uniquely it we get the representation in Figure 33.1(ii). Updates now need to be done with a little care: one cannot, for example, add a new child to a node, as the subtree in which the node lies may be shared among a number of sets. The solution is to use *nondestructive* updates, namely, implementing updates by making a *copy* of the path from the leaf to be added/deleted to the name of the sets (cf. *persistent* data structures). Figure 33.1(iii) shows the state of the data structure after executing the commands $\mathsf{insert}(a, Y)$ and $\mathsf{insert}(b, X)$, which we now explain.

First, let us consider the operation $\mathsf{insert}(a, Y)$. We need to insert a as a sibling of c in the trie for Y. Since the path from c up to the root of Y could potentially be shared by other sets, we do not modify the node that is the parent of c, instead making a copy of the entire path from c to the node that is the root of the representation of Y, as shown in the figure. The pointer for Y is then appropriately modified. The nodes that were previously part of the representation of Y (shown in dotted outline) are, in this example, not reachable as part of any set and may be cleared as part of a garbage-collection phase (they are not explicitly freed during the insertion). Now coming to $\mathsf{insert}(b, X)$, we proceed as before, and do not insert b directly as a sibling of (c, f) under X. However, before creating a new node with the leaves b and (c, f) as children, we check the data structure to see if such a node exists and find one (this is the node representing the set Z_{01}). Therefore, we avoid creating this node. Continuing, we discover that all the nodes that we would have tried to create as part of this insertion already exist and therefore conclude that the sets X and Z are now the same (and hence their tries should be represented by the same node).

To support the reuse of existing nodes, a dictionary data structure that stores all nodes currently in the data structure is maintained. A node x with left child y and right child z is stored in this dictionary with the key $\langle y, z \rangle$ (either of y or z could be NIL). Each insertion requires $\Theta(b)$ lookups or insertions into this dictionary. Avoiding this overhead is

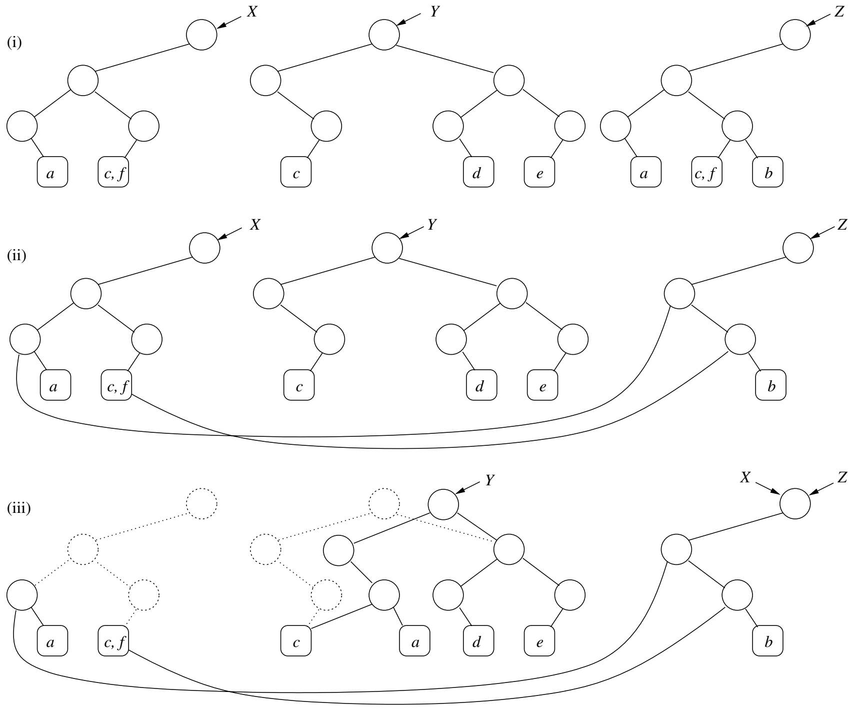


FIGURE 33.1: Unique set representations.

important, but in a randomized setting the dictionary can be implemented using dynamic perfect hashing, giving $O(1)$ overhead. This idea is also used in practical implementations of functional programming languages such as LISP, and is referred to as *hashed consing* in that context[†].

In a randomized setting, the other problem, that of obtaining suitable labels, can also be solved easily. We simply let ℓ be a function chosen at random from a universal class of hash functions (cf. Chapter 9). By choosing $\ell : U \to \{1, \ldots, n^3\}$(for example) we can ensure that the number of *collisions*, or pairs of distinct elements x, y in U with $\ell(x) = \ell(y)$ is $O(1)$ with a probability that tends to 1 as $n \to \infty$. This means that we can essentially ignore the possibility that there are more than a constant number of elements at the leaf of any trie. Clearly, labels are $O(\log n)$ bits long.

Note that we can ensure that both the lengths of the labels and the number of collisions stays under control as n changes, by simply rehashing whenever the value of n doubles or halves since the last rehash. We now analyze some parameters of this data structure. Clearly, equality testing takes $O(1)$ time. Each insertion or deletion takes time and space proportional to the depth of the tries, which is $O(\log n)$. Both insertions and deletions may cause some nodes to become 'garbage' (i.e. unreachable from any set)—these need to be compacted periodically, for example, when a rehash occurs. It is easy to verify that the amortized cost of rehashing and garbage collection is negligible. This gives parts (i) and (ii) of the following theorem; for part (iii) we refer the reader to [33, Chapter 8]:

[†] The operation for creating a new node is called a CONS.

THEOREM 33.1 *There is a data structure for a family of sets that:*

(i) supports insert *and* delete *in $O(\log n)$ amortized expected time and equality-testing in $O(1)$ worst-case time;*

(ii) uses $O(n \log n)$ space, and

(iii) given two sets A and B, supports the constructive version of A op *B for* op $\in \{\cup, \cap, -, \subseteq\}$ *in $O(|S_2|(1 + \log(S_3/S_2)$ amortized expected time, where S_2 and S_3 are the middle and largest sets of $A - B, B - A$ and $A \cap B$.*

REMARK 33.1 Note that a fairly naive bound for an operation A op B, for op $\in \{\cap, \cup, -, \subseteq\}$ is $O(|A| \log |B|)$, implemented by storing each tree in the family in a binary-tree based dictionary. Thus, Theorem 33.1 is not, in the worst case, a lot better. On the other hand, if we have two large sets S and T that differ in only a constant number k of elements then the expected running time of the above algorithm is $O(k \log \max\{|S|, |T|\})$.

33.2.2 Some Remarks on Unique Representations

There are interesting issues regarding the unique representation problem, in particular for unlabeled universes. We first mention here that both the *randomized search tree* and the *skip list* (cf. Chapter 13) satisfy the unique representation described above, even for unlabeled universes, and support the dictionary operations (insert, delete and member) in $O(\log n)$ expected time. In each of these cases, the algorithm described there should be modified so that instead of choosing a random height for a node in a skip list, or a random priority for a node in a randomized search tree, we should choose a function $U \rightarrow [0, 1]$ that behaves 'randomly' (we refer the reader to [36] for a precise theoretical statement).

By representing each set in a family using either one of these representations we get alternatives to parts (i) and (ii) for Theorem 33.1. However, one may ask about deterministic unique representations that support rapid updates. The reader is directed to [3, 38] and the references therein for pointers to the literature on this fascinating area. We only note here that [3] show a $\Theta(n^{1/3})$ time bound for supporting the above dictionary operations, provided that the class of representations is restricted to "graph-like" representations (including, of course, trees). This shows an exponential separation between randomized uniquely represented dictionaries and deterministic (non-unique) dictionaries on the other hand, and deterministic uniquely represented dictionaries on the other. This result is not entirely conclusive, however: by labeling elements in U with integers from $\{0, \ldots, m - 1\}$, as observed in [38, Section 2.3], the above trie-based approach gives uniquely represented dictionaries that support all operations in $O(\log m)$ time, where $m = |U|$. Thus, there is a "dichotomy" depending upon whether we are interested in bounds that depend on m alone or n alone (cf. Chapter 39). It is not known how the complexity of unique representation depends on the relationship between m and n. Indeed, the gap in knowledge is quite large, as the $\Omega(n^{1/3})$ applies when $n \sim \log^* |U|$, while the $O(\log m)$ algorithm is optimal only when $n \sim |U|^\epsilon$ for some constant $\epsilon > 0$.

33.3 Equality Testing

We now consider representations that only support the base repertoire (33.1) and the equal operation (33.4). Clearly we would like solutions that are better (at least in some respects) than those given by Theorem 33.1, and we begin by considering deterministic data structures. We first discuss one due to [38], which is based on binary tries (cf. Section 33.2).

The operations applied to our data structure are divided into *epochs*. At the start of an epoch the data structure is rebuilt 'from scratch'. Without loss of generality we can consider U to consist of all the elements that were present in $\mathcal{F}$ at the start of an epoch, plus all the elements that have appeared as the argument to an insert or delete operation since then (note that U may contain elements no longer present in $\mathcal{F}$). We can then label elements in U with values from $\{0, \ldots, |U| - 1\}$, using $\lceil \log m \rceil$-bit integers, where $m = |U|$. Whenever we insert an element that is not in U, we give it the next higher integer. If $\sum_{A \in \mathcal{F}} |A| = n_0$ at the start of an epoch, we start the next epoch after $n_0/2$ updates (inserts or deletes). Rebuilding from scratch involves resetting U to be only those elements that are currently present in $\mathcal{F}$, relabeling all elements with integers from $\{0, \ldots, |U| - 1\}$, and recreating the data structure with these labels, as well as any auxiliary data structures. It will be easy to see that the amortized cost of rebuilding is negligible, and that $|U| = O(n)$ at all times.

Each set is represented as a binary trie, with shared subtrees as in the previous section. Updates are also done non-destructively. A key difference is the method used to avoid creating nodes that already exist. Nodes are numbered serially in order of creation. If node p points to node v, we say that p is one of (possibly many) parents of v. Each node v, maintains a set $\mathit{parents}(v)$ of all its parents. Each parent $p \in \mathit{parents}(v)$ is assigned a key equal to $\langle \mathit{node\text{-}number}(w), b \rangle$, where w is the other node (besides v) pointed to by p, and b equals 0 or 1 depending on whether v is a left child of p or not. When doing an update, before creating a new node with left child x and right child y, we we search set $\mathit{parents}(x)$ for the key $\langle \mathit{node\text{-}number}(y), 0 \rangle$. If such a key is found, we return the matching parent, which is precisely a node with left child x and right child y. Otherwise, create a new node p with pointers to v and w, set *parents(p)* to empty, insert p into $\mathit{parents}(v)$ and $\mathit{parents}(w)$, and return p. The issue is how to represent the sets $\mathit{parents}(v)$ (cf. the dictionary in the previous section).

Each set $\mathit{parents}(v)$ is represented by a binary search tree and a variation of the splay algorithm is used to perform searches. The splay algorithm is especially appropriate as it can be used to exploit patterns in the sequence of accesses. Although a detailed consideration of splay trees is beyond the scope of this chapter, the necessary facts are, roughly, that (a) insertion of a key that is larger than the maximum key currently in the tree (a *passive* insertion) has constant amortized cost while an insertion that is not passive (an *active*) insertion) has logarithmic amortized cost (b) if the frequency of search for an item i is $0 < \alpha_i < 1$ then the amortized cost of all searches for item i is essentially $O(1 + \log(1/\alpha_i))$. This is summarized in the lemma below:

LEMMA 33.1 Consider a sequence of insertions and searches performed on an (initially empty) binary search tree using splays. Let:

$$\begin{aligned} s_i &= \text{the number of searches of item } i \\ s &= \text{the total number of searches,} \\ a &= \text{the number of active insertions, and} \\ p &= \text{the number of passive insertions.} \end{aligned}$$

The cost of this sequence is $O((p + a) + s + a \log a + \sum_{s_i \geq 1} s_i \log(s/s_i))$.

REMARK 33.2 Insertions into an initially non-empty tree are easily handled: pretend to start with an empty tree and initialize the tree with a sequence of passive insertions. By the above lemma, the additional cost is linear in the size of the initial tree.

We now argue that although an update (insert or delete) may result in $\Theta(\log m)$ node creations, and hence as many insertions into the *parents* dictionaries, most of the insertions are passive. Specifically, let $x\langle y, z\rangle$ denote the creation of a node x with children y and z. Then, an update may result in a sequence of node creations: $y_1\langle x_0, x_1\rangle, y_2\langle y_1, x_2\rangle, \ldots, y_l\langle y_{l-1}, x_l\rangle$, where $y_1, \ldots, y_l$ are 'new' nodes and $x_0, \ldots, x_l$ are nodes that existed previously in the data structure. Note that because the y_i's are new nodes, they have higher *node-number*s than the x_is. As a result, of the $2l$ insertions into *parents* dictionaries caused by this update, all but two—the insertion of y_i with key $\langle \mathit{node\text{-}number}(x_0), b\rangle$ $(\langle \mathit{node\text{-}number}(x_1), 1-b\rangle)$ into $\mathit{parents}(x_1)$ $(\mathit{parents}(x_0))$—are passive.

We now need to bound the time to perform searches on the *parents* dictionary. For this, we consider a single epoch, and let $G = (V, E)$ be directed acyclic graph formed by the nodes at the end of the epoch. The sources (nodes of zero in degree) are the roots of (current and past) sets and the sinks (nodes of zero outdegree) are the elements of U. Note that all nodes have outdegree at most 2.

An update to sets in $\mathcal{F}$ results searches in the the *parents* dictionary of nodes that lie along a path π from a source to a sink in G. It should be noted that the path is traversed in reverse, from a sink to a source. Note that each time that a search traverses an edge (v, w), where w is a parent of v, the key that is searched for in $\mathit{parents}(v)$ is the same. Thus we can identify the traversal of an edge in G with a search of a particular key in a particular dictionary. Let a_e denote the number of times an edge $e \in E$ is traversed, and note that we can delete from G all edges with $a_e = 0$. Letting $A_v = \sum_{(w,v)\in E} a_{(w,v)}$, the cost of searches in $\mathit{parents}(v)$ for any vertex $v \in V$, denoted $\mathit{cost}(v)$, is given by $\sum_{(w,v)\in E} a_{(w,v)} \log A_v / a_{w,v}$, by Lemma 33.1. Let V' denote the set of nodes that are neither sinks nor sources, and note that for any $v \in V'$, $\sum_{(w,v)\in E} a_{(w,v)} = \sum_{(v,w)\in E} a_{(v,w)}$. Thus, we have:

$$
\begin{aligned}
\sum_{v\in V} \mathit{cost}(v) &= \sum_{(w,v)\in E} a_{(w,v)} \log A_v / a_{w,v} \\
&= \sum_{v\in V'} \sum_{(w,v)\in E} a_{(w,v)} \log A_v + \sum_{v\in V-V'} \sum_{(w,v)\in E} a_{(w,v)} \log A_v + \sum_{e\in E} a_e \log 1/a_e \\
&= \sum_{v\in V'} \sum_{(v,w)\in E} a_{(v,w)} \log A_v + \sum_{v\in V-V'} \sum_{(w,v)\in E} a_{(w,v)} \log A_v + \sum_{e\in E} a_e \log 1/a_e \\
&\le \sum_{v\in V'} \sum_{(v,w)\in E} a_{(v,w)} \log A_v / a_{(v,w)} + \sum_{v\in V-V'} \sum_{(w,v)\in E} a_{(w,v)} \log A_v \\
&\le \sum_{v\in V'} \log \mathit{out}(v) + \sum_{v\in V-V'} \sum_{(w,v)\in E} a_{(w,v)} \log A_v
\end{aligned}
$$

where $\mathit{out}(v)$ denotes the out-degree of v. The last step uses the fact that for all $\alpha_1, \ldots, \alpha_d$, $\alpha_i \in [0, 1]$ and $\sum_i \alpha_i = 1$, $\sum_{i=1}^d \alpha_i \log(1/\alpha_i) \le \log d$ (the 'entropy' inequality).

Note that $\sum_{v\in V-V'} \sum_{(w,v)\in E} a_{(w,v)}$ is just the number of updates in this epoch and is therefore $\Theta(n)$. Since $A_v = O(n)$ we can bound the latter term by $O(n \log n)$. Since the outdegree of each node is 2, the former term is $O(V)$, which is also $O(n \log n)$. We thus find that all dictionary operations in the sets $\mathit{parents}(v)$ take $O(n \log n)$ time, and so the amortized cost of an update is $O(\log n)$. To conclude:

THEOREM 33.2 *There is a data structure for a family of sets that supports* insert, delete *and* member *in $O(\log n)$ amortized time,* equal *in $O(1)$ worst-case time, and uses $O(n \log n)$ words of space, where n is the current size of the family of sets.*

We now describe the *partition tree* data structure for this problem [24, 44]. Since the partition tree is closely related to the above data structure, results based on the partition tree are not significantly different from those of Theorem 33.2. We describe a slightly simplified version of the partition tree by assuming that U is fixed. The partition tree is a *full* binary tree T with $|U|$ leaves, i.e. a tree where the leaves are all at depth $\lfloor|\log U|\rfloor$ or $\lceil|\log U|\rceil$, and all nodes have either two or zero children. At the leaves of this tree we place the elements of U (in any order). For each internal node v of T, we let $D(v)$ denote the elements of U that are at leaves descended from v. A set $A \in \mathcal{F}$ is stored at an internal node v if $D(v) \cap A \neq \emptyset$. Furthermore, all sets that are stored at an internal node v are grouped into equivalence classes, where the equivalence relation is given by $A \equiv B \Leftrightarrow (A \cup D(v) = B \cup D(v))$. Clearly, two sets are equal iff they belong to the same equivalence class at the root of the tree and so this representation supports constant-time equality testing. Note that if n_v is the number of sets stored at an internal node v of T, then $\sum_v n_v = O(n \log|U|)$. This is because each set containing x appears once on each node from the path leading from x to the root (and hence $O(\log|U|)$ times in all), and $\sum_{x\in U} |\{A \in \mathcal{F} \mid x \in A\}| = n$. The key issue, of course, is how to update the partition tree when executing $\mathsf{insert}(x, A)$ ($\mathsf{delete}(x, A)$). We traverse T from x to the root, storing (or removing) A from all the nodes along the path. We now show how to maintain the equivalence classes at these nodes.

At the leaf, there is only one equivalence class, consisting of all sets that contain x. We merely need to add (delete) A to (from) this equivalence class. In general, however, at each node we need to determine whether adding/deleting x to/from A causes A to move into a new equivalence class of its own or into an existing equivalence class. This can be done as follows. Suppose γ is a (non-empty) equivalence class at a (non-leaf) node u in T and suppose that v, w are u's children. A little thought shows that must be (non-empty) equivalence classes α, β at v, w respectively such that $\gamma = \alpha \cap \beta$. A global dictionary stores the name of γ with the key $\langle\alpha, \beta\rangle$. Inductively assume that following an operation $\mathsf{insert}(x, A)$ (or $\mathsf{delete}(x, A)$) we have updated the equivalence class of A at all ancestors of x up to and including a node v, and suppose that A is in the equivalence class α at v. Next we determine the equivalence class β of A in v's sibling (this would not have changed by the update). We then look up up the dictionary with the key $\langle\alpha, \beta\rangle$; if we find an equivalence class γ stored with this key then A belongs to γ in v's parent u, otherwise we create a new equivalence class in u and update the dictionary.

Lam and Lee [24] asked whether a solution could be found that performed all operations in good single-operation worst-case time. The main obstacles to achieving this are the amortization in the splay tree and the periodic rebuilding of the partition tree. Again dividing the operation sequence into *epochs*, Lam and Lee noted that at the end of an epoch, the partition tree could be copied and rebuilt incrementally whilst allowing the old partition tree to continue to answer queries for a while (this is an implementation of the *global rebuilding* approach of Overmars [26]). By 'naming' the equivalence classes using integers from an appropriate range, they note that the dictionary may be implemented using a two-dimensional array of size $O(n^2 \log n)$ words, which supports dictionary operations in $O(1)$ worst-case time. Alternatively, using standard balanced binary trees or Willard's q-fast tries [42], or the more complex data structure of Beame and Fich [5] gives a worst-case complexity of $O(\log n)$, $O(\sqrt{\log n})$ and $O((\log\log n)^2)$ for the dictionary lookup, respectively. Using these data structures for the dictionary we get:

THEOREM 33.3 *There is a data structure for a family of sets that supports* equal *in* $O(1)$ *worst-case time, and* insert *and* delete *in either (i)* $O(\log n(\log\log n)^2)$ *worst-case time*

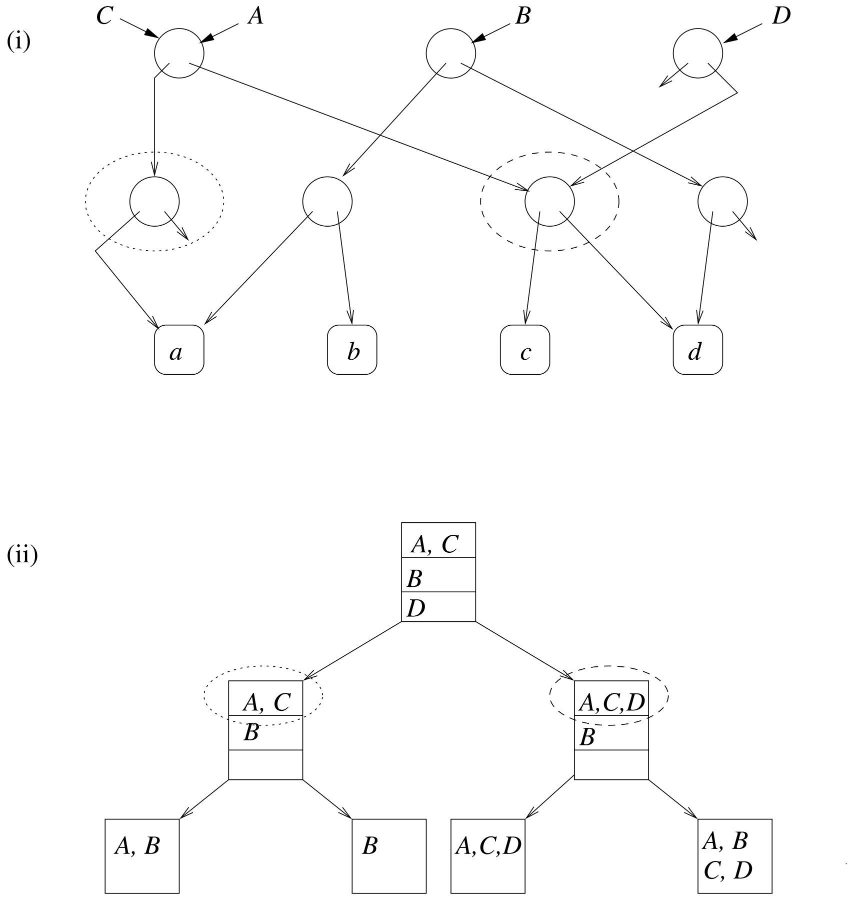


FIGURE 33.2: The partition tree (ii) and its relation to the DAG created by the binary trie representation with shared subtrees.

and $O(n \log n)$ space or (ii) $O(\log n)$ worst-case time and $O(n^2 \log n)$ space.

REMARK 33.3 The reader may have noticed the similarities between the lookups in partition trees and the lookup needed to avoid creating existing nodes in the solutions of Theorems 33.1 and 33.2; indeed the examples in Figure 33.2 should make it clear that, at least in the case that $|U|$ is a power of 2, there is a mapping from nodes in the DAG of Theorem 33.2 and partitions in the partition tree. The figure illustrates the following example: $U = \{a, b, c, d\}$, $\mathcal{F} = \{A, B, C, D\}$, $A = \{a, c, d\} = C$, $B = \{a, b, d\}$, $D = \{c, d\}$, and an assumed labeling function that labels a, b, c and d with 00, 01, 10 and 11 respectively. The partitions in (ii) shown circled with a dashed/dotted line correspond to the nodes circled with a dashed/dotted line in (i).

33.4 Extremal Sets and Subset Testing

This section deals with testing sets in $\mathcal{F}$ for the subset containment relationship. We first survey a static version of this problem, and then consider a dynamisation.

33.4.1 Static Extremal Sets

Here we assume that we are given a family $\mathcal{F}$ of sets from a common universe as input. A set S is *maximal* in $\mathcal{F}$ if there is no set $T \in \mathcal{F}$ such that $S \subset T$. A set S is *minimal* in $\mathcal{F}$ if there is no set $T \in \mathcal{F}$ such that $S \supset T$. The *extremal sets* problem is that of finding all the maximal (or all the minimal) sets in $\mathcal{F}$. A closely related problem is the computation of the *complete subset graph*, which represents all edges in the partial order induced among members of $\mathcal{F}$ by the subset relation. Specifically, the complete subset graph is a directed graph whose vertices are the members of $\mathcal{F}$ and there is an edge from x to y iff $x \subset y$. This is a problem that arises in a number of practical contexts, for example, it can be used to maximally simplify formulae in restricted conjunctive normal form [29].

This problem has been considered in a number of papers including [29–32, 46]. We now summarize the results in these papers. As before, we let $k = |\mathcal{F}|$ denote the size of the family, and $n = \sum_{S \in \mathcal{F}} |S|$ be the total cardinality of all sets. A number of straightforward approaches can be found with a worst-case complexity of $O(n^2)$. It can be shown that the subset graph has $\Omega(n^2/(\log n)^2)$ edges [46], which gives a lower bound for any algorithm that explicitly lists the complete subset graph. An aim, therefore, is to bridge the gap between these bounds; all results below are in the word RAM model.

Yellin and Jutla gave an algorithm that computes the subset graph using $O(n^2/\log n)$ dictionary operations. Using hashing, all dictionary operations take $O(1)$ expected time, and we get an $O(n^2/\log n)$ expected running time. Using Willard's q-fast tries [42], or the more complex data structure of Beame and Fich [5] gives running times of $O(n^2/\sqrt{\log n})$ or $O((n \log\log n)^2/\log n)$ respectively. Pritchard [30] gave a simple algorithm that ran in $O(n^2/\log n)$ time. In [31] he re-analyzed an earlier algorithm [29] to show that this algorithm, too, ran in $O(n^2/\log n)$ time (the algorithm of [29, 31] uses RAM operations less extensively and is a good candidate for porting to the pointer machine model). Finally, Pritchard [32] gave an algorithm that uses the bit-parallelism of the word RAM extensively to achieve an $O(n \log\log n/(\log n)^2)$ running time.

All of the above are static problems—it is very natural to ask about the complexity of this problem when updates may change the sets in $\mathcal{F}$. Again, if one wishes explicitly to maintain the entire subset graph, it is easy to see that $\Omega(n)$ edges may change as the result of a single update. Take $U = \{1, \ldots, u\}$ and $\mathcal{F} = \{A, B_2, \ldots, B_u\}$, where $A = U$ and $B_i = \{1, i\}$ for $i = 2, \ldots, u$. The sum of the sizes of the sets in $\mathcal{F}$ is $2u - 2$, but deleting the element 1 from A removes all $u - 1$ edges from the complete subset graph. It is not very hard to come up with algorithms that can achieve an $O(n)$ running time (see e.g. [46]). To obtain a better complexity, therefore, we must consider algorithms that do not explicitly store this graph. One way of doing this is to consider the *dynamic subset testing* problem, defined as the problem of supporting the base repertoire (33.1) and the subset operation (33.5). Since the query subset(A, B) tests if there is an edge between A and B in the complete subset graph, this problem seems to be an appropriate dynamisation of the extremal sets problem, and it is dealt with in the next section.

33.4.2 Dynamic Set Intersections and Subset Testing

Rather than consider the dynamic subset testing problem directly, we consider a related problem, the *dynamic set intersection* problem. In addition to the base repertoire (33.1) of actions above, we consider an enumerative version of the following:

$$\text{intersect}(A, B) \quad \text{Report the intersection of sets } A \text{ and } B. \qquad (33.6)$$

Other variations could include returning the size of the intersection, or retrieving some values associated with the elements in the intersection. A unifying way to study these problems is as follows: we are given a set $\mathcal{M}$ of *information* items that will be associated with elements of U, a function $I : U \to \mathcal{M}$ that associates values from M with keys in U. We assume that $\mathcal{M} = (\mathcal{M}, +, 0)$ is a monoid. The query $\mathsf{intersect}(A, B)$ then returns $\sum_{x \in A \cap B} I(x)$, where the sum is taken with respect to $+$. It is easy to cast the intersection problem and its variants in this framework. The basic problem defined above can be obtained by letting $\mathcal{M} = (2^U, \cup, \{\})$ and $I(x) = \{x\}$ for all x, and the problem where one merely has to report the size of the intersection can be obtained by setting $\mathcal{M} = (\mathbb{N}, +, 0)$ and $I(x) = 1$ for all x, where $+$ here is normal (arithmetic) addition. Note that dynamic subset testing, defined in the previous section, easily reduces to the intersection problem: $A \subseteq B$ iff $|A| = |A \cap B|$.

We now survey the results in this area. It is useful to use the notation $\widetilde{O}(f(n)) = \cup_{c=0}^{\infty} O(f(n) \log^c n)$, which ignores polylogarithmic factors are ignored (a similar convention is used for the Ω notation, with inverse polylogarithmic factors being ignored).

For this problem, Yellin [45] gave an algorithm that processed a sequence of n insertions and deletions and q $\mathsf{intersect}$ operations in time $\widetilde{O}(n \cdot n^{1/k} + qn^{(1-1/k)})$ time for any fixed k. The intuition behind this result is as follows. Take $t = n^{1-1/k}$ and say that a set $S \in \mathcal{F}$ is *large* if $|S| \geq t$ and *small* otherwise. All sets are represented as dictionaries (allowing membership testing in logarithmic time). Intersections between two small sets, or between a small set and a large one, are handled by testing each element of one set for membership in the other. Clearly this takes $\widetilde{O}(t)$ time, and insertion into and deletion from a small set takes $\widetilde{O}(1)$ time. For every pair of large sets S, T, the algorithm keeps track of $I(S \cap T)$ explicitly, by storing the elements of $S \cap T$ in a balanced binary search tree, and storing at any internal node x the monoid sums of all the elements under x. [‡] Since there are at most $n/t = n^{1/k}$ large sets, an insertion into, or a deletion from, a large set requires updating at most $n^{1/k}$ of the pairwise intersections with other sets, and takes $\widetilde{O}(n^{1/k})$ time. This proves the time bound, modulo some details such as dealing with changes in the value of t caused by changes in n and so on.

Dietz et al. [11] noticed that if one were to process a sequence of n updates and q queries, where n and q are known in advance, then the overall cost of Yellin's algorithm for processing the sequence is minimized by taking $n^{1/k} = \min\{n, \sqrt{q}\}$, giving an overall time bound of $\widetilde{O}(q + n\sqrt{q})$. They gave an algorithm that achieves this bound even when n and q are not known in advance.

We now show that this bound is essentially tight in the arithmetic model, by giving a lower bound of $\widetilde{\Omega}(q + n\sqrt{q})$. For simplicity, consider the problem where the elements in the intersection are to be reported, i.e., take $\mathcal{M} = (2^U, \cup, \{\})$ and $I(x) = \{x\}$. Starting from an empty family $\mathcal{F}$, we build it up by insertions so that the (sums of) sizes of the pairwise intersections of sets in $\mathcal{F}$ are large, and query all possible intersections of the sets. If the answers to all the queries were to be obtained by adding (unioning) together singleton sets, then the lower bound would follow. Unfortunately, this is too simplistic: subsets obtained as temporary values during the computation of one answer may be re-used to answer another query. To get around this, we note that a subset that is used to compute the answer to several intersection queries must lie in the common intersection of all the sets involved,

[‡] If the monoid is $\mathcal{M} = (2^U, \cup, \{\})$, we do not store the monoid sum explicitly, but instead take the monoid sum at an internal node to be implicitly represented by the subtree rooted at it.

and construct $\mathcal{F}$ so that the common intersection of any of a sufficiently large (yet small) number of sets in $\mathcal{F}$ is small. This means that no large subset can be reused very often.

We assume that $q \le n^2$ (otherwise the lower bound is trivial). Starting with an empty family, we perform a sequence of $n = \Theta(N)$ insert operations and q queries. We choose $U = \{1, \ldots, 2N/\sqrt{q}\}$ and $k = |\mathcal{F}| = \sqrt{q}$; we let $m = |U|$ and let $\ell = c \log N$ for some sufficiently large constant $c > 0$. We assume the existence of a family $\mathcal{F} = \{S_1, \ldots, S_k\}$ with the following properties:

(a) $|S_i \cap S_j| = \Omega(m)$ for all i, j with $1 \le i < j \le k$.

(b) for any pairwise distinct indices $i_1, \ldots, i_\ell$, $|\cap_{j=1}^{\ell} S_{i_j}| < \ell$.

The existence of such a family is easily shown by a probabilistic argument (a collection of random sets of the appropriate size suffices). We now represent $\mathcal{F}$ in the data structure by executing the appropriate insertions. Note that this requires at most $km = N = \Theta(n)$ update operations, since $|S_i| \le m$ for all i. We then query the pairwise intersections of all the sets; there are $\binom{k}{2} = \Theta(q)$ queries in all.

Firstly, note that the sizes of all the output sets sum to $\Omega(mk^2)$ by (a) above. The output to each query is conceptually obtained by a binary *union tree* in which each internal node combines the answers from its children; the external nodes represent singleton sets. Each node can be labeled with a set in the obvious way. Consider the entire forest; we wish to count the number of distinct nodes in the forest (that is, nodes labeled with distinct sets). Since each distinct set corresponds to at least one instruction that is not counted elsewhere, counting the number of distinct sets is a lower bound on the number of instructions executed.

We consider only nodes that correspond to sets of size ℓ or larger. Clearly, the number of such sets is $\Omega(mk^2/\ell)$. Furthermore, no such set can be used to answer more than $\binom{\ell}{2}$ different queries by (b). It follows that there are $\Omega(mk^2/\ell^3) = \widetilde{\Omega}(n\sqrt{q})$ distinct sets, giving us a lower bound of this magnitude (as mentioned above, a lower bound of $\Omega(q)$ is trivial whenever $q = \widetilde{\Omega}(n\sqrt{q})$).

Dietz et al. also considered the relatively high memory usage of the above algorithms. For example, if $q = \Theta(n)$ then both Yellin's algorithm and Dietz et al.'s algorithm use $\Omega(n^{3/2})$ space. Dietz et al. also considered the complexity of the above problem when the algorithms were restricted to use s memory locations, where $n \le s \le n^2$, and gave an algorithm that processed n operations (queries and updates) in $\widetilde{O}(n/s^{1/3})$ time. Both the upper bound and the lower bound are more complex than those for the unlimited-memory case. We summarize with the main theorems of Dietz et al.:

THEOREM 33.4 *For the problem of maintaining a family of sets under* create*,* insert*,* delete *and* intersect*, we have the following results:*

(i) Any algorithm requires $\widetilde{\Omega}(q + n\sqrt{q})$ time to process a sequence of n updates and q queries in the arithmetic model of computation.

(ii) Any intermixed sequence of n updates and q queries can be processed in $\widetilde{O}(n\sqrt{q} + q)$ time using $O(\min\{n\sqrt{q}, n^2\})$ space.

(iii) There is a sequence of $O(n)$ updates and queries that requires $\widetilde{\Omega}(n^2 s^{-1/3})$ operations in the arithmetic model, if the algorithm is restricted to using s memory locations.

(iv) Any intermixed sequence of $O(n)$ updates and queries can be performed in $\widetilde{O}(n^2/s^{1/3})$ time and $O(s)$ space.

As mentioned above, the arithmetic model is a relatively weak one (making it easy to prove lower bounds), but all the algorithms that realise the upper bounds fit into this model. It would be interesting to prove similar lower bounds for stronger models or to design algorithms that improve upon the performance of the above in stronger models. Note also, that the lower bounds are for the intersection problem, not the problem that we originally considered, that of subset testing. Since we get back a boolean answer from a subset testing query, it does not fit into the arithmetic framework.

33.5 The Disjoint Set Union-Find Problem

In this section we cover the best-studied set problem: the *disjoint set union-find* problem. The study of this problem and its variants has generated dozens of papers; indeed, it would not be unreasonable to devote an entire chapter to this problem alone. Fortunately, much of the work on this problem took place from the 1960s through to the early 1990's, and this work is surveyed in an excellent article by Galil and Italiano [18]. In this section, we summarise the main results prior to 1991 but focus on developments since then.

We begin with by formulating the problem in a recent, more general way [19], that fits better in our framework. We start with the base repertoire, but now require that $\mathsf{insert}(x, A)$ is only permitted if $x \notin B$, for all sets $B \in \mathcal{F} - \{A\}$. This ensures that all sets in $\mathcal{F}$ are pairwise disjoint. We now add the following operations:

$\mathsf{dunion}(A, B, C)$ This sets $C \leftarrow A \cup B$, but destroys (removes from $\mathcal{F}$) A and B.

$\mathsf{find}(x)$ For any $x \in \cup_{A \in \mathcal{F}} A$, returns the name of the set that x is in.

This problem has a number of applications, beginning historically with the efficient implementation of EQUIVALENCE and COMMON statements in early FORTRAN compilers in the 1960s, and continues to find applications in diverse areas such as dynamic graph algorithms, meldable data structures and implementations of unification algorithms.

It is difficult to discuss the disjoint-set union-find problem without reference to *Ackermann's function*, which is defined for integers $i, n \geq 0$ as follows:

$$A(i,n) = \begin{cases} 2n & \text{for } i = 0, n \geq 0 \\ 1 & \text{for } i \geq 1, n = 0 \\ A(i-1, A(i, n-1)) & \text{for } i, n \geq 1 \end{cases}$$

For a fixed i, the *row* inverse of Ackermann's function, denoted by $a(i, n)$, is defined as:

$$a(i, n) = \min\{j \mid A(i, j) \geq n\}.$$

It can be verified that $a(1, n) = \Theta(\log n)$ and $a(2, n) = \Theta(\log^* n)$, where $\log^* n$ is the *iterated logarithm* function. The functional inverse of Ackermann's function is defined for $m, n \geq 1$ by:

$$\alpha(m, n) = \min\{i \geq 1 \mid A(i, 4\lceil m/n \rceil) \geq n\}$$

Since $A(3, 4) = 2^{2^{\cdot^{\cdot^{2}}}} \Big\} {}^{65,536 \text{ times}}$, for all practical purposes, $\alpha(m, n) \leq 3$ whenever $m, n \geq 1$. Indeed, if m grows sufficiently faster than n (e.g. $m = \Omega(n \log^* n)$) then $\alpha(m, n)$ is indeed a constant.

The union-find problem in the above formalisation was studied by [20], who gave the following results:

THEOREM 33.5

(i) An intermixed sequence of f find*, m* insert*, m* create*, $d \leq m$* delete *operations and at most $m-1$* dunion *operations takes $O(m+(f+d)\alpha(f+m,m))$ time. The size of the data structure at any time during the sequence is proportional to the number of live items in it.*

(ii) For any parameter $k>1$, an intermixed sequence of find*,* dunion*,* insert*,* create*, and* delete *operations can be processed in the following worst-case time bounds:* find *and* delete *in $O(\log m/\log k)$ time,* create *and* insert *in $O(1)$ time and* dunion *in $O(k)$ time, where m is the current number of elements in the data structure.*

In fact, [20] showed that this problem could be reduced to the classical union-find problem (see below). This reduction is such that the time bounds for find and dunion change only by a constant factor, but the time to delete an element x is the same as the time it takes to find the set containing x plus the time it takes to unite a singleton set with this set. The results follow by applying this result, respectively, to the classical union-find data structures given in Theorem 33.6(i) and Theorem 33.6(ii) (in fact they use a variant of the latter result due to Smid [37]).

An interesting variant of the above problem was considered by the same authors in [19]. They replaced the find operation with the seemingly simpler:

bfind(x, A) Return 'true' if $x \in A$ and 'false' otherwise.

This 'boolean' union-find problem was motivated by the problem of implementing *meldable* priority queues. Kaplan et al. showed that the lower bounds established for the classical union-find problem apply to this problem as well (see Theorem 33.7).

33.5.1 The Classical Union-Find Problem and Variants

In the classical version of this problem, we have $U = \{1,\ldots,m\}$ and $\mathcal{F}$ initially equals $\{\{1\},\{2\},\ldots,\{m\}\}$; we assume that the name of the set $\{i\}$ is simply i. The operation dunion is modified as follows:

dunion(A, B) This sets $A \leftarrow A \cup B$, but destroys (removes from $\mathcal{F}$) B.

The operation find(x) operates as described above. We now mention the main upper bounds that are known for the classical version of the problem:

THEOREM 33.6 *A sequence of $m-1$* dunion *and f* find *operations can be processed as follows:*

(i) in $O(m+f\alpha(f+m,m))$ time;

(ii) for any parameter $k>1$, each find *takes $O(\log m/(\log k+\log\log m))$ worst-case time and each* dunion *takes $O(k)$ worst-case time;*

(iii) the i-th find *operation takes $O(\alpha(i,m))$ worst-case time, and the entire sequence takes $O(m+f\alpha(f+m,m))$ time;*

(iv) for any parameter $k>0$, the entire sequence takes $O(ma(k,m)+kf)$ time;

(v) for any parameter $k>1$, dunion *takes $O(k)$ worst-case time,* find *takes $t_q = O(\log m/\log k)$ worst-case time, and the entire sequence takes time $O((m+f)(\alpha(m+f,m)+a(t_q,m))$ time.*

We begin by remarking that the bounds (i) and (ii) are in fact implied by Theorem 33.5, but are included here for completeness (see [18] for a more complete history). The upper bound of Theorem 33.5(i) is due to Tarjan [39] (with many interesting refinements due to [41]). Note that since $\alpha(f+m,m)$ is essentially constant, the running time is linear for all practical purposes. In (i), each dunion takes $O(1)$ worst-case time, but a single find may take $\Omega(\log m)$ time. Result (iii) allows us to process each find quickly, in essentially constant time, but an individual dunion could be quite slow and may in fact take $\Omega(m)$ time. The overall cost of the sequence remains as in (i), though. This was explicitly proved in [27], but the main ideas were already present in [16]. In (iv), the time for a dunion is traded off for find, and the overall running time remains essentially linear (at least for $k \geq 2$) [6].

In either (i) or (iii), although the overall cost of the operation sequence is very low, an individual operation could take $\Omega(\log m)$ or $\Omega(m)$ time respectively. If we wish to minimise the maximum cost of a single operation over the sequence, then choosing (say) $k = \Theta(\sqrt{\log m})$ in (ii), we get that no operation takes more than $O(\log m/\log\log m)$ time. However, the cost of the entire sequence increases to $\Omega(f \log m/\log\log m)$. This result is due to [8].

In [2], the authors ask the question whether one can combine the best features of the results (i) (or (iii)) and (ii). Except possibly for a few extreme cases, this is achieved in [2, Theorem 11]. The result presented there has some technical restrictions (e.g. the algorithm needs to know the value of m in advance), but the authors claim that these are not essential.

We now come to lower bounds for this problem, which are either proved in the pointer machine model or the cell probe model. In general the cell probe model is more powerful (since there are no restrictions on the way that memory is accessed), but the lower bounds are proven assuming that all memory locations can store only numbers of $O(\log m)$ bits. By contrast, the pointer machine model does not place any restrictions on the size of the numbers that can be stored at each node. In the pointer machine model, however, the algorithm is required to maintain the name of each set in distinct nodes, and an operation such as find(x) must traverse the graph of nodes from the node for x to the node that contains the name of the set that x is in. Lower bounds for the pointer machine model sometimes make the following *separability* assumption [40]:

> At any time during the computation, the contents of the memory can be partitioned into collections of records such that each collection corresponds to a currently existing set, and no record in one collection contains a pointer to a record in another collection.

We now give some of the known lower bounds:

THEOREM 33.7

(i) Any algorithm that processes a sequence of $m-1$ dunion *and f find operations must take $\Omega(m+f\alpha(f+m,m))$ steps on either the pointer machine model or the cell probe model with a word size of $O(\log m)$ bits.*

(ii) Any algorithm that processes a sequence of $m-1$ dunion *and f find operations must take $\Omega(\log m/\log\log m)$ steps for some operation, on either the separable pointer machine model or the cell probe model with a word size of $O(\log m)$ bits.*

The lower bound (i) for the pointer machine model is due to [28]; this generalizes an earlier lower bound for separable pointer machine algorithms due to [40] (see also the result due to [4]). The lower bound (ii) is due to [8], while both cell probe lower bounds are due to [15]. Additional lower bounds may be found in [2].

We refer the reader to [18] for variations of the classical union-find problem, including union-find with backtracking and persistent union-find (for the latter, see also [12, 13]). Another important variant, which has a similar complexity to that of the union-find problem, is the *split-find* problem. Here we are given an ordered universe U and start with $\mathcal{F}$ containing a single set which contains all of U. We perform an intermixed series of operations $S, T \leftarrow \textsf{split}(S, x)$ for some $x \in S$, which removes from S all elements $> x$ and places them in a new set T, and $\textsf{find}(x)$, which as before returns the name of the set containing x. Upper and lower bounds of the form given in Theorem 33.6(i) or 33.7(i) have been proven by [16, 27, 28]. Finally, we mention the very interesting *Union-copy* problem studied in [23]. In addition to the classical union-find operations, they support an additional operation copy, which takes one set as its argument, and adds an exact copy of it to $\mathcal{F}$. Although it is obviously no longer true that all sets in $\mathcal{F}$ are disjoint, they require that two sets that are to be united must be disjoint. In addition, they want the data structure to support *dual* operations, in the sense that the roles of sets and elements are reversed. For example, the operation $\textsf{element-union}(x, y)$ takes two elements x and y that do not both belong to any set in $\mathcal{F}$, and turns them into a single element that is member of all the sets to which x or y previously belonged. Duals of copy and find are also supported. They show applications of this data structure to the (geometric) dynamic segment tree data structure.

33.6 Partition Maintenance Algorithms

A partition P of a given universe $U = \{1, 2, \ldots, m\}$ is a collection of $\#P$ disjoint subsets (*parts*), $P^{(1)}, P^{(2)}, \ldots, P^{(\#P)}$, of U such that $\cup_i P_i = U$. A case of special interest is that of *bipartitions*, which is a partition with two parts. A partition P is an equivalence relation on U, which we denote as $\equiv_P$. Given two partitions P and Q, the *induced* partition of P and Q is the partition that represents the equivalence relation $x \equiv y \Leftrightarrow ((x \equiv_P y) \vee (x \equiv_Q y))$ (in words, two elements belong to the same part of the induced partition iff they belong to the same part in both P and Q). The problem we consider is the following. Given a collection $\mathcal{F}$ of partitions (initially empty), to support the following operations:

$$\left.\begin{array}{ll}\textsf{report}(\mathcal{F}) & \text{Report the partition induced by the members of } \mathcal{F}.\\ \textsf{insert}(P) & \text{Add partition } P \text{ to } \mathcal{F}.\\ \textsf{delete}(P) & \text{Remove partition } P \text{ from } \mathcal{F}.\end{array}\right\} \quad (33.7)$$

We assume here that each new partition P is given by an array $A[1..m]$ containing integers from 0 to $\#P - 1$. Other operations include asking if two elements belong to the same partition of the global induced partition, reporting the set of elements that belong to the same part of the global induced partition, or, for any two elements, reporting the number of partitions in which these two elements are in different parts.

As noted by Bender et al. [7] and Calinescu [9], this problem has a number of applications. The general issue is supporting a *classification system* that attempts to categorize objects based on a number of tests, where each test realizes a partition of the objects (a bipartition, for instance, could model the outcome of a boolean test). The above data structure would be useful in the pre-processing phase of such a classification system, where, for example, an algorithm may repeatedly insert and delete partitions (tests) until it finds a small set of tests that distinguish all items of the universe. Examples in optical character recognition (OCR) and VLSI are given by the above authors.

We now discuss some solutions to these problems. The model used is the word RAM model with word size $\Theta(\log n)$, where we take $k = |\mathcal{F}|$ and $n = km$. We first consider the case of general partitions, and then that of bipartitions. A simple data structure for general partitions was given by Bender et al. [7], and resembles the partition tree of Section 33.3. A key fact they use is (apply radix sorting on triples of the form $\langle P[i], Q[i], i\rangle$):

PROPOSITION 33.1 Given two partitions P and Q of $U = \{1, \ldots, m\}$ we can calculate the induced partition of P and Q in $O(m)$ time.

This immediately implies that we can maintain the induced partition of $\mathcal{F}$ under inserts alone in $O(m)$ time and also support report in $O(m)$ time—we simply maintain the global induced partition and update it with each insert. deletes are not so easy, however.

We place the partitions at the leaves of a full binary tree with k leaves. At each internal node v of this binary tree we store the partition induced by all the partitions descended from v. Clearly, the root contains the global induced partition. Inserting a partition P involves replacing a leaf containing a partition Q by an internal node that has P and Q as children, and updating the partitions stored at this internal node and all its ancestors. Deleting a partition involves deleting the appropriate leaf, and maintaining fullness by possibly swapping a 'rightmost' leaf with the deleted leaf. In either case, we update partitions from at most two leaves up to the root. This gives a data structure that supports insert and delete in $O(m \log k)$ time and report in $O(m)$ time. The space used is $O(mn)$ words of memory.

The time for insert can be improved to amortised $O(m)$, while leaving the time complexity of all other operations the same. The idea is to group all partitions into disjoint groups of $t = \lceil \log k \rceil$ each, leaving perhaps one incomplete group of size smaller than t. For each group we store its induced partition, and also store the induced partitions of all groups, except the incomplete group, at the leaves of a tree T (which may now have $O(k/\log k)$ leaves) as before. In addition, we explicitly store the global induced partition G.

When performing insert(P) we add P to the incomplete group and in $O(m)$ time, update the global partition G. If the incomplete group reaches size t, in addition, we calculate the group's induced partition in $O(mt) = O(m \log k)$ time and insert this induced partition into T, also taking $O(m \log k)$ time, and start a new empty incomplete group. Deleting a partition P is done as follows. We delete P from its group. If P is in the incomplete group we recompute the G in $O(mt)$ time. If P's group is stored in T, we recompute the new partition induced by P's group in $O(mt)$ time and update T in $O(m \log k)$ time (if P is the last remaining partition in its group we delete the corresponding leaf, as before). We then recompute G in $O(mt) = O(m \log k)$ time as well. Note that the amortised cost of an insertion is now $O(m)$, since we spend $O(m \log k)$ time every $\log k$ insertions. Finally, Bender et al. note that a signature-based scheme gives a Monte-Carlo method that performs all operations in $O(m)$ time, but has a small chance of outputting an incorrect result (the algorithm runs correctly with probability $1 - O(m^{-c})$ on all inputs).

We now consider the important special case of bi-partitions, and give a sketch of Calinescu's [9] algorithm for solving this problem in optimal amortised time. Again letting $k = |\mathcal{F}|$, one can associate a k-bit binary string σ_x with each $x \in U = \{1, \ldots, m\}$, which specifies in which part of each of the k partitions x lies. Let π denote a permutation such that $\sigma_{\pi^{-1}(i)} \leq \sigma_{\pi^{-1}(i+1)}$, for $i = 1, \ldots, m-1$; i.e., π represents a sorted order on the (multiset) of strings $\{\sigma_x \mid x \in U\}$. Furthermore, let lcp_i denote the most significant position where $\sigma_{\pi^{-1}(i)}$ and $\sigma_{\pi^{-1}(i+1)}$ differ, for $i = 1, \ldots, m-1$ ($lcp_i = k + 1$ if $\sigma_{\pi^{-1}(i)} = \sigma_{\pi^{-1}(i+1)}$). We can now clearly support report in $O(m)$ time, as elements in the same part of the global induced partition will be consecutive in sorted order, and the *lcp* values allow us to determine the boundaries of parts of the global induced partition without inspecting the strings. An insert of a partition is also easily supported, as only local re-arrangements of elements lying within the same part of the previous global induced partition are required.

Deleting a partition is a little harder, and requires a few observations. Suppose that we delete the partition that gives the t-th most significant bit of $\sigma_1, \ldots, \sigma_m$. Suppose that for two indices i, j, $j > i$, $lcp_i < t$ and $lcp_j < t$, but all *lcp*'s in between are at least t. Then we can conclude that the strings in positions 1 to $i-1$ of the sorted order continue to appear in (possibly a different order in) positions 1 to $i-1$ after the deletion of this partition, and likewise the strings in positions $j+1$ to m also do not 'move' except internally. Let Σ denote the strings that appear in positions i through j in sorted order, i.e., $\Sigma = \{\sigma_l \mid l \in \{\pi^{-1}(i), \pi^{-1}(i+1), \ldots, \pi^{-1}(j)\}\}$. We now show how to sort Σ, and repeated application of this procedure suffices to re-sort the array. Note that all the strings in Σ that have 0 in the t-th position maintain their relative order after the deletion, and likewise those strings with 1 in the t-th position. Thus, re-sorting Σ is simply a matter of merging these two sets of strings. At a high level, the merging procedure proceeds like the (standard, trivial) algorithm. However, a naive approach would require $\Theta(k)$ time per comparison, which is excessive. Instead, we note that at each step of the merging, the next candidate can either be determined in $O(1)$ time (when the relative order of the two candidates is implicit from the *lcp* data) or a number, c, of comparisons need to be made. However, if c comparisons are made, then there is at least one *lcp* value in the new array that is c more than its counterpart in the old array. Since *lcp* values are bounded by $O(k)$, no string will be involved in more than $O(k)$ comparisons during its lifetime, and the cost of these comparisons can be charged to the insertions of the partitions. This intuition can be formalized by a potential function argument to show that the amortised cost of a deletion is indeed $O(n)$, thus giving an optimal (amortised) algorithm for this problem.

33.7 Conclusions

We have presented a number of algorithms and data structures for supporting (largely) basic set-theoretic operations. Aside from the union-find problem, which has been extensively studied, relatively little research has been done into these problems. For instance, even the most basic problem, that of finding a general-purpose data structure that supports basic set operations, is not yet satisfactorily solved. The problem becomes more acute if one is concerned about the space usage of the data structures—for example, it is not known whether one can solve set equality testing efficiently in linear space.

Due to the proliferation of unstructured data, set operations are increasingly important. For instance, many search engines return a set of documents that match a given boolean keyword query by means of set operations on the sets of documents that contain each of the keywords in the query. The characteristics of this kind of application also suggest directions for research. For example, given the large data-sets that could be involved, it is a little surprising that work on external-memory algorithms for these problems is somewhat limited. Another issue is that these sets usually have patterns (e.g. the number of sets that contain a given keyword may satisfy a power law; certain sets of keywords may be more likely to be queried together etc.), which should be exploited by efficient algorithms.

With the latter motivation in mind, Demaine et al. [10] have considered the *adaptive* complexity of these problems. They assume that they are given a collection of sorted lists that comprise the sets, and need to compute unions, intersections and differences of these sets. If one is only interested in worst-case complexity (across all instances) then this problem is uninteresting (it essentially boils down to merging). However, some instances can be harder than others: for instance, computing the intersection of two sets when all elements in one set are smaller than the other is much easier than for sets that interleave substantially. Building on this idea, they develop a notion of the complexity of a given instance of a problem and develop algorithms that, for each particular instance, are efficient with respect to the difficulty of that instance.

References

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley, 1974.

[2] S. Alstrup, A. M. Ben-Amram and T. Rauhe. Worst-case and amortised optimality in union-find (extended abstract). *Proc. 31st Annual Symposium on Theory of Computing*, ACM, 1999, pp. 499–506.

[3] A. Andersson and T. Ottmann. New tight bounds on uniquely represented dictionaries. *SIAM Journal on Computing* **24**(1995), pp. 1091–1101.

[4] L. Banachowsky. A complement to Tarjan's result about the lower bound on the complexity of the set union problem. *Information Processing Letters*, **11** (1980), pp. 59–65.

[5] P. Beame and F. E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences* **65** (2002), pp. 38–72.

[6] A. M. Ben-Amram and Z. Galil. On data structure tradeoffs and an application to union-find. *Electronic Colloquium on Computational Complexity*, Technical Report TR95-062, 1995.

[7] M. A. Bender, S. Sethia and S. Skiena. Data structures for maintaining set partitions. Manuscript, 2004. Preliminary version in *Proc. 7th Scandinavian Workshop on Algorithm Theory*, Lecture Notes in Computer Science, Springer-Verlag, Berlin, vol. 1851, pp. 83–96, 2000.

[8] N. Blum. The single-operation worst case

[9] G. Calinescu. A note on data structures for maintaining partitions. Manuscript, 2003.

[10] E. D. Demaine, A. López-Ortiz and J. I. Munro. Adaptive Set Intersections, Unions and Differences. In *Proc. 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, ACM/SIAM, 2001, pp. 743–752.

[11] P. F. Dietz, K. Mehlhorn, R. Raman and C. Uhrig. Lower bounds for set intersection queries. *Algorithmica*, **14** (1995), pp. 154–168.

[12] P. F. Dietz and R. Raman. Persistence, amortisation and randomisation. In *Proc. 2nd Annual ACM-SIAM Symposium on Discrete Algorithms*, ACM/SIAM, pp. 77-87, 1991.

[13] P. F. Dietz and R. Raman. Persistence, randomization and parallelization: On some combinatorial games and their applications (Abstract). In *Proc. 3rd Workshop on Algorithms and Data Structures*, Lecture Notes in Computer Science, Springer-Verlag, vol. 709, pp. 289–301, 1993.

[14] M. L. Fredman. A lower bound on the complexity of orthogonal range queries. *Journal of the ACM*, **28** (1981), pp. 696–705.

[15] M. L. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st Annual ACM Symposium on Theory of Computing*, ACM, pp. 345–354, 1989.

[16] H. N. Gabow. A scaling algorithm for weighted matching on general graphs. In *Proc. 26th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society, pp. 90-100, 1985.

[17] T. Hagerup. Sorting and searching on the word RAM. In *Proc. 15th Symposium on Theoretical Aspects of Computer Science (STACS 1998)*, Lecture Notes in Computer Science, Springer-Verlag, Berlin, vol. 1373, pp. 366–398, 1998.

[18] Z. Galil and G. F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys*, **23** (1991), pp. 319–344.

[19] H. Kaplan, N. Shafrir and R. E. Tarjan. Meldable heaps and boolean union-find. In *Proc. 34th Annual ACM Symposium on Theory of Computing*, ACM, 2002, pp. 573–582.

[20] H. Kaplan, N. Shafrir and R. E. Tarjan. Union-find with deletions. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, ACM/SIAM, 2002, pp. 19–28.

[21] D. E. Knuth. *The Art of Computer Programming vol. 3*, 2nd Ed. Addison Wesley, 1998.

[22] A. N. Kolmogorov. On the notion of algorithm. *Uspekhi Matematicheskikh Nauk* **8** (1953), pp. 175–176.

[23] M. J. van Kreveld and M. H. Overmars. Union-copy data structures and dynamic segment trees. *Journal of the ACM*, **40** (1993), pp. 635–652.

[24] T. W. Lam and K. H. Lee. An improved scheme for set equality testing and updating. *Theoretical Computer Science*, **201** (1998), pp. 85–97.

[25] K. Mehlhorn. *Data Structures and Algorithms, Vol. III: Multi-dimensional Searching and Computational Geometry.* Springer-Verlag, Berlin, 1984.

[26] M. H. Overmars. *The Design of Dynamic Data Structures.* Lecture Notes in Computer Science, Springer-Verlag, Berlin, vol. 156, 1983.

[27] J. A. La Poutré. New techniques for the union-find problem. In *Proc. 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, ACM/SIAM, pp. 54–63, 1990.

[28] J. A. La Poutré. Lower bounds for the union-find and the split-find problem on pointer machines. *Journal of Computer and System Sciences*, **52** (1996), pp. 87–99.

[29] P. Pritchard. Opportunistic algorithms for eliminating supersets. *Acta Informatica* **28** (1991), pp. 733–754.

[30] P. Pritchard. A simple sub-quadratic algorithm for computing the subset partial order. *Information Processing Letters*, **56** (1995), pp. 337–341.

[31] P. Pritchard. An old sub-quadratic algorithm for finding extremal sets. *Information Processing Letters*, **62** (1997), pp. 329–334.

[32] P. Pritchard. A fast bit-parallel algorithm for computing the subset partial order. *Algorithmica*, **24** (1999), pp. 76–86.

[33] W. Pugh. *Incremental computation and the incremental evaluation of function programs.* PhD Thesis, Cornell University, 1989.

[34] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, ACM, 1989, pp. 315–328.

[35] A. Schönage. Storage modification machines. *SIAM Journal on Computing*, **9** (1980), pp. 490–508.

[36] R. Seidel and C. Aragon. Randomized search trees *Algorithmica*, **16** (1996), pp. 464–497.

[37] M. Smid. A data structure for the union-find problem having good single-operation complexity. In *Algorithms Review, Newsletter of the ESPRIT II Basic Research Action program project no. 3075*, **1**, ALCOM, 1990.

[38] R. Sundar and R. E. Tarjan. Unique binary-search-tree representations and equality testing of sets and sequences. *SIAM Journal on Computing*, **23** (1994), pp. 24–44.

[39] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the*

ACM **22** (1975), pp. 215–225.

[40] R. E. Tarjan. A class of algorithms which require non linear time to maintain disjoint sets. *Journal of Computer and System Sciences* **18** (1979), pp. 110–127.

[41] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM* **31** (1984), pp. 245–281.

[42] D. E. Willard. New trie data structures which support very fast search operations.

[43] A. C. Yao. On the complexity of maintaining partial sums. *SIAM Journal on Computing*, **14** (1985), pp. 277–288.

[44] D. M. Yellin. Representing sets with constant time equality testing. *Journal of Algorithms*, **13** (1992), pp. 353–373.

[45] D. M. Yellin. An algorithm for dynamic subset and intersection testing. *Theoretical Computer Science*, **129** (1994), pp. 397–406.

[46] D. M. Yellin and C. S. Jutla. Finding extremal sets in less than quadratic time. *Information Processing Letters*, **48** (1993), pp. 29–34.

34

Cache-Oblivious Data Structures

Lars Arge
Duke University

Gerth Stølting Brodal
University of Aarhus

Rolf Fagerberg
University of Southern Denmark

34.1 The Cache-Oblivious Model

The memory system of most modern computers consists of a hierarchy of memory levels, with each level acting as a cache for the next; for a typical desktop computer the hierarchy consists of registers, level 1 cache, level 2 cache, level 3 cache, main memory, and disk. One of the essential characteristics of the hierarchy is that the memory levels get larger and slower the further they get from the processor, with the access time increasing most dramatically between main memory and disk. Another characteristic is that data is moved between levels in large blocks. As a consequence of this, the memory access pattern of an algorithm has a major influence on its practical running time. Unfortunately, the RAM model (Figure 34.1) traditionally used to design and analyze algorithms is not capable of capturing this, since it assumes that all memory accesses take equal time.

Because of the shortcomings of the RAM model, a number of more realistic models have been proposed in recent years. The most successful of these models is the simple two-level I/O-model introduced by Aggarwal and Vitter [2] (Figure 34.2). In this model the memory hierarchy is assumed to consist of a fast memory of size M and a slower infinite memory, and data is transfered between the levels in blocks of B consecutive elements. Computation

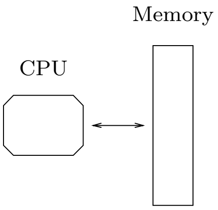


FIGURE 34.1: The RAM model.

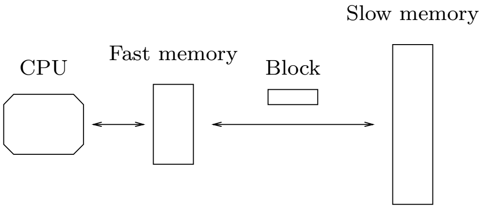


FIGURE 34.2: The I/O model.

can only be performed on data in the fast memory, and it is assumed that algorithms have complete control over transfers of blocks between the two levels. We denote such a transfer a *memory transfer*. The complexity measure is the number of memory transfers needed to solve a problem. The strength of the I/O model is that it captures part of the memory hierarchy, while being sufficiently simple to make design and analysis of algorithms feasible. In particular, it adequately models the situation where the memory transfers between two levels of the memory hierarchy dominate the running time, which is often the case when the size of the data exceeds the size of main memory. Agarwal and Vitter showed that comparison based sorting and searching require $\Theta(\mathrm{Sort}_{M,B}(N)) = \Theta(\frac{N}{B}\log_{M/B}\frac{N}{B})$ and $\Theta(\log_B N)$ memory transfers in the I/O-model, respectively [2]. Subsequently a large number of other results have been obtained in the model; see the surveys by Arge [4] and Vitter [27] for references. Also see Chapter 27.

More elaborate models of multi-level memory than the I/O-model have been proposed (see e.g. [27] for an overview) but these models have been less successful, mainly because of their complexity. A major shortcoming of the proposed models, including the I/O-model, have also been that they assume that the characteristics of the memory hierarchy (the level and block sizes) are known. Very recently however, the *cache-oblivious* model, which assumes no knowledge about the hierarchy, was introduced by Frigo et al. [20]. In essence, a cache-oblivious algorithm is an algorithm formulated in the RAM model but analyzed in the I/O model, with the analysis required to hold for any B and M. Memory transfers are assumed to be performed by an off-line optimal replacement strategy. The beauty of the cache-oblivious model is that since the I/O-model analysis holds for any block and memory size, it holds for *all* levels of a multi-level memory hierarchy (see [20] for details). In other words, by optimizing an algorithm to one unknown level of the memory hierarchy, it is optimized on all levels simultaneously. Thus the cache-oblivious model is effectively a way of modeling a complicated multi-level memory hierarchy using the simple two-level I/O-model.

Frigo et al. [20] described optimal $\Theta(\mathrm{Sort}_{M,B}(N))$ memory transfer cache-oblivious algorithms for matrix transposition, fast Fourier transform, and sorting; Prokop also described a static search tree obtaining the optimal $O(\log_B N)$ transfer search bound [24]. Subsequently, Bender et al. [11] described a cache-oblivious dynamic search trees with the same search cost, and simpler and improved cache-oblivious dynamic search trees were then developed by several authors [10, 12, 18, 25]. Cache-oblivious algorithms have also been developed for e.g. problems in computational geometry [1, 10, 15], for scanning dynamic sets [10], for layout of static trees [8], for partial persistence [10], and for a number of fundamental graph problems [5] using cache-oblivious priority queues [5, 16]. Most of these results make the so-called *tall cache assumption*, that is, they assume that $M > \Omega(B^2)$; we make the same assumption throughout this chapter.

Empirical investigations of the practical efficiency of cache-oblivious algorithms for sorting [19], searching [18, 23, 25] and matrix problems [20] have also been performed. The overall conclusion of these investigations is that cache-oblivious methods often outperform RAM algorithms, but not always as much as algorithms tuned to the specific memory hierarchy and problem size. On the other hand, cache-oblivious algorithms perform well on all levels of the memory hierarchy, and seem to be more robust to changing problem sizes than cache-aware algorithms.

In the rest of this chapter we describe some of the most fundamental and representative cache-oblivious data structure results. In Section 34.2 we discuss two fundamental primitives used to design cache-oblivious data structures. In Section 34.3 we describe two cache-oblivious dynamic search trees, and in Section 34.4 two priority queues. Finally, in Section 34.5 we discuss structures for 2-dimensional orthogonal range searching.

34.2 Fundamental Primitives

The most fundamental cache-oblivious primitive is scanning—scanning an array with N elements incurs $\Theta(\frac{N}{B})$ memory transfers for any value of B. Thus algorithms such as median finding and data structures such as stacks and queues that only rely on scanning are automatically cache-oblivious. In fact, the examples above are optimal in the cache-oblivious model. Other examples of algorithms that only rely on scanning include Quicksort and Mergesort. However, they are not asymptotically optimal in the cache-oblivious model since they use $O(\frac{N}{B}\log\frac{N}{M})$ memory transfers rather than $\Theta(\mathrm{Sort}_{M,B}(N))$.

Apart from algorithms and data structures that only utilize scanning, most cache-oblivious results use recursion to obtain efficiency; in almost all cases, the sizes of the recursive problems decrease double-exponentially. In this section we describe two of the most fundamental such recursive schemes, namely the *van Emde Boas layout* and the *k-merger*.

34.2.1 Van Emde Boas Layout

One of the most fundamental data structures in the I/O-model is the B-tree [7]. A B-tree is basically a fanout $\Theta(B)$ tree with all leaves on the same level. Since it has height $O(\log_B N)$ and each node can be accessed in $O(1)$ memory transfers, it supports searches in $O(\log_B N)$ memory transfers. It also supports range queries, that is, the reporting of all K elements in a given query range, in $O(\log_B N + \frac{K}{B})$ memory transfers. Since B is an integral part of the definition of the structure, it seems challenging to develop a cache-oblivious B-tree structure. However, Prokop [24] showed how a binary tree can be laid out in memory in order to obtain a (static) cache-oblivious version of a B-tree. The main idea is to use a recursively defined layout called the *van Emde Boas layout* closely related to the definition of a van Emde Boas tree [26]. The layout has been used as a basic building block of most cache-oblivious search structures (e.g in [1, 8, 10–12, 18, 25]).

Layout

For simplicity, we only consider complete binary trees. A binary tree is complete if it has $N = 2^h - 1$ nodes and height h for some integer h. The basic idea in the van Emde Boas layout of a complete binary tree $\mathcal{T}$ with N leaves is to divide $\mathcal{T}$ at the middle level and lay out the pieces recursively (Figure 34.3). More precisely, if $\mathcal{T}$ only has one node it is simply laid out as a single node in memory. Otherwise, we define the *top tree* $\mathcal{T}_0$ to be the subtree consisting of the nodes in the topmost $\lfloor h/2 \rfloor$ levels of $\mathcal{T}$, and the *bottom trees* $\mathcal{T}_1, \ldots, \mathcal{T}_k$ to be the $\Theta(\sqrt{N})$ subtrees rooted in the nodes on level $\lceil h/2 \rceil$ of $\mathcal{T}$; note that all the subtrees have size $\Theta(\sqrt{N})$. The van Emde Boas layout of $\mathcal{T}$ consists of the van Emde Boas layout of $\mathcal{T}_0$ followed by the van Emde Boas layouts of $\mathcal{T}_1, \ldots, \mathcal{T}_k$.

Search

To analyze the number of memory transfers needed to perform a search in $\mathcal{T}$, that is, traverse a root-leaf path, we consider the first recursive level of the van Emde Boas layout where the subtrees are smaller than B. As this level $\mathcal{T}$ is divided into a set of *base trees* of size between $\Theta(\sqrt{B})$ and $\Theta(B)$, that is, of height $\Omega(\log B)$ (Figure 34.4). By the definition of the layout, each base tree is stored in $O(B)$ contiguous memory locations and can thus be accessed in $O(1)$ memory transfers. That the search is performed in $O(\log_B N)$ memory transfers then follows since the search path traverses $O((\log N)/\log B) = O(\log_B N)$ different base trees.

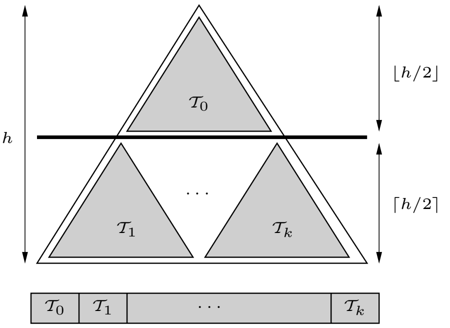


FIGURE 34.3: The van Emde Boas layout.

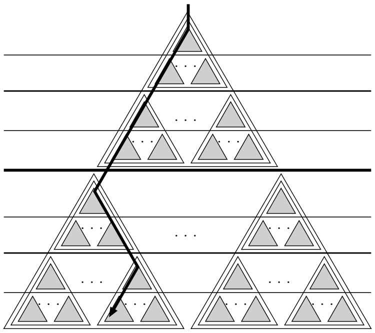


FIGURE 34.4: A search path.

Range query

To analyze the number of memory transfers needed to answer a range query $[x_1, x_2]$ on $\mathcal{T}$ using the standard recursive algorithm that traverses the relevant parts of $\mathcal{T}$ (starting at the root), we first note that the two paths to x_1 and x_2 are traversed in $O(\log_B N)$ memory transfers. Next we consider traversed nodes v that are not on the two paths to x_1 and x_2. Since all elements in the subtree $\mathcal{T}_v$ rooted at such a node v are reported, and since a subtree of height $\log B$ stores $\Theta(B)$ elements, $O(\frac{K}{B})$ subtrees $\mathcal{T}_v$ of height $\log B$ are visited. This in turn means that the number of visited nodes above the last $\log B$ levels of $\mathcal{T}$ is also $O(\frac{K}{B})$; thus they can all be accessed in $O(\frac{K}{B})$ memory transfers. Consider the smallest recursive level of the van Emde Boas layout that completely contain $\mathcal{T}_v$. This level is of size between $\Omega(B)$ and $O(B^2)$ (Figure 34.5(a)). On the next level of recursion $\mathcal{T}_v$ is broken into a top part and $O(\sqrt{B})$ bottom parts of size between $\Omega(\sqrt{B})$ and $O(B)$ each (Figure 34.5(b)). The top part is contained in a recursive level of size $O(B)$ and is thus stored within $O(B)$ consecutive memory locations; therefore it can be accessed in $O(1)$ memory accesses. Similarly, the $O(B)$ nodes in the $O(\sqrt{B})$ bottom parts are stored consecutively in memory; therefore they can all be accessed in a total of $O(1)$ memory transfers. Therefore, the optimal paging strategy can ensure that any traversal of $\mathcal{T}_v$ is performed in $O(1)$ memory transfers, simply by accessing the relevant $O(1)$ blocks. Thus overall a range query is performed in $O(\log_B N + \frac{K}{B})$ memory transfers.

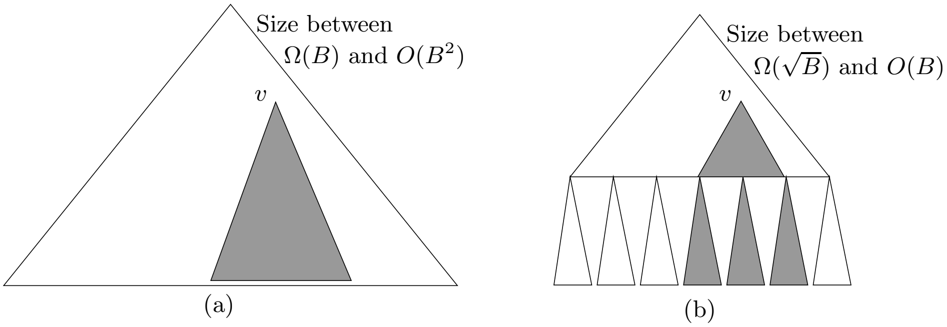


FIGURE 34.5: Traversing tree $\mathcal{T}_v$ with $O(B)$ leaves; (a) smallest recursive van Emde Boas level containing $\mathcal{T}_v$ has size between $\Omega(B)$ and $O(B^2)$; (b) next level in recursive subdivision.

THEOREM 34.1 *Let $\mathcal{T}$ be a complete binary tree with N leaves laid out using the van Emde Boas layout. The number of memory transfers needed to perform a search (traverse a root-to-leaf path) and a range query in $\mathcal{T}$ is $O(\log_B N)$ and $O(\log_B N + \frac{K}{B})$, respectively.*

Note that the navigation from node to node in the van Emde Boas layout is straightforward if the tree is implemented using pointers. However, navigation using arithmetic on array indexes is also possible [18]. This avoids the use of pointers and hence saves space.

The constant in the $O(\log_B N)$ bound for searching in Theorem 34.1 can be seen to be four. Further investigations of which constants are possible for cache-oblivious comparison based searching appear in [9].

34.2.2 k-Merger

In the I/O-model the two basic optimal sorting algorithms are multi-way versions of Mergesort and distribution sorting (Quicksort) [2]. Similarly, Frigo et al. [20] showed how both merge based and distribution based optimal cache-oblivious sorting algorithms can be developed. The merging based algorithm, *Funnelsort*, is based on a so-called *k-merger*. This structure has been used as a basic building block in several cache-oblivious algorithms. Here we describe a simplified version of the k-merger due to Brodal and Fagerberg [15].

Binary mergers and merge trees

A *binary merger* merges two sorted input streams into a sorted output stream: In one merge step an element is moved from the head of one of the input streams to the tail of the output stream; the heads of the input streams, as well as the tail of the output stream, reside in *buffers* of a limited capacity.

Binary mergers can be combined to form *binary merge trees* by letting the output buffer of one merger be the input buffer of another—in other words, a binary merge tree is a binary tree with mergers at the nodes and buffers at the edges, and it is used to merge a set of sorted input streams (at the leaves) into one sorted output stream (at the root). Refer to Figure 34.6 for an example.

An *invocation* of a binary merger in a binary merge tree is a recursive procedure that performs merge steps until the output buffer is full (or both input streams are exhausted); if

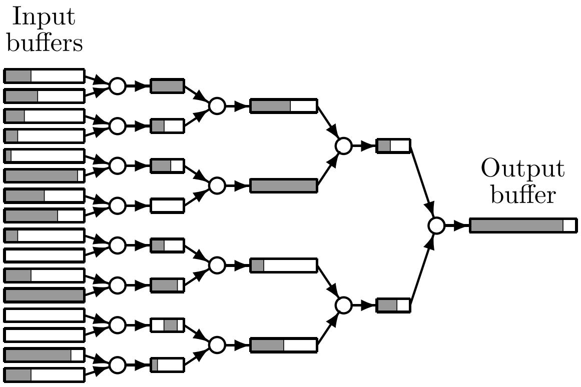


FIGURE 34.6: A 16-merger consisting of 15 binary mergers. Shaded parts represent elements in buffers.

```
Procedure FILL($v$)
    while $v$'s output buffer is not full
        if left input buffer empty
            FILL(left child of $v$)
        if right input buffer empty
            FILL(right child of $v$)
        perform one merge step
```

FIGURE 34.7: Invocation of binary merger v.

an input buffer becomes empty during the invocation (and the corresponding stream is not exhausted), the input buffer is recursively filled by an invocation of the merger having this buffer as output buffer. If both input streams of a merger get exhausted, the corresponding output stream is marked as exhausted. A procedure FILL(v) performing an invocation of a binary merger v is shown in Figure 34.7 (ignoring exhaustion issues). A single invocation FILL(r) on the root r of a merge tree will merge the streams at the leaves of the tree.

k-merger

A k-merger is a binary merge tree with specific buffer sizes. For simplicity, we assume that k is a power of two, in which case a k-merger is a complete binary tree of $k-1$ binary mergers. The output buffer at the root has size k^3, and the sizes of the rest of the buffers are defined recursively in a manner resembling the definition of the van Emde Boas layout: Let $i = \log k$ be the height of the k-merger. We define the *top tree* to be the subtree consisting of all mergers of depth at most $\lceil i/2 \rceil$, and the *bottom trees* to be the subtrees rooted in nodes at depth $\lceil i/2 \rceil + 1$. We let the edges between the top and bottom trees have buffers of size $k^{3/2}$, and define the sizes of the remaining buffers by recursion on the top and bottom trees. The input buffers at the leaves hold the input streams and are not part of the k-merger definition. The space required by a k-merger, excluding the output buffer at the root, is given by $S(k) = k^{1/2} \cdot k^{3/2} + (k^{1/2}+1) \cdot S(k^{1/2})$, which has the solution $S(k) = \Theta(k^2)$.

We now analyze the number of memory transfers needed to fill the output buffer of size k^3 at the root of a k-merger. In the recursive definition of the buffer sizes in the k-merger, consider the first level where the subtrees (excluding output buffers) have size less than $M/3$; if $\bar{k}$ is the number of leaves of one such subtree, we by the space usage of k-mergers have $\bar{k}^2 \leq M/3$ and $(\bar{k}^2)^2 = \bar{k}^4 = \Omega(M)$. We call these subtrees of the k-merger *base trees* and the buffers between the base trees *large buffers*. Assuming $B^2 \leq M/3$, a base tree $\mathcal{T}_v$ rooted in v together with one block from each of the large buffers surrounding it (i.e., its single output buffer and $\bar{k}$ input buffers) can be contained in fast memory, since $M/3 + B + \bar{k} \cdot B \leq M/3 + B + (M/3)^{1/2} \cdot (M/3)^{1/2} \leq M$. If the k-merger consists of a single base tree, the number of memory transfers used to fill its output buffer with k^3 elements during an invocation is trivially $O(k^3/B + k)$. Otherwise, consider an invocation of the root v of a base tree $\mathcal{T}_v$, which will fill up the size $\Omega(\bar{k}^3)$ output buffer of v. Loading $\mathcal{T}_v$ and one block for each of the $\bar{k}$ buffers just below it into fast memory will incur $O(\bar{k}^2/B + \bar{k})$ memory transfers. This is $O(1/B)$ memory transfer for each of the $\Omega(\bar{k}^3)$ elements output, since $\bar{k}^4 = \Omega(M)$ implies $\bar{k}^2 = \Omega(M^{1/2}) = \Omega(B)$, from which $\bar{k} = O(\bar{k}^3/B)$ follows. Provided that none of the input buffers just below $\mathcal{T}_v$ become empty, the output buffer can then be filled in $O(\bar{k}^3/B)$ memory transfers since elements can be read from the input buffers in

$O(1/B)$ transfers amortized. If a buffer below $\mathcal{T}_v$ becomes empty, a recursive invocation is needed. This invocation may evict $\mathcal{T}_v$ from memory, leading to its reloading when the invocation finishes. We charge this cost to the $\Omega(\bar{k}^3)$ elements in the filled buffer, or $O(1/B)$ memory transfers per element. Finally, the last time an invocation is used to fill a particular buffer, the buffer may not be completely filled (due to exhaustion). However, this happens only once for each buffer, so we can pay the cost by charging $O(1/B)$ memory transfers to each position in each buffer in the k-merger. As the entire k-merger uses $O(k^2)$ space and merges k^3 elements, these charges add up to $O(1/B)$ memory transfers per element.

We charge an element $O(1/B)$ memory transfers each time it is inserted into a large buffer. Since $\bar{k} = \Omega(M^{1/4})$, each element is inserted in $O(\log_{\bar{k}} k) = O(\log_M k^3)$ large buffers. Thus we have the following.

THEOREM 34.2 *Excluding the output buffers, the size of a k-merger is $O(k^2)$ and it performs $O(\frac{k^3}{B}\log_M k^3 + k)$ memory transfers during an invocation to fill up its output buffer of size k^3.*

Funnelsort

The cache-oblivious sorting algorithm Funnelsort is easily obtained once the k-merger structure is defined: Funnelsort breaks the N input elements into $N^{1/3}$ groups of size $N^{2/3}$, sorts them recursively, and then merges the sorted groups using an $N^{1/3}$-merger.

Funnelsort can be analyzed as follows: Since the space usage of a k-merger is sub-linear in its output, the elements in a recursive sort of size $M/3$ only need to be loaded into memory once during the entire following recursive sort. For k-mergers at the remaining higher levels in the recursion tree, we have $k^3 \geq M/3 \geq B^2$, which implies $k^2 \geq B^{4/3} > B$ and hence $k^3/B > k$. By Theorem 34.2, the number of memory transfers during a merge involving N' elements is then $O(\log_M(N')/B)$ per element. Hence, the total number of memory transfers per element is

$$O\left(\frac{1}{B}\left(1+\sum_{i=0}^{\infty}\log_M N^{(2/3)^i}\right)\right) = O\left((\log_M N)/B\right) .$$

Since $\log_M x = \Theta(\log_{M/B} x)$ when $B^2 \leq M/3$, we have the following theorem.

THEOREM 34.3 *Funnelsort sorts N element using $O(\mathrm{Sort}_{M,B}(N))$ memory transfers.*

In the above analysis, the exact (tall cache) assumption on the size of the fast memory is $B^2 \leq M/3$. In [15] it is shown how to generalize Funnelsort such that it works under the weaker assumption $B^{1+\varepsilon} \leq M$, for fixed $\varepsilon > 0$. The resulting algorithm incurs the optimal $O(\mathrm{Sort}_{M,B}(N))$ memory transfers when $B^{1+\varepsilon} = M$, at the price of incurring $O(\frac{1}{\varepsilon}\cdot\mathrm{Sort}_{M,B}(N))$ memory transfers when $B^2 \leq M$. It is shown in [17] that this trade-off is the best possible for comparison based cache-oblivious sorting.

34.3 Dynamic B-Trees

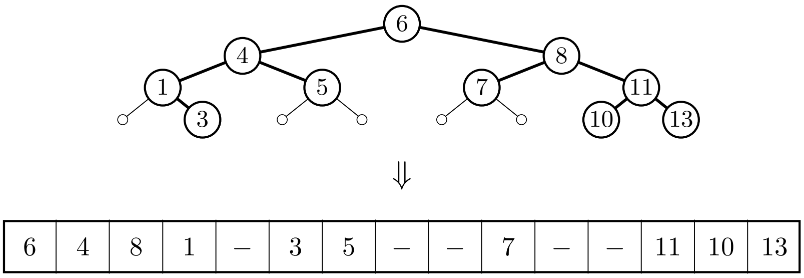
The van Emde Boas layout of a binary tree provides a static cache-oblivious version of B-trees. The first dynamic solution was given Bender et al. [11], and later several simplified structures were developed [10, 12, 18, 25]. In this section, we describe two of these structures [10, 18].

34.3.1 Density Based

In this section we describe the dynamic cache-oblivious search tree structure of Brodal et al. [18]. A similar proposal was given independently by Bender et al. [12].

The basic idea in the structure is to embed a dynamic binary tree of height $\log N + O(1)$ into a static complete binary tree, that is, in a tree with $2^h - 1$ nodes and height h, which in turn is embedded into an array using the van Emde Boas layout. Refer to Figure 34.8.

To maintain the dynamic tree we use techniques for maintaining small height in a binary tree developed by Andersson and Lai [3]; in a different setting, similar techniques has also been given by Itai et al. [21]. These techniques give an algorithm for maintaining height $\log N + O(1)$ using amortized $O(\log^2 N)$ time per update. If the height bound is violated after performing an update in a leaf l, this algorithm performs rebalancing by rebuilding the subtree rooted at a specific node v on the search path from the root to l. The subtree is rebuilt to perfect balance in time linear in the size of the subtree. In a binary tree of perfect balance the element in any node v is the median of all the elements stored in the subtree $\mathcal{T}_v$ rooted in v. This implies that only the lowest level in $\mathcal{T}_v$ is not completely filled and the empty positions appearing at this level are evenly distributed across the level. Hence, the net effect of the rebuilding is to redistribute the empty positions in $\mathcal{T}_v$. Note that this can lower the cost of future insertions in $\mathcal{T}_v$, and consequently it may in the long run be better to rebuild a subtree larger than strictly necessary for reestablishment of the height bound. The criterion for choosing how large a subtree to rebuild, i.e. for choosing the node v, is the crucial part of the algorithms by Andersson and Lai [3] and Itai et al. [21]. Below we give the details of how they can be used in the cache-oblivious setting.



6	4	8	1	–	3	5	–	–	7	–	–	11	10	13

FIGURE 34.8: Illustration of embedding a height H tree into a complete static tree of height H, and the van Emde Boas layout of this tree.

Structure

As mentioned, the data structure consists of a dynamic binary tree $\mathcal{T}$ embedded into a static complete binary tree $\mathcal{T}'$ of height H, which in turn is embedded into an array using the van Emde Boas layout.

In order to present the update and query algorithms, we define the *density* $\rho(u)$ of a node u as $|T_u|/|T'_u|$, where $|T_u|$ and $|T'_u|$ are the number of nodes in the trees rooted in u in $\mathcal{T}$ and $\mathcal{T}'$, respectively. In Figure 34.8, the node containing the element 4 has balance 4/7. We also define two *density thresholds* τ_i and γ_i for the nodes on each level $i = 1, 2, \ldots, H$ (where the root is at level 1). The upper density thresholds τ_i are evenly space values between 3/4 and 1, and the lower density thresholds γ_i are evenly spaced values between 1/4 and 1/8. More precisely, $\tau_i = 3/4 + (i-1)/(4(H-1))$ and $\gamma_i = 1/4 - (i-1)/(8(H-1))$.

Updates

To insert a new element into the structure we first locate the position in $\mathcal{T}$ of the new node w. If the insertion of w violates the height bound H, we rebalance $\mathcal{T}$ as follows: First we find the lowest ancestor v of w satisfying $\gamma_i \leq \rho(v) \leq \tau_i$, where i is the level of v. If no ancestor v satisfies the requirement, we rebuild the entire structure, that is, $\mathcal{T}$, $\mathcal{T}'$ and the layout of $\mathcal{T}'$: For k the integer such that $2^k \leq N < 2^{k+1}$ we choose the new height H of the tree $\mathcal{T}'$ as $k+1$ if $N \leq 5/4 \cdot 2^k$; otherwise we choose $H = k+2$. On the other hand, if the ancestor v exists we rebuild $\mathcal{T}_v$: We first create a sorted list of all elements in $\mathcal{T}_v$ by an in-order traversal of $\mathcal{T}_v$. The $\lceil |T_v|/2 \rceil$th element becomes the element stored at v, the smallest $\lfloor (|T_v|-1)/2 \rfloor$ elements are recursively distributed in the left subtree of v, and the largest $\lceil (|T_v|-1)/2 \rceil$ elements are recursively distributed in the right subtree of v.

We can delete an element from the structure in a similar way: We first locate the node w in $\mathcal{T}$ containing the element e to be deleted. If w is not a leaf and has a right subtree, we then locate the node w' containing the immediate successor of e (the node reached by following left children in the right subtree of w), swap the elements in w and w', and let $w = w'$. We repeat this until w is a leaf. If on the other hand w is not a leaf but only has a left subtree, we instead repeatedly swap w with the node containing the predecessor of e. Finally, we delete the leaf w from $\mathcal{T}$, and rebalance the tree by rebuilding the subtree rooted at the lowest ancestor v of w satisfying satisfying $\gamma_i \leq \rho(v) \leq \tau_i$, where i is the level of v; if no such node exists we rebuild the entire structure completely.

Similar to the proof of Andersson and Lai [3] and Itai et al. [21] that updates are performed in $O(\log^2 N)$ time, Brodal et al. [18] showed that using the above algorithms, updates can be performed in amortized $O(\log_B N + (\log^2 N)/B)$ memory transfers.

Range queries

In Section 34.2, we discussed how a range query can be answered in $O(\log_B N + \frac{K}{B})$ memory transfers on a complete tree $\mathcal{T}'$ laid out using the van Emde Boas layout. Since it can be shown that the above update algorithm maintains a lower density threshold of 1/8 for all nodes, we can also perform range queries in $\mathcal{T}$ efficiently: To answer a range query $[x_1, x_2]$ we traverse the two paths to x_1 and x_2 in $\mathcal{T}$, as well as $O(\log N)$ subtrees rooted in children of nodes on these paths. Traversing one subtree $\mathcal{T}_v$ in $\mathcal{T}$ incurs at most the number of memory transfers needed to traverse the corresponding (full) subtree $\mathcal{T}'_v$ in $\mathcal{T}'$. By the lower density threshold of 1/8 we know that the size of $\mathcal{T}'_v$ is at most a factor of eight larger than the size of $\mathcal{T}_v$. Thus a range query is answered in $O(\log_B N + \frac{K}{B})$ memory transfers.

THEOREM 34.4 *There exists a linear size cache-oblivious data structure for storing N elements, such that updates can be performed in amortized $O(\log_B N + (\log^2 N)/B)$ memory transfers and range queries in $O(\log_B N + \frac{K}{B})$ memory transfers.*

Using the method for moving between nodes in a van Emde Boas layout using arithmetic on the node indices rather than pointers, the above data structure can be implemented as a single size $O(N)$ array of data elements. The amortized complexity of updates can also be lowered to $O(\log_B N)$ by changing leaves into pointers to buckets containing $\Theta(\log N)$ elements each. With this modification a search can still be performed in $O(\log_B N)$ memory transfers. However, then range queries cannot be answered efficiently, since the $O(\frac{K}{\log N})$ buckets can reside in arbitrary positions in memory.

34.3.2 Exponential Tree Based

The second dynamic cache-oblivious search tree we consider is based on the so-called *exponential layout* of Bender et al. [10]. For simplicity, we here describe the structure slightly differently than in [10].

Structure

Consider a complete balanced binary tree $\mathcal{T}$ with N leaves. Intuitively, the idea in an exponential layout of $\mathcal{T}$ is to recursively decompose $\mathcal{T}$ into a set of *components*, which are each laid out using the van Emde Boas layout. More precisely, we define component $\mathcal{C}_0$ to consist of the first $\frac{1}{2}\log N$ levels of $\mathcal{T}$. The component $\mathcal{C}_0$ contains $\sqrt{N}$ nodes and is called an N-component because its root is the root of a tree with N leaves (that is, $\mathcal{T}$). To obtain the exponential layout of $\mathcal{T}$, we first store $\mathcal{C}_0$ using the van Emde Boas layout, followed immediately by the recursive layout of the $\sqrt{N}$ subtrees, $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_{\sqrt{N}}$, of size $\sqrt{N}$, beneath $\mathcal{C}_0$ in $\mathcal{T}$, ordered from left to right. Note how the definition of the exponential layout naturally defines a decomposition of $\mathcal{T}$ into $\log\log N + O(1)$ *layers*, with layer i consisting of a number of $N^{1/2^{i-1}}$-components. An X-component is of size $\Theta(\sqrt{X})$ and its $\Theta(\sqrt{X})$ leaves are connected to $\sqrt{X}$-components. Thus the root of an X-component is the root of a tree containing X elements. Refer to Figure 34.9. Since the described layout of $\mathcal{T}$ is really identical to the van Emde Boas layout, it follows immediately that it uses linear space and that a root-to-leaf path can be traversed in $O(\log_B N)$ memory transfers.

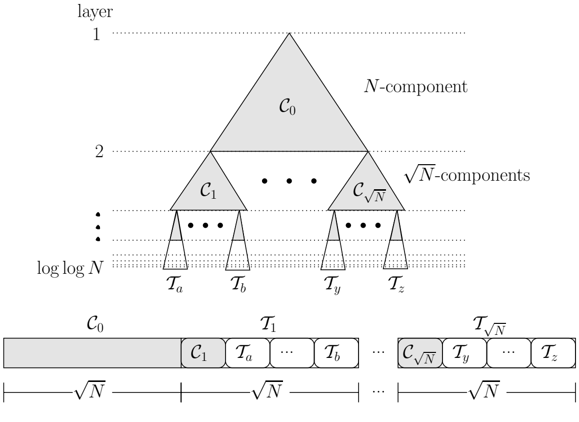


FIGURE 34.9: Components and exponential layout.

By slightly relaxing the requirements on the layout described above, we are able to maintain it dynamically: We define an *exponential layout* of a balanced binary tree $\mathcal{T}$ with N leaves to consist of a composition of $\mathcal{T}$ into $\log\log N + O(1)$ layers, with layer i consisting of a number of $N^{1/2^{i-1}}$-components, each laid out using the van Emde Boas layout (Figure 34.9). An X-component has size $\Theta(\sqrt{X})$ but unlike above we allow its root to be root in a tree containing between X and $2X$ elements. Note how this means that an X-component has between $X/2\sqrt{X} = \frac{1}{2}\sqrt{X}$ and $2X/\sqrt{X} = 2\sqrt{X}$ leaves. We store the layout of $\mathcal{T}$ in memory almost as previously: If the root of $\mathcal{T}$ is root in an X-component $\mathcal{C}_0$, we store $\mathcal{C}_0$ first in $2 \cdot 2\sqrt{X} - 1$ memory locations (the maximal size of an X-component), followed immediately by the layouts of the subtrees ($\sqrt{X}$-components) rooted in the leaves of $\mathcal{C}_0$ (in no particular order). We make room in the layout for the at most $2\sqrt{X}$ such subtrees. This exponential layout for $\mathcal{T}$ uses $S(N) = \Theta(\sqrt{N}) + 2\sqrt{N} \cdot S(\sqrt{N})$ space, which is $\Theta(N \log N)$.

Search

Even with the modified definition of the exponential layout, we can traverse any root-to-leaf path in $\mathcal{T}$ in $O(\log_B N)$ memory transfers: The path passes through exactly one $N^{1/2^{i-1}}$-component for $1 \leq i \leq \log\log N + O(1)$. Each X-component is stored in a van Emde Boas layout of size $\Theta(\sqrt{X})$ and can therefore be traversed in $\Theta(\log_B \sqrt{X})$ memory transfers (Theorem 34.1). Thus, if we use at least one memory transfer in each component, we perform a search in $O(\log_B N) + \log\log N$ memory accesses. However, we do not actually use a memory transfer for each of the $\log\log N + O(1)$ components: Consider the traversed X-component with $\sqrt{B} \leq X \leq B$. This component is of size $O(\sqrt{B})$ and can therefore be loaded in $O(1)$ memory transfers. All smaller traversed components are of total size $O(\sqrt{B} \log \sqrt{B}) = O(B)$, and since they are stored in consecutively memory locations they can also be traversed in $O(1)$ memory transfers. Therefore only $O(1)$ memory transfers are used to traverse the last $\log\log B - O(1)$ components. Thus, the total cost of traversing a root-to-leaf path is $O(\log_B N + \log\log N - \log\log B) = O(\log_B N)$.

Updates

To perform an insertion in $\mathcal{T}$ we first search for the leaf l where we want to perform the insertion; inserting the new element below l will increase the number of elements stored below each of the $\log\log N + O(1)$ components on the path to the root, and may thus result in several components needing *rebalancing* (an X-component with $2X$ elements stored below it). We perform the insertion and rebalance the tree in a simple way as follows: We find the topmost X-component $\mathcal{C}_j$ on the path to the root with $2X$ elements below it. Then we divide these elements into two groups of X elements and store them separately in the exponential layout (effectively we *split* $\mathcal{C}_j$ with $2X$ elements below it into two X-components with X elements each). This can easily be done in $O(X)$ memory transfers. Finally, we update a leaf and insert a new leaf in the X^2-component above $\mathcal{C}_j$ (corresponding to the two new X-components); we can easily do so in $O(X)$ memory transfers by rebuilding it. Thus overall we have performed the insertion and rebalancing in $O(X)$ memory transfers. The rebuilding guarantees that after rebuilding an X-component, X inserts have to be performed below it before it needs rebalancing again. Therefore we can charge the $O(X)$ cost to the X insertions that occurred below $\mathcal{C}_j$ since it was last rebuilt, and argue that each insertion is charged $O(1)$ memory accesses on each of the $\log\log N + O(1)$ levels. In fact, using the same argument as above for the searching cost, we can argue that we only need to charge an insertion $O(1)$ transfers on the last $\log\log B - O(1)$ levels of $\mathcal{T}$, since rebalancing on any of these levels can always be performed in $O(1)$ memory transfers. Thus overall we perform an insertion in $O(\log_B N)$ memory transfers amortized.

Deletions can easily be handled in $O(\log_B N)$ memory transfers using global rebuilding: To delete the element in a leaf l of $\mathcal{T}$ we simply mark l as deleted. If l's sibling is also marked as deleted, we mark their parent deleted too; we continue this process along one path to the root of $\mathcal{T}$. This way we can still perform searches in $O(\log_B N)$ memory transfers, as long as we have only deleted a fraction of the elements in the tree. After $\frac{N}{2}$ deletes we therefore rebuild the entire structure in $O(N \log_B N)$ memory accesses, or $O(\log_B N)$ accesses per delete operation.

Bender et al. [10] showed how to modify the update algorithms to perform updates "lazily" and obtain worst case $O(\log_B N)$ bounds.

Reducing space usage

To reduce the space of the layout of a tree $\mathcal{T}$ to linear we simply make room for $2 \log N$ elements in each leaf, and maintain that a leaf contains between $\log N$ and $2 \log N$ elements. This does not increase the $O(\log_B N)$ search and update costs since the $O(\log N)$ elements in a leaf can be scanned in $O((\log N)/B) = O(\log_B N)$ memory accesses. However, it reduces the number of elements stored in the exponential layout to $O(N/\log N)$.

THEOREM 34.5 *The exponential layout of a search tree $\mathcal{T}$ on N elements uses linear space and supports updates in $O(\log_B N)$ memory accesses and searches in $O(\log_B N)$ memory accesses.*

Note that the analogue of Theorem 34.1 does not hold for the exponential layout, i.e. it does not support efficient range queries. The reason is partly that the $\sqrt{X}$-components below an X-component are not located in (sorted) order in memory because components are rebalanced by splitting, and partly because of the leaves containing $\Theta(\log N)$ elements. However, Bender et al [10] showed how the exponential layout can be used to obtain a number of other important results: The structure as described above can easily be extended such that if two subsequent searched are separated by d elements, then the second search can be performed in $O(\log^* d + \log_B d)$ memory transfers. It can also be extended such that R queries (*batched searching*) can be answered simultaneously in $O(R \log_B \frac{N}{R} + \mathrm{Sort}_{M,B}(R))$ memory transfers. The exponential layout can also be used to develop a *persistent B-tree*, where updates can be performed in the current version of the structure and queries can be performed in the current as well as all previous versions, with both operations incurring $O(\log_B N)$ memory transfers. It can also be used as a basic building block in a linear space *planar point location* structure that answers queries in $O(\log_B N)$ memory transfers.

34.4 Priority Queues

A priority queue maintains a set of elements with a priority (or key) each under the operations INSERT and DELETEMIN, where an INSERT operation inserts a new element in the queue, and a DELETEMIN operation finds and deletes the element with the minimum key in the queue. Frequently we also consider a DELETE operation, which deletes an element with a given key from the priority queue. This operation can easily be supported using INSERT and DELETEMIN: To perform a DELETE we insert a special delete-element in the queue with the relevant key, such that we can detect if an element returned by a DELETEMIN has really been deleted by performing another DELETEMIN.

A balanced search tree can be used to implement a priority queue. Thus the existence of a dynamic cache-oblivious B-tree immediately implies the existence of a cache-oblivious priority queue where all operations can be performed in $O(\log_B N)$ memory transfers, where N is the total number of elements inserted. However, it turns out that one can design a priority queue where all operations can be performed in $\Theta(\mathrm{Sort}_{M,B}(N)/N) = O(\frac{1}{B} \log_{M/B} \frac{N}{B})$

memory transfers; for most realistic values of N, M, and B, this bound is less than 1 and we can, therefore, only obtain it in an amortized sense. In this section we describe two different structures that obtain these bounds [5, 16].

34.4.1 Merge Based Priority Queue: Funnel Heap

The cache-oblivious priority queue *Funnel Heap* due to Brodal and Fagerberg [16] is inspired by the sorting algorithm Funnelsort [15, 20]. The structure only uses binary merging; essentially it is a heap-ordered binary tree with mergers in the nodes and buffers on the edges.

Structure

The main part of the Funnel Heap structure is a sequence of k-mergers (Section 34.2.2) with double-exponentially increasing k, linked together in a list using binary mergers; refer to Figure 34.10. This part of the structure constitutes a single binary merge tree. Additionally, there is a single insertion buffer I.

More precisely, let k_i and s_i be values defined inductively by

$$\begin{aligned}(k_1, s_1) &= (2, 8)\ ,\\ s_{i+1} &= s_i(k_i+1)\ ,\\ k_{i+1} &= \lceil\lceil {s_{i+1}}^{1/3} \rceil\rceil\ ,\end{aligned} \tag{34.1}$$

where $\lceil\lceil x \rceil\rceil$ denotes the smallest power of two above x, i.e. $\lceil\lceil x \rceil\rceil = 2^{\lceil \log x \rceil}$. We note that ${s_i}^{1/3} \le k_i < 2{s_i}^{1/3}$, from which ${s_i}^{4/3} < s_{i+1} < 3{s_i}^{4/3}$ follows, so both s_i and k_i grow double-exponentially: $s_{i+1} = \Theta(s_i^{4/3})$ and $k_{i+1} = \Theta(k_i^{4/3})$. We also note that by induction on i we have $s_i = s_1 + \sum_{j=1}^{i-1} k_j s_j$ for all i.

A Funnel Heap consists of a linked list with *link i* containing a binary merger v_i, two buffers A_i and B_i, and a k_i-merger K_i having k_i input buffers $S_{i1}, \ldots, S_{ik_i}$. We refer to B_i,

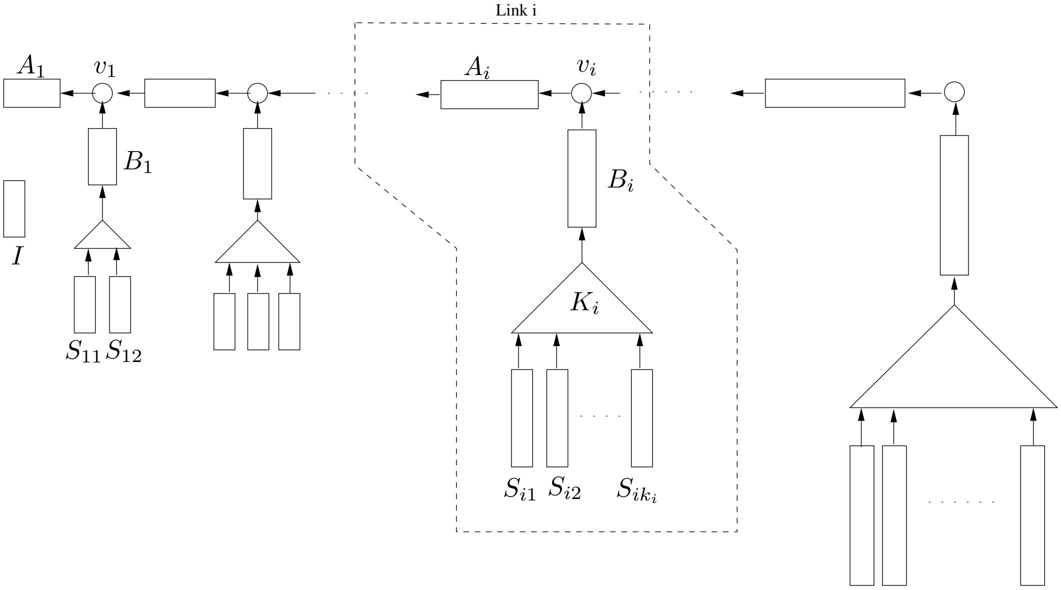


FIGURE 34.10: Funnel Heap: Sequence of k-mergers (triangles) linked together using buffers (rectangles) and binary mergers (circles).

K_i, and $S_{i1}, \ldots, S_{ik_i}$ as the *lower part* of the link. The size of both A_i and B_i is k_i^3, and the size of each S_{ij} is s_i. Link i has an associated counter c_i for which $1 \leq c_i \leq k_i + 1$. The initial value of c_i is one for all i. The structure also has one insertion buffer I of size s_1. We maintain the following invariants:

Invariant 1 *For link i, $S_{ic_i}, \ldots, S_{ik_i}$ are empty.*

Invariant 2 *On any path in the merge tree from some buffer to the root buffer A_1, elements appear in decreasing order.*

Invariant 3 *Elements in buffer I appear in sorted order.*

Invariant 2 can be rephrased as the entire merge tree being in heap order. It implies that in all buffers in the merge tree, the elements appear in sorted order, and that the minimum element in the queue will be in A_1 or I, if buffer A_1 is non-empty. Note that an invocation (Figure 34.7) of any binary merger in the tree maintains the invariants.

Layout

The Funnel Heap is laid out in consecutive memory locations in the order I, link 1, link 2, ..., with link i being laid out in the order c_i, A_i, v_i, B_i, K_i, S_{i1}, S_{i2}, ..., S_{ik_i}.

Operations

To perform a DeleteMin operation we compare the smallest element in I with the smallest element in A_1 and remove the smallest of these; if A_1 is empty we first perform an invocation of v_1. The correctness of this procedure follows immediately from Invariant 2.

To perform an Insert operation we insert the new element among the (constant number of) elements in I, maintaining Invariant 3. If the number of elements in I is now s_1, we examine the links in order to find the lowest index i for which $c_i \leq k_i$. Then we perform the following Sweep(i) operation.

In Sweep(i), we first traverse the path p from A_1 to S_{ic_i} and record how many elements are contained in each encountered buffer. Then we traverse the part of p going from A_i to S_{ic_i}, remove the elements in the encountered buffers, and form a sorted stream σ_1 of the removed elements. Next we form another sorted stream σ_2 of all elements in links $1, \ldots, i-1$ and in buffer I; we do so by marking A_i temporarily as exhausted and calling DeleteMin repeatedly. We then merge σ_1 and σ_2 into a single stream σ, and traverse p again while inserting the front (smallest) elements of σ in the buffers on p such that they contain the same numbers of elements as before we emptied them. Finally, we insert the remaining elements from σ into S_{ic_i}, reset c_l to one for $l = 1, 2, \ldots, i-1$, and increment c_i.

To see that Sweep(i) does not insert more than the allowed s_i elements into S_{ic_i}, first note that the lower part of link i is emptied each time c_i is reset to one. This implies that the lower part of link i never contains more than the number of elements inserted into $S_{i1}, S_{i2}, \ldots, S_{ik_i}$ by the at most k_i Sweep(i) operations occurring since last time c_i was reset. Since $s_i = s_1 + \sum_{j=1}^{i-1} k_j s_j$ for all i, it follows by induction on time that no instance of Sweep(i) inserts more than s_i elements into S_{ic_i}.

Clearly, Sweep(i) maintains Invariants 1 and 3, since I and the lower parts of links $1, \ldots, i-1$ are empty afterwards. Invariant 2 is also maintained, since the new elements in the buffers on p are the smallest elements in σ, distributed such that each buffer contains exactly the same number of elements as before the Sweep(i) operation. After the operation, an element on this path can only be smaller than the element occupying the same location before the operation, and therefore the merge tree is in heap order.

Analysis

To analyze the amortized cost of an Insert or DeleteMin operation, we first consider the number of memory transfers used to move elements upwards (towards A_1) by invocations of binary mergers in the merge tree. For now we assume that all invocations result in full buffers, i.e., that no exhaustions occur. We imagine charging the cost of filling a particular buffer evenly to the elements being brought into the buffer, and will show that this way an element from an input buffer of K_i is charged $O(\frac{1}{B}\log_{M/B} s_i)$ memory transfers during its ascent to A_1.

Our proof rely on the optimal replacement strategy keeping as many as possible of the first links of the Funnel Heap in fast memory at all times. To analyze the number of links that fit in fast memory, we define Δ_i to be the sum of the space used by links 1 to i and define i_M to be the largest i for which $\Delta_i \leq M$. By the space bound for k-mergers in Theorem 34.2 we see that the space used by link i is dominated by the $\Theta(s_i k_i) = \Theta({k_i}^4)$ space use of $S_{i1}, \ldots, S_{ik_i}$. Since $k_{i+1} = \Theta(k_i^{4/3})$, the space used by link i grows double-exponentially with i. Hence, Δ_i is a sum of double-exponentially increasing terms and is therefore dominated by its last term. In other words, $\Delta_i = \Theta({k_i}^4) = \Theta({s_i}^{4/3})$. By the definition of i_M we have $\Delta_{i_M} \leq M < \Delta_{i_M+1}$. Using $s_{i+1} = \Theta(s_i^{4/3})$ we see that $\log_M(s_{i_M}) = \Theta(1)$.

Now consider an element in an input buffer of K_i. If $i \leq i_M$ the element will not get charged at all in our charging scheme, since no memory transfers are used to fill buffers in the links that fit in fast memory. So assume $i > i_M$. In that case the element will get charged for the ascent through K_i to B_i and then through v_j to A_j for $j = i, i-1, \ldots, i_M$. First consider the cost of ascending through K_i: By Theorem 34.2, an invocation of the root of K_i to fill B_i with k_i^3 elements incurs $O(k_i + \frac{{k_i}^3}{B}\log_{M/B} {k_i}^3)$ memory transfers altogether. Since $M < \Delta_{i_M+1} = \Theta(k_{i_M+1}^4)$ we have $M = O({k_i}^4)$. By the tall cache assumption $M = \Omega(B^2)$ we get $B = O({k_i}^2)$, which implies $k_i = O({k_i}^3/B)$. Under the assumption that no exhaustions occur, i.e., that buffers are filled completely, it follows that an element is charged $O(\frac{1}{B}\log_{M/B} {k_i}^3) = O(\frac{1}{B}\log_{M/B} s_i)$ memory transfers to ascend through K_i and into B_i. Next consider the cost of ascending through v_j, that is, insertion into A_j, for $j = i, i-1, \ldots, i_M$: Filling of A_j incurs $O(1 + |A_j|/B)$ memory transfers. Since $B = O({k_{i_M+1}}^2) = O({k_{i_M}}^{8/3})$ and $|A_j| = {k_j}^3$, this is $O(|A_j|/B)$ memory transfers, so an element is charged $O(1/B)$ memory transfers for each A_j (under the assumption of no exhaustions). It only remains to bound the number of such buffers A_j, i.e., to bound $i - i_M$. From $s_i^{4/3} < s_{i+1}$ we have $s_{i_M}^{(4/3)^{i-i_M}} < s_i$. Using $\log_M(s_{i_M}) = \Theta(1)$ we get $i - i_M = O(\log\log_M s_i)$. From $\log\log_M s_i = O(\log_M s_i)$ and the tall cache assumption $M = \Omega(B^2)$ we get $i - i_M = O(\log_M s_i) = O(\log_{M/B} s_i)$. In total we have proved our claim that, assuming no exhaustions occur, an element in an input buffer of K_i is charged $O(\frac{1}{B}\log_{M/B} s_i)$ memory transfers during its ascent to A_1.

We imagine maintaining the *credit invariant* that each element in a buffer holds enough credits to be able to pay for the ascent from its current position to A_1, at the cost analyzed above. In particular, an element needs $O(\frac{1}{B}\log_{M/B} s_i)$ credits when it is inserted in an input buffer of K_i. The cost of these credits we will attribute to the Sweep(i) operation inserting it, effectively making all invocations of mergers be prepaid by Sweep(i) operations.

A Sweep(i) operation also incurs memory transfers by itself; we now bound these. In the Sweep(i) operation we first form σ_1 by traversing the path p from A_1 to S_{ic_i}. Since the links are laid out sequentially in memory, this traversal at most constitutes a linear scan of the consecutive memory locations containing A_1 through K_i. Such a scan takes $O((\Delta_{i-1} + |A_i| + |B_i| + |K_i|)/B) = O({k_i}^3/B) = O(s_i/B)$ memory transfers. Next we form

σ_2 using DELETEMIN operations; the cost of which is paid for by the credits placed on the elements. Finally, we merge of σ_1 and σ_2 into σ, and place some of the elements in buffers on p and some of the elements in S_{ic_i}. The number of memory transfers needed for this is bounded by the $O(s_i/B)$ memory transfers needed to traverse p and S_{ic_i}. Hence, the memory transfers incurred by the SWEEP(i) operation itself is $O(s_i/B)$.

After the SWEEP(i) operation, the credit invariant must be reestablished. Each of the $O(s_i)$ elements inserted into S_{ic_i} must receive $O(\frac{1}{B}\log_{M/B} s_i)$ credits. Additionally, the elements inserted into the part of the path p from A_1 through A_{i-1} must receive enough credits to cover their ascent to A_1, since the credits that resided with elements in the same positions before the operations were used when forming σ_2 by DELETEMIN operations. This constitutes $O(\Delta_{i-1}) = o(s_i)$ elements, which by the analysis above, must receive $O(\frac{1}{B}\log_{M/B} s_i)$ credits each. Altogether $O(s_i/B) + O(\frac{s_i}{B}\log_{M/B} s_i) = O(\frac{s_i}{B}\log_{M/B} s_i)$ memory transfers are attributed to a SWEEP(i) operation, again under the assumption that no exhaustions occur during invocations.

To actually account for exhaustions, that is, the memory transfers incurred when filling buffers that become exhausted, we note that filling a buffer partly incurs at most the same number of memory transfers as filling it entirely. This number was analyzed above to be $O(|A_i|/B)$ for A_i and $O(\frac{|B_i|}{B}\log_{M/B} s_i)$ for B_i, when $i > i_M$. If B_i become exhausted, only a SWEEP(i) can remove that status. If A_i become exhausted, only a SWEEP(j) for $j \geq i$ can remove that status. As at most a single SWEEP(j) with $j > i$ can take place between one SWEEP(i) and the next, B_i can only become exhausted once for each SWEEP(i), and A_i can only become exhausted twice for each SWEEP(i). From $|A_i| = |B_i| = {k_i}^3 = \Theta(s_i)$ it follows that charging SWEEP(i) an additional cost of $O(\frac{s_i}{B}\log_{M/B} s_i)$ memory transfers will cover all costs of filling buffers when exhaustion occurs.

Overall we have shown that we can account for all memory transfers if we attribute $O(\frac{s_i}{B}\log_{M/B} s_i)$ memory transfers to each SWEEP(i). By induction on i, we can show that at least s_i insertions have to take place between each SWEEP(i). Thus, if we charge the SWEEP(i) cost to the last s_i insertions preceding the SWEEP(i), each insertion is charged $O(\frac{1}{B}\log_{M/B} s_i)$ memory transfers. Given a sequence of operation on an initial empty priority queue, let $i_{\max}$ be the largest i for which SWEEP(i) takes place. We have $s_{i_{\max}} \leq N$, where N is the number of insertions in the sequence. An insertion can be charged by at most one SWEEP(i) for $i = 1, \ldots, i_{\max}$, so by the double-exponential growth of s_i, the number of memory transfers charged to an insertion is

$$O\left(\sum_{k=0}^{\infty} \frac{1}{B}\log_{M/B} N^{(3/4)^k}\right) = O\left(\frac{1}{B}\log_{M/B} N\right) = O\left(\frac{1}{B}\log_{M/B}\frac{N}{B}\right),$$

where the last equality follows from the tall cache assumption $M = \Omega(B^2)$.

Finally, we bound the space use of the entire structure. To ensure a space usage linear in N, we create a link i when it is first used, i.e., when the first SWEEP(i) occurs. At that point in time, c_i, A_i, v_i, B_i, K_i, and S_{i1} are created. These take up $\Theta(s_i)$ space combined. At each subsequent SWEEP(i) operation, we create the next input buffer S_{ic_i} of size s_i. As noted above, each SWEEP(i) is preceded by at least s_i insertions, from which an $O(N)$ space bound follows. To ensure that the entire structure is laid out in consecutive memory locations, the structure is moved to a larger memory area when it has grown by a constant factor. When allocated, the size of the new memory area is chosen such that it will hold the input buffers S_{ij} that will be created before the next move. The amortized cost of this is $O(1/B)$ per insertion.

THEOREM 34.6 *Using* $\Theta(M)$ *fast memory, a sequence of* N INSERT, DELETEMIN, *and* DELETE *operations can be performed on an initially empty Funnel Heap using* $O(N)$ *space in* $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ *amortized memory transfers each.*

Brodal and Fagerberg [16] gave a refined analysis for a variant of the Funnel Heap that shows that the structure adapts to different usage profiles. More precisely, they showed that the ith insertion uses amortized $O(\frac{1}{B}\log_{M/B}\frac{N_i}{B})$ memory transfers, where N_i can be defined in any of the following three ways: (a) N_i is the number of elements present in the priority queue when the ith insertion is performed, (b) if the ith inserted element is removed by a DELETEMIN operation prior to the jth insertion then $N_i = j - i$, or (c) N_i is the maximum rank of the ith inserted element during its lifetime in the priority queue, where rank denotes the number of smaller elements in the queue.

34.4.2 Exponential Level Based Priority Queue

While the Funnel Heap is inspired by Mergesort and uses k-mergers as the basic building block, the exponential level priority queue of Arge et al. [5] is somewhat inspired by distribution sorting and uses sorting as a basic building block.

Structure

The structure consists of $\Theta(\log\log N)$ *levels* whose sizes vary from N to some small size c below a constant threshold c_t; the size of a level corresponds (asymptotically) to the number of elements that can be stored within it. The i'th level from above has size $N^{(2/3)^{i-1}}$ and for convenience we refer to the levels by their size. Thus the levels from largest to smallest are level N, level $N^{2/3}$, level $N^{4/9}$, ..., level $X^{9/4}$, level $X^{3/2}$, level X, level $X^{2/3}$, level $X^{4/9}$, ..., level $c^{9/4}$, level $c^{3/2}$, and level c. In general, a level can contain any number of elements less than or equal to its size, except level N, which always contains $\Theta(N)$ elements. Intuitively, smaller levels store elements with smaller keys or elements that were more recently inserted. In particular, the minimum key element and the most recently inserted element are always in the smallest (lowest) level c. Both insertions and deletions are initially performed on the smallest level and may propagate up through the levels.

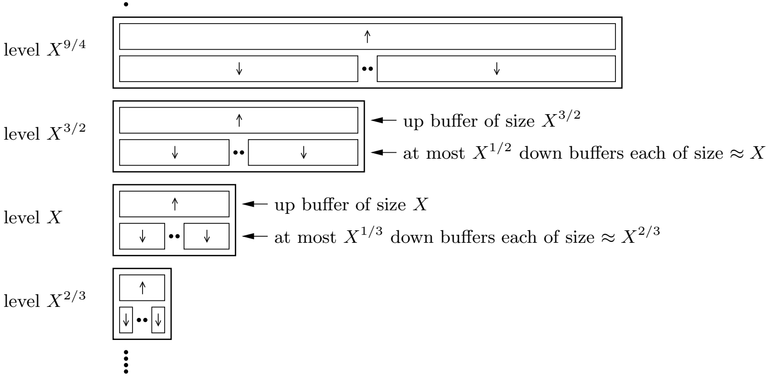

level $X^{9/4}$
level $X^{3/2}$
up buffer of size $X^{3/2}$
at most $X^{1/2}$ down buffers each of size ≈ X
level X
up buffer of size X
at most $X^{1/3}$ down buffers each of size ≈ $X^{2/3}$
level $X^{2/3}$

FIGURE 34.11: Levels $X^{2/3}$, X, $X^{3/2}$, and $X^{9/4}$ of the priority queue data structure.

Elements are stored in a level in a number of *buffers*, which are also used to transfer elements between levels. Level X consists of one *up buffer* u^X that can store up to X elements, and at most $X^{1/3}$ *down buffers* $d_1^X, \ldots, d_{X^{1/3}}^X$ each containing between $\frac{1}{2}X^{2/3}$ and $2X^{2/3}$ elements. Thus level X can store up to $3X$ elements. We refer to the maximum possible number of elements that can be stored in a buffer as the *size* of the buffer. Refer to Figure 34.11. Note that the size of a down buffer at one level matches the size (up to a constant factor) of the up buffer one level down.

We maintain three invariants about the relationships between the elements in buffers of various levels:

Invariant 4 *At level X, elements are sorted* **among** *the down buffers, that is, elements in d_i^X have smaller keys than elements in d_{i+1}^X, but elements within d_i^X are unordered.*

The element with largest key in each down buffer d_i^X is called a *pivot element*. Pivot elements mark the boundaries between the ranges of the keys of elements in down buffers.

Invariant 5 *At level X, the elements in the down buffers have smaller keys than the elements in the up buffer.*

Invariant 6 *The elements in the down buffers at level X have smaller keys than the elements in the down buffers at the next higher level $X^{3/2}$.*

The three invariants ensure that the keys of the elements in the down buffers get larger as we go from smaller to larger levels of the structure. Furthermore, an order exists between the buffers on one level: keys of elements in the up buffer are larger than keys of elements in down buffers. Therefore, down buffers are drawn below up buffers on Figure 34.11. However, the keys of the elements in an up buffer are unordered relative to the keys of the elements in down buffers one level up. Intuitively, up buffers store elements that are "on their way up", that is, they have yet to be resolved as belonging to a particular down buffer in the next (or higher) level. Analogously, down buffers store elements that are "on their way down"— these elements are by the down buffers partitioned into several clusters so that we can quickly find the cluster of smallest key elements of size roughly equal to the next level down. In particular, the element with overall smallest key is in the first down buffer at level c.

Layout

The priority queue is laid out in memory such that the levels are stored consecutively from smallest to largest with each level occupying a single region of memory. For level X we reserve space for exactly $3X$ elements: X for the up buffer and $2X^{2/3}$ for each possible down buffer. The up buffer is stored first, followed by the down buffers stored in an arbitrary order but linked together to form an ordered linked list. Thus $O(\sum_{i=0}^{\log_{3/2}\log_c N} N^{(2/3)^i}) = O(N)$ is an upper bound on the total memory used by the priority queue.

Operations

To implement the priority queue operations we use two general operations, *push* and *pull*. Push inserts X elements into level $X^{3/2}$, and pull removes the X elements with smallest keys from level $X^{3/2}$ and returns them in sorted order. An Insert or a DeleteMin is performed simply by performing a push or pull on the smallest level c.

Push. To push X elements into level $X^{3/2}$, we first sort the X elements cache-obliviously using $O(1 + \frac{X}{B}\log_{M/B}\frac{X}{B})$ memory transfers. Next we distribute the elements in the sorted list into the down buffers of level $X^{3/2}$ by scanning through the list and simultaneously visiting the down buffers in (linked) order. More precisely, we append elements to the end of the current down buffer $d_i^{X^{3/2}}$, and advance to the next down buffer $d_{i+1}^{X^{3/2}}$ as soon as we encounter an element with larger key than the pivot of $d_i^{X^{3/2}}$. Elements with larger keys than the pivot of the last down buffer are inserted in the up buffer $u^{X^{3/2}}$. Scanning through the X elements take $O(1 + \frac{X}{B})$ memory transfers. Even though we do not scan through every down buffer, we might perform at least one memory transfer for each of the $X^{1/2}$ possible buffers. Thus the total cost of distributing the X elements is $O(\frac{X}{B} + X^{1/2})$ memory transfers.

During the distribution of elements a down buffer may run full, that is, contain $2X$ elements. In this case, we split the buffer into two down buffers each containing X elements using $O(1 + \frac{X}{B})$ transfers. We place the new buffer in any free down buffer location for the level and update the linked list accordingly. If the level already has the maximum number $X^{1/2}$ of down buffers, we remove the last down buffer $d_{X^{1/2}}^X$ by inserting its no more than $2X$ elements into the up buffer using $O(1 + \frac{X}{B})$ memory transfers. Since X elements must have been inserted since the last time the buffer split, the amortized splitting cost per element is $O(\frac{1}{X} + \frac{1}{B})$ transfers. In total, the amortized number of memory transfers used on splitting buffers while distributing the X elements is $O(1 + \frac{X}{B})$.

If the up buffer runs full during the above process, that is, contains more than $X^{3/2}$ elements, we recursively *push* all of these elements into the next level up. Note that after such a recursive push, $X^{3/2}$ elements have to be inserted (pushed) into the up buffer of level $X^{3/2}$ before another recursive push is needed.

Overall we can perform a push of X elements from level X into level $X^{3/2}$ in $O(X^{1/2} + \frac{X}{B}\log_{M/B}\frac{X}{B})$ memory transfers amortized, not counting the cost of any recursive push operations; it is easy to see that a push maintains all three invariants.

Pull. To describe how to pull the X smallest keys elements from level $X^{3/2}$, we first assume that the down buffers contain at least $\frac{3}{2}X$ elements. In this case the first three down buffers $d_1^{X^{3/2}}$, $d_2^{X^{3/2}}$, and $d_3^{X^{3/2}}$ contain the between $\frac{3}{2}X$ and $6X$ smallest elements (Invariants 4 and 5). We find and remove the X smallest elements simply by sorting these elements using $O(1 + \frac{X}{B}\log_{M/B}\frac{X}{B})$ memory transfers. The remaining between $X/2$ and $5X$ elements are left in one, two, or three down buffers containing between $X/2$ and $2X$ elements each. These buffers can easily be constructed in $O(1 + \frac{X}{B})$ transfers. Thus we use $O(1 + \frac{X}{B}\log_{M/B}\frac{X}{B})$ memory transfers in total. It is easy to see that Invariants 4–6 are maintained.

In the case where the down buffers contain fewer than $\frac{3}{2}X$ elements, we first *pull* the $X^{3/2}$ elements with smallest keys from the next level up. Because these elements do not necessarily have smaller keys than the, say U, elements in the up buffer $u^{X^{3/2}}$, we then sort this up buffer and merge the two sorted lists. Then we insert the U elements with largest keys into the up buffer, and distribute the remaining between $X^{3/2}$ and $X^{3/2} + \frac{3}{2}X$ elements into $X^{1/2}$ down buffers containing between X and $X + \frac{3}{2}X^{1/2}$ each (such that the $O(\frac{1}{X} + \frac{1}{B})$ amortized down buffer split bound is maintained). It is easy to see that this maintains the three invariants. Afterwards, we can find the X minimal key elements as above. Note that after a recursive pull, $X^{3/2}$ elements have to be deleted (pulled) from the down buffers of level $X^{3/2}$ before another recursive pull is needed. Note also that a pull on level $X^{3/2}$ does not affect the number of elements in the up buffer $u^{X^{3/2}}$. Since we distribute elements into the down and up buffers after a recursive pull using one sort and one scan of $X^{3/2}$ elements, the cost of doing so is dominated by the cost of the recursive pull operation itself. Thus ignoring the cost of recursive pulls, we have shown that a pull of X elements from level $X^{3/2}$ down to level X can be performed in $O(1 + \frac{X}{B}\log_{M/B}\frac{X}{B})$

memory transfers amortized, while maintaining Invariants 4–6.

Analysis

To analyze the amortized cost of an Insert or DeleteMin operation, we consider the total number of memory transfers used to perform push and pull operations during $\frac{N}{2}$ operations; to ensure that the structure always consists of $O(\log\log N)$ levels and use $O(N)$ space we rebuild it using $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ memory transfers (or $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ transfers per operation) after every $\frac{N}{2}$ operations [5].

The total cost of $\frac{N}{2}$ such operations is analyzed as follows: We charge a push of X elements from level X up to level $X^{3/2}$ to level X. Since X elements have to be inserted in the up buffer u^X of level X between such pushes, and as elements can only be inserted in u^X when elements are inserted (pushed) into level X, $O(N/X)$ pushes are charged to level X during the $\frac{N}{2}$ operations. Similarly, we charge a pull of X elements from level $X^{3/2}$ down to level X to level X. Since between such pulls $\Theta(X)$ elements have to be deleted from the down buffers of level X by pulls on X, $O(N/X)$ pulls are charged to level X during the $\frac{N}{2}$ operations.

Above we argued that a push or pull charged to level X uses $O(X^{1/2}+\frac{X}{B}\log_{M/B}\frac{X}{B})$ memory transfers. We can reduce this cost to $O(\frac{X}{B}\log_{M/B}\frac{X}{B})$ by more carefully examining the costs for differently sized levels. First consider a push or pull of $X \geq B^2$ elements into or from level $X^{3/2} \geq B^3$. In this case $\frac{X}{B} \geq \sqrt{X}$, and we trivially have that $O(X^{1/2}+\frac{X}{B}\log_{M/B}\frac{X}{B}) = O(\frac{X}{B}\log_{M/B}\frac{X}{B})$. Next, consider the case $B^{4/3} \leq X < B^2$, where the $X^{1/2}$ term in the push bound can dominate and we have to analyze the cost of a push more carefully. In this case we are working on a level $X^{3/2}$ where $B^2 \leq X^{3/2} < B^3$; there is only one such level. Recall that the $X^{1/2}$ cost was from distributing X sorted elements into the less than $X^{1/2}$ down buffers of level $X^{3/2}$. More precisely, a block of each buffer may have to be loaded and written back without transferring a full block of elements into the buffer. Assuming $M = \Omega(B^2)$, we from $X^{1/2} \leq B$ see that a block for each of the buffers can fit into fast memory. Consequently, if a fraction of the fast memory is used to keep a partially filled block of each buffer of level $X^{3/2}$ ($B^2 \leq X^{3/2} \leq B^3$) in fast memory at all times, and full blocks are written to disk, the $X^{1/2}$ cost would be eliminated. In addition, if all of the levels of size less than B^2 (of total size $O(B^2)$) are also kept in fast memory, all transfer costs associated with them would be eliminated. The optimal paging strategy is able to keep the relevant blocks in fast memory at all times and thus eliminates these costs.

Finally, since each of the $O(N/X)$ push and pull operations charged to level X ($X > B^2$) uses $O(\frac{X}{B}\log_{M/B}\frac{X}{B})$ amortized memory transfers, the total amortized transfer cost of an Insert or DeleteMin operation in the sequence of $\frac{N}{2}$ such operations is

$$O\left(\sum_{i=0}^{\infty}\frac{1}{B}\log_{M/B}\frac{N^{(2/3)^i}}{B}\right) = O\left(\frac{1}{B}\log_{M/B}\frac{N}{B}\right) .$$

THEOREM 34.7 *Using $\Theta(M)$ fast memory, N Insert, DeleteMin, and Delete operations can be performed on an initially empty exponential level priority queue using $O(N)$ space in $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ amortized memory transfers each.*

34.5 2d Orthogonal Range Searching

As discussed in Section 34.3, there exist cache-oblivious B-trees that support updates and queries in $O(\log_B N)$ memory transfers (e.g. Theorem 34.5); several cache-oblivious B-tree variants can also support (one-dimensional) range queries in $O(\log_B N + \frac{K}{B})$ memory transfers [11, 12, 18], but at an increased amortized update cost of $O(\log_B N + \frac{\log^2 N}{B}) = O(\log_B^2 N)$ memory transfers (e.g. Theorem 34.4).

In this section we discuss cache-oblivious data structures for two-dimensional orthogonal range searching, that is, structures for storing a set of N points in the plane such that the points in a axis-parallel query rectangle can be reported efficiently. In Section 34.5.1 we first discuss a cache-oblivious version of a *kd-tree*. This structure uses linear space and answers queries in $O(\sqrt{N/B} + \frac{K}{B})$ memory transfers; this is optimal among linear space structures [22]. It supports updates in $O(\frac{\log N}{B} \cdot \log_{M/B} N) = O(\log_B^2 N)$ transfers. In Section 34.5.2 we then discuss a cache-oblivious version of a two-dimensional *range tree*. The structure answers queries in the optimal $O(\log_B N + \frac{K}{B})$ memory transfers but uses $O(N \log^2 N)$ space. Both structures were first described by Agarwal et al. [1].

34.5.1 Cache-Oblivious kd-Tree

Structure

The cache-oblivious kd-tree is simply a normal kd-tree laid out in memory using the van Emde Boas layout. This structure, proposed by Bentley [13], is a binary tree of height $O(\log N)$ with the N points stored in the leaves of the tree. The internal nodes represent a recursive decomposition of the plane by means of axis-orthogonal lines that partition the set of points into two subsets of equal size. On even levels of the tree the dividing lines are horizontal, and on odd levels they are vertical. In this way a rectangular region R_v is naturally associated with each node v, and the nodes on any particular level of the tree partition the plane into disjoint regions. In particular, the regions associated with the leaves represent a partition of the plane into rectangular regions containing one point each. Refer to Figure 34.12.

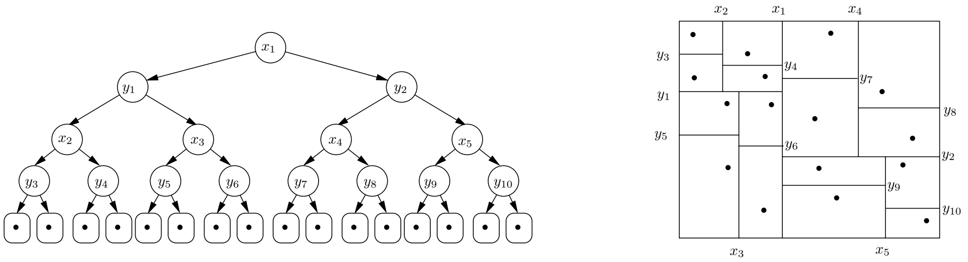


FIGURE 34.12: kd-tree and the corresponding partitioning.

Query

An orthogonal range query Q on a kd-tree $\mathcal{T}$ is answered recursively starting at the root: At a node v we advance the query to a child v_c of v if Q intersects the region R_{v_c} associated with v_c. At a leaf w we return the point in w if it is contained in Q. A standard argument shows that the number of nodes in $\mathcal{T}$ visited when answering Q, or equivalently, the number of nodes v where R_v intersects Q, is $O(\sqrt{N}+K)$; $\sqrt{N}$ nodes v are visited where R_v is intersected by the boundary of Q and K nodes u with R_u completely contained in Q [13].

If the kd-tree $\mathcal{T}$ is laid out using the van Emde Boas layout, we can bound the number of memory transfers used to answer a query by considering the nodes $\log B$ levels above the leaves of $\mathcal{T}$. There are $O(\frac{N}{B})$ such nodes as the subtree $\mathcal{T}_v$ rooted in one such node v contains B leaves. By the standard query argument, the number of these nodes visited by a query is $O(\sqrt{N/B}+\frac{K}{B})$. Thus, the number of memory transfers used to visit nodes more than $\log B$ levels above the leaves is $O(\sqrt{N/B}+\frac{K}{B})$. This is also the overall number of memory transfers used to answer a query, since (as argued in Section 34.2.1) the nodes in $\mathcal{T}_v$ are contained in $O(1)$ blocks, i.e. any traversal of (any subset of) the nodes in a subtree $\mathcal{T}_v$ can be performed in $O(1)$ memory transfers.

Construction

In the RAM model, a kd-tree on N points can be constructed recursively in $O(N\log N)$ time; the root dividing line is found using an $O(N)$ time median algorithm, the points are distributed into two sets according to this line in $O(N)$ time, and the two subtrees are constructed recursively. Since median finding and distribution can be performed cache-obliviously in $O(N/B)$ memory transfers [20, 24], a cache-oblivious kd-tree can be constructed in $O(\frac{N}{B}\log N)$ memory transfers. Agarwal et al. [1] showed how to construct $\log\sqrt{N}=\frac{1}{2}\log N$ levels in $O(\mathrm{Sort}_{M,B}(N))$ memory transfers, leading to a recursive construction algorithms using only $O(\mathrm{Sort}_{M,B}(N))$ memory transfers.

Updates

In the RAM model a kd-tree $\mathcal{T}$ can relatively easily be modified to support deletions in $O(\log N)$ time using global rebuilding. To delete a point from $\mathcal{T}$, we simply find the relevant leaf w in $O(\log N)$ time and remove it. We then remove w's parent and connect w's grandparent to w's sibling. The resulting tree is no longer a kd-tree but it still answers queries in $O(\sqrt{N}+T)$ time, since the standard argument still applies. To ensure that N is proportional to the actual number of points in $\mathcal{T}$, the structure is completely rebuilt after $\frac{N}{2}$ deletions. Insertions can be supported in $O(\log^2 N)$ time using the so-called logarithmic method [14], that is, by maintaining $\log N$ kd-trees where the i'th kd-tree is either empty or of size 2^i and then rebuilding a carefully chosen set of these structures when performing an insertion.

Deletes in a cache-oblivious kd-tree is basically done as in the RAM version. However, to still be able to load a subtree $\mathcal{T}_v$ with B leaves in $O(1)$ memory transfers and obtain the $O(\sqrt{N/B}+\frac{K}{B})$ query bound, data locality needs to be carefully maintained. By laying out the kd-tree using (a slightly relaxed version of) the exponential layout (Section 34.3.2) rather than the van Emde Boas layout, and by periodically rebuilding parts of this layout, Agarwal et al. [1] showed how to perform a delete in $O(\log_B N)$ memory transfers amortized while maintaining locality. They also showed how a slightly modified version of the logarithmic method and the $O(\mathrm{Sort}_{M,B}(N))$ construction algorithms can be used to perform inserts in $O(\frac{\log N}{B}\log_{M/B} N)=O(\log_B^2 N)$ memory transfers amortized.

THEOREM 34.8 *There exists a cache-oblivious (kd-tree) data structure for storing a set of N points in the plane using linear space, such that an orthogonal range query can be answered in $O(\sqrt{N/B}+\frac{K}{B})$ memory transfers. The structure can be constructed cache-*

obliviously in $O(\mathrm{Sort}_{M,B}(N))$ *memory transfers and supports updates in* $O(\frac{\log N}{B}\log_{M/B} N) = O(\log_B^2 N)$ *memory transfers.*

34.5.2 Cache-Oblivious Range Tree

The main part of the cache-oblivious range tree structure for answering (four-sided) orthogonal range queries is a structure for answering three-sided queries $Q = [x_l, x_r] \times [y_b, \infty)$, that is, for finding all points with x-coordinates in the interval $[x_l, x_r]$ and y-coordinates above y_b. Below we discuss the two structures separately.

Three-Sided Queries.

Structure

Consider dividing the plane into $\sqrt{N}$ vertical *slabs* $X_1, X_2, \ldots, X_{\sqrt{N}}$ containing $\sqrt{N}$ points each. Using these slabs we define $2\sqrt{N}-1$ *buckets.* A bucket is a rectangular region of the plane that completely spans one or more consecutive slabs and is unbounded in the positive y-direction, like a three-sided query. To define the $2\sqrt{N}-1$ buckets we start with $\sqrt{N}$ *active* buckets $b_1, b_2, \ldots, b_{\sqrt{N}}$ corresponding to the $\sqrt{N}$ slabs. The x-range of the slabs define a natural linear ordering on these buckets. We then imagine sweeping a horizontal sweep line from $y = -\infty$ to $y = \infty$. Every time the total number of points above the sweep line in two adjacent active buckets, b_i and b_j, in the linear order falls to $\sqrt{N}$, we mark b_i and b_j as *inactive.* Then we construct a new active bucket spanning the slabs spanned by b_i and b_j with a bottom y-boundary equal to the current position of the sweep line. This bucket replaces b_i and b_j in the linear ordering of active buckets. The total number of buckets defined in this way is $2\sqrt{N}-1$, since we start with $\sqrt{N}$ buckets and the number of active buckets decreases by one every time a new bucket is constructed. Note that the procedure defines an *active* y-interval for each bucket in a natural way. Buckets overlap but the set of buckets with active y-intervals containing a given y-value (the buckets active when the sweep line was at that value) are non-overlapping and span all the slabs. This means that the active y-intervals of buckets spanning a given slab are non-overlapping. Refer to Figure 34.13(a).

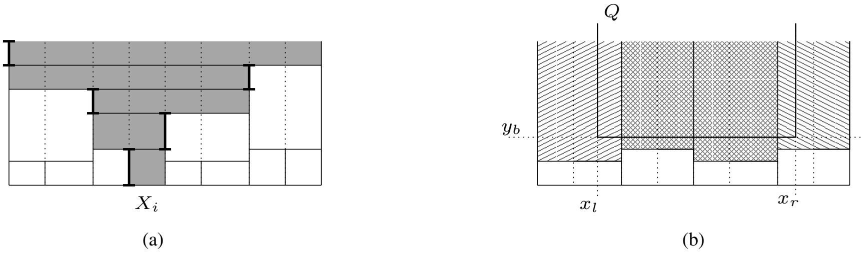


FIGURE 34.13: (a) Active intervals of buckets spanning slab X_i; (b) Buckets active at y_b.

After defining the $2\sqrt{N}-1$ buckets, we are ready to present the three-sided query data structure; it is defined recursively: It consists of a cache-oblivious B-tree $\mathcal{T}$ on the $\sqrt{N}$ boundaries defining the $\sqrt{N}$ slabs, as well as a cache-oblivious B-tree for each of the $\sqrt{N}$ slabs; the tree $\mathcal{T}_i$ for slab i contains the bottom endpoint of the active y-intervals of the $O(\sqrt{N})$ buckets spanning the slab. For each bucket b_i we also store the $\sqrt{N}$ points in b_i in a list $\mathcal{B}_i$ sorted by y-coordinate. Finally, recursive structures $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_{2\sqrt{N}-1}$ are built on the $\sqrt{N}$ points in each of the $2\sqrt{N}-1$ buckets.

Layout

The layout of the structure in memory consists of $O(N)$ memory locations containing $\mathcal{T}$, then $\mathcal{T}_1, \ldots, \mathcal{T}_{\sqrt{N}}$, and $\mathcal{B}_1, \ldots, \mathcal{B}_{2\sqrt{N}-1}$, followed by the recursive structures $\mathcal{S}_1, \ldots, \mathcal{S}_{2\sqrt{N}-1}$. Thus the total space use of the structure is $S(N) \leq 2\sqrt{N} \cdot S(\sqrt{N}) + O(N) = O(N \log N)$.

Query

To answer a three-sided query Q, we consider the buckets whose active y-interval contain y_b. These buckets are non-overlapping and together they contain all points in Q, since they span all slabs and have bottom y-boundary below y_b. We report all points that satisfy Q in each of the buckets with x-range completely between x_l and x_r. At most two other buckets b_l and b_r—the ones containing x_l and x_r—can contain points in Q, and we find these points recursively by advancing the query to $\mathcal{S}_l$ and $\mathcal{S}_r$. Refer to Figure 34.13(b).

We find the buckets b_l and b_r that need to be queried recursively and report the points in the completely spanned buckets as follows. We first query $\mathcal{T}$ using $O(\log_B \sqrt{N})$ memory transfers to find the slab X_l containing x_l. Then we query $\mathcal{T}_l$ using another $O(\log_B \sqrt{N})$ memory transfers to find the bucket b_l with active y-interval containing y_b. We can similarly find b_r in $O(\log_B \sqrt{N})$ memory transfers. If b_l spans slabs $X_l, X_{l+1}, \ldots, X_m$ we then query $\mathcal{T}_{m+1}$ with y_b in $O(\log_B \sqrt{N})$ memory transfers to find the active bucket b_i to the right of b_l completely spanned by Q (if it exists). We report the relevant points in b_i by scanning $\mathcal{B}_i$ top-down until we encounter a point not contained in Q. If K' is the number or reported points, a scan of $\mathcal{B}_i$ takes $O(1 + \frac{K'}{B})$ memory transfers. We continue this procedure for each of the completely spanned active buckets. By construction, we know that every two adjacent such buckets contain at least $\sqrt{N}$ points above y_b. First consider the part of the query that takes place on recursive levels of size $N \geq B^2$, such that $\sqrt{N}/B \geq \log_B \sqrt{N} \geq 1$. In this case the $O(\log_B \sqrt{N})$ overhead in finding and processing two consecutive completely spanned buckets is smaller than the $O(\sqrt{N}/B)$ memory transfers used to report output points; thus we spend $O(\log_B \sqrt{N} + \frac{K_i}{B})$ memory transfers altogether to answer a query, not counting the recursive queries. Since we perform at most two queries on each level of the recursion (in the active buckets containing x_l and x_r), the total cost over all levels of size at least B^2 is $O(\sum_{i=1}^{\log \log_B N} \log_B N^{1/2^i} + \frac{K_i}{B}) = O(\log_B N + \frac{K}{B})$ transfers. Next consider the case where $N = B$. In this case the whole level, that is, $\mathcal{T}$, $\mathcal{T}_1, \ldots, \mathcal{T}_{\sqrt{B}}$ and $\mathcal{B}_1, \ldots, \mathcal{B}_{2\sqrt{B}-1}$, is stored in $O(B)$ contiguously memory memory locations and can thus be loaded in $O(1)$ memory transfers. Thus the optimal paging strategy can ensure that we only spend $O(1)$ transfers on answering a query. In the case where $N \leq \sqrt{B}$, the level and *all* levels of recursion below it occupies $O(\sqrt{B} \log \sqrt{B}) = O(B)$ space. Thus the optimal paging strategy can load it and all relevant lower levels in $O(1)$ memory transfers. This means that overall we answer a query in $O(\log_B N + \frac{K}{B})$ memory transfers, *provided* that N and B are such that we have a level of size B^2 (and thus of size B and $\sqrt{B}$); when answering a query on a level of size between B and B^2 we cannot charge the $O(\log_B \sqrt{N})$ cost of visiting two active consecutive buckets to the ($< B$) points found in the two buckets. Agarwal et al. [1] showed how to guarantee that we have a level of size B^2 by assuming that $B = 2^{2^d}$ for some non-negative integer d. Using a somewhat different construction, Arge et al. [6] showed how to remove this assumption.

THEOREM 34.9 *There exists a cache-oblivious data structure for storing N points in the plane using $O(N \log N)$ space, such that a three-sided orthogonal range query can be answered in $O(\log_B N + \frac{K}{B})$ memory transfers.*

Four-sided queries.

Using the structure for three-sided queries, we can construct a cache-oblivious range tree structure for four-sided orthogonal range queries in a standard way. The structure consists of a cache-oblivious B-tree $\mathcal{T}$ on the N points sorted by x-coordinates. With each internal node v we associate a secondary structure for answering three-sided queries on the points stored in the leaves of the subtree rooted at v: If v is the left child of its parent then we have a three-sided structure for answering queries with the opening to the right, and if v is the right child then we have a three-sided structure for answering queries with the opening to the left. The secondary structures on each level of the tree use $O(N \log N)$ space, for a total space usage of $O(N \log^2 N)$.

To answer an orthogonal range query Q, we search down $\mathcal{T}$ using $O(\log_B N)$ memory transfers to find the first node v where the left and right x-coordinate of Q are contained in different children of v. Then we query the right opening secondary structure of the left child of v, and the left opening secondary structure of the right child of v, using $O(\log_B N + \frac{K}{B})$ memory transfers. Refer to Figure 34.14. It is easy to see that this correctly reports all K points in Q.

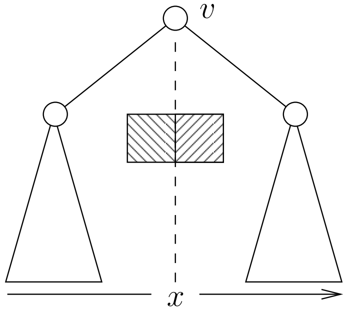


FIGURE 34.14: Answering a four-sided query in v using two three-sided queries in v's children.

THEOREM 34.10 *There exists a cache-oblivious data structure for storing N points in the plane using $O(N \log^2 N)$ space, such that an orthogonal range query can be answered in $O(\log_B N + \frac{K}{B})$ memory transfers.*

Acknowledgments

Lars Arge was supported in part by the National Science Foundation through ITR grant EIA–0112849, RI grant EIA–9972879, CAREER grant CCR–9984099, and U.S.–Germany Cooperative Research Program grant INT–0129182.

Gerth Stølting Brodal was supported by the Carlsberg Foundation (contract number ANS-0257/20), BRICS (Basic Research in Computer Science, www.brics.dk, funded by the Danish National Research Foundation), and the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

Rolf Fagerberg was supported by BRICS (Basic Research in Computer Science, www.brics.dk, funded by the Danish National Research Foundation), and the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT). Part of this work was done while at University of Aarhus.

References

[1] P. K. Agarwal, L. Arge, A. Danner, and B. Holland-Minkley. Cache-oblivious data structures for orthogonal range searching. In *Proc. 19th ACM Symposium on Computational Geometry*, pages 237–245. ACM Press, 2003.

[2] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, Sept. 1988.

[3] A. Andersson and T. W. Lai. Fast updating of well-balanced trees. In *Proc. 2nd Scandinavian Workshop on Algorithm Theory*, volume 447 of *Lecture Notes in Computer Science*, pages 111–121. Springer, 1990.

[4] L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–358. Kluwer Academic Publishers, 2002.

[5] L. Arge, M. Bender, E. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority-queue and graph algorithms. In *Proc. 34th ACM Symposium on Theory of Computation*, pages 268–276. ACM Press, 2002.

[6] L. Arge, G. S. Brodal, and R. Fagerberg. Improved cache-oblivious two-dimensional orthogonal range searching. Unpublished results, 2004.

[7] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[8] M. Bender, E. Demaine, and M. Farach-Colton. Efficient tree layout in a multilevel memory hierarchy. In *Proc. 10th Annual European Symposium on Algorithms*, volume 2461 of *Lecture Notes in Computer Science*, pages 165–173. Springer, 2002. Full version at http://www.cs.sunysb.edu/~bender/pub/treelayout-full.ps.

[9] M. A. Bender, G. S. Brodal, R. Fagerberg, D. Ge, S. He, H. Hu, J. Iacono, and A. López-Ortiz. The cost of cache-oblivious searching. In *Proc. 44th Annual IEEE Symposium on Foundations of Computer Science*, pages 271–282. IEEE Computer Society Press, 2003.

[10] M. A. Bender, R. Cole, and R. Raman. Exponential structures for cache-oblivious algorithms. In *Proc. 29th International Colloquium on Automata, Languages, and Programming*, vol. 2380, Lecture Notes in Computer Science, pages 195–207. Springer, 2002.

[11] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. 41st Annual IEEE Symposium on Foundations of Computer Science*, pages 339–409. IEEE Computer Society Press, 2000.

[12] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious

dynamic dictionary. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 29–38. SIAM, 2002.

[13] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communication of the ACM*, 18:509–517, 1975.

[14] J. L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, 1979.

[15] G. S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. In *Proc. 29th International Colloquium on Automata, Languages, and Programming*, vol. 2380, Lecture Notes in Computer Science, pages 426–438. Springer, 2002.

[16] G. S. Brodal and R. Fagerberg. Funnel heap - a cache oblivious priority queue. In *Proc. 13th International Symposium on Algorithms and Computation*, volume 2518 of *Lecture Notes in Computer Science*, pages 219–228. Springer, 2002.

[17] G. S. Brodal and R. Fagerberg. On the limits of cache-obliviousness. In *Proc. 35th ACM Symposium on Theory of Computation*, pages 307–315. ACM Press, 2003.

[18] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 39–48. SIAM, 2002.

[19] G. S. Brodal, R. Fagerberg, and K. Vinther. Engineering a cache-oblivious sorting algorithm. In *Proc. 6th Workshop on Algorithm Engineering and Experiments*, 2004.

[20] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Annual IEEE Symposium on Foundations of Computer Science*, pages 285–298. IEEE Computer Society Press, 1999.

[21] A. Itai, A. G. Konheim, and M. Rodeh. A sparse table implementation of priority queues. In *Proc. 8th International Colloquium on Automata, Languages, and Programming*, volume 115 of *Lecture Notes in Computer Science*, pages 417–431. Springer, 1981.

[22] K. V. R. Kanth and A. K. Singh. Optimal dynamic range searching in non-replicating index structures. In *Proc. International Conference on Database Theory*, volume 1540 of *Lecture Notes in Computer Science*, pages 257–276. Springer, 1999.

[23] R. E. Ladner, R. Fortna, and B.-H. Nguyen. A comparison of cache aware and cache oblivious static search trees using program instrumentation. In *Experimental Algorithmics, From Algorithm Design to Robust and Efficient Software (Dagstuhl seminar, September 2000)*, volume 2547 of *Lecture Notes in Computer Science*, pages 78–92. Springer, 2002.

[24] H. Prokop. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1999.

[25] N. Rahman, R. Cole, and R. Raman. Optimized predecessor data structures for internal memory. In *Proc. 3rd Workshop on Algorithm Engineering*, volume 2141 of *Lecture Notes in Computer Science*, pages 67–78. Springer, 2001.

[26] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6:80–82, 1977.

[27] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, June 2001.

35

Dynamic Trees

Camil Demetrescu
Università di Roma

Irene Finocchi
Università di Roma

Giuseppe F. Italiano
Università di Roma

35.1 Introduction

In this chapter we consider the problem of maintaining properties of a collection of vertex-disjoint trees that change over time as edges are added or deleted. The trees can be rooted or free, and vertices and edges may be associated with real-valued costs that may change as well. A straightforward solution would be to store explicitly with each vertex its parent and cost, if any: with this representation each update would cost only $O(1)$ time, but answering queries would be typically proportional to the size or to the depth of the tree, which may be linear in the worst case. By representing the structure of the trees implicitly, one can reduce the query time while slightly increasing the update time. The typical achieved bounds are logarithmic in the number of vertices of the forest, either in the worst-case or amortized over a sequence of operations.

While the basic tree update operations are edge insertions, edge deletions, and possibly vertex/edge cost changes, many properties of dynamically changing trees have been considered in the literature. The basic query operation is tree membership: while the forest of trees is dynamically changing, we would like to know at any time which tree contains a given vertex, or whether two vertices are in the same tree. Dynamic tree membership is a special case of dynamic connectivity in undirected graphs, and indeed in Chapter 36 we will see that some of the data structures developed here for trees are used to solve the more general problem on graphs. We remark that, if only edge insertions are allowed, the tree membership problem is equivalent to maintaining disjoint sets under union operations and thus the well known set union data structures can solve it [17]. In this chapter we will instead consider the problem in a fully dynamic setting, in which also edge deletions are allowed, and present efficient data structures such as the *linking and cutting trees* of Sleator and Tarjan [15] and the *topology trees* of Frederickson [6].

Other properties that have been considered are finding the least common ancestor of two vertices, the center, the median, or the diameter of a tree [1, 2, 15]. When costs are associated either to vertices or to edges, one could also ask what is the minimum or maximum cost in a given path. A variant of topology trees, known as *top trees* [1], are especially well suited at maintaining this kind of path information.

ET trees, first introduced in [18] and much used in [10], allow it to deal easily with forests whose vertices are associated with weighted or unweighted keys, supporting, e.g., minkey queries, which require to return a key of minimum weight in the tree that contains a given vertex. *Reachability trees*, introduced by Even and Shiloach in [5], support instead distance and shortest path queries and have been widely used to solve dynamic path problems on directed graphs (see, e.g., [10, 12]).

35.2 Linking and Cutting Trees

In this section we present a data structure due to Sleator and Tarjan [15] useful to maintain a collection of rooted trees, each of whose vertices has a real-valued cost, under an arbitrary sequence of the following operations:

- `maketree`(v): initialize a new tree consisting of a single vertex v with cost zero.
- `findroot`(v): return the root of the tree containing vertex v.
- `findcost`(v): return a vertex of minimum cost in the path from v to `findroot`(v).
- `addcost`(v, δ): add the real number δ to the cost of every vertex in the path from v to `findroot`(v).
- `link`(v, w): merge the trees containing vertices v and w by inserting edge (v, w). This operation assumes that v and w are in different trees and that v is a tree root.
- `cut`(v): delete the edge from v to its parent, thus splitting the tree containing vertex v into two trees. This operation assumes that v is not a tree root.

The data structure of Sleator and Tarjan is known as *linking and cutting trees* and supports all the above operations in $O(\log n)$ time, by representing the structure of the trees implicitly. Other operations that can be supported within the same time bound are changing the root of a tree, finding the parent of a vertex, and finding the least common ancestor of two vertices. In particular, the possibility of making a given vertex v root of a tree makes the data structure powerful enough to handle problems requiring linking and cutting of free (i.e., unrooted) trees. Furthermore, the same time bounds can be obtained when real costs are associated with edges rather than with vertices.

The rest of this section is organized as follows. In Section 35.2.1 we show how to implement the operations given above using simpler primitives defined on paths (rather than on trees), and in Section 35.2.2 we describe the implementation of these primitives on paths. For simplicity, we only describe a solution that achieves $O(\log n)$ amortized (rather than worst-case) time per operation. Details of all these results may be found in [15].

35.2.1 Using Operations on Vertex-Disjoint Paths

In this section we show the reduction between the operations on trees and a suitable collection of operations on vertex-disjoint paths. Assume we know how to perform the following operations:

- `makepath`(v): initialize a new path consisting of a single vertex v with cost zero;

- `findpath`(v): return the path containing vertex v;
- `findpathtail`(p): return the tail (last vertex) of path p;
- `findpathcost`(p): return a vertex of minimum cost in path p;
- `addpathcost`(p, δ): add the real value δ to the cost of every vertex in path p;
- `join`(p, v, q): merge path p, vertex v, and path q into a single path by inserting one edge from the tail of p to v, and one edge from v to the head (first vertex) of q, and return the new path. Either p or q can be empty;
- `split`(v): divide the path containing vertex v into at most three paths by deleting the edges incident to v. Return the two new paths p (containing all the vertices before v) and q (containing all the vertices after v). Again, either p or q can be empty.

In order to solve the problem of linking and cutting trees, we partition each tree into a set of vertex disjoint paths. Each tree operation will be defined in terms of one or more path operations. This partition is defined by allowing each tree edge to be either *solid* or *dashed* and by maintaining the invariant that at most one solid edge enters each vertex (we consider an edge oriented from a child to its parent). Removing dashed edges therefore partitions the tree into vertex-disjoint *solid paths*. Dashed edges are represented implicitly: we associate with each path p its *successor*, that is the vertex entered by the dashed edge leaving the tail of p. If the tail of p is a root, *successor*(p) is *null*. Each path will be represented by a vertex on it (an empty path being represented by *null*). In order to convert dashed edges to solid (and vice-versa) we will be using the following operation:

- `expose`(v): make the tree path from v to `findroot`(v) solid. This is done by converting dashed edges in the path to solid, and solid edges incident to the path to dashed. Return the resulting solid path.

Now we describe how to implement tree operations in terms of path operations:

- a `maketree`(v) is done by a `makepath`(v) followed by setting *successor*(v) to *null*;
- a `findroot`(v) is a `findpathtail(expose(`v`))`;
- a `findcost`(v) is a `findpathcost(expose(`v`))`;
- an `addcost`(v, δ) is an `addpathcost(expose(`v`),`δ`)`;
- a `link`(v, w) is implemented by performing first an `expose`(v) that makes v into a one-vertex solid path, then an `expose`(w) that makes the path from w to its root solid, and then by joining these two solid paths: in short, this means assigning *null* to `successor(join(`*null*`,expose(`v`),expose(`w`)))`;
- to perform a `cut`(v), we first perform an `expose`(v), which leaves v with no entering solid edge. We then perform a `split`(v), which returns paths p and q: since v is the head of its solid path and is not a tree root, p will be empty, while q will be non-empty. We now complete the operation by setting both *successor*(v) and *successor*(q) to *null*.

To conclude, we need to show how to perform an `expose`, i.e., how to convert all the dashed edges in the path from a given vertex to the tree root to solid maintaining the invariant that at most one solid edge enters each vertex. Let x be a vertex of this path such that the edge from x to its parent w is dashed (and thus $w = successor(x)$). What we would like to do is to convert edge (x, w) into solid, and to convert the solid edge previously entering w (if any) into dashed. We call this operation a *splice*. The pseudocode in Figure 35.1 implements `expose`(v) as a sequence of splices. Path p, initialized to be

```
function expose ($v$)
begin
    $p \leftarrow null$
    while $v \neq null$ do
        $w \leftarrow$ successor(findpath($v$))
        $[q, r] \leftarrow$ split($v$)
        if $q \neq null$ then successor($q$)$\leftarrow v$
        $p \leftarrow$ join($p, v, r$)
        $v \leftarrow w$
    successor($p$)$\leftarrow null$
end
```

FIGURE 35.1: Implementation of `expose`(v).

empty, at the end of the execution will contain the solid path from v to `findroot`(v). Each iteration of the **while** loop performs a splice at v by converting to solid the edge from the tail of p to v (if $p \neq null$) and to dashed the edge from the tail of q to v (if $q \neq null$). A step-by-step execution of `expose` on a running example is shown in Figure 35.2.

From the description above, each tree operation takes $O(1)$ path operations and at most one `expose`. Each splice within an `expose` requires $O(1)$ path operations. Hence, in order to compute the running time, we need first to count the number of splices per `expose` and then to show how to implement the path operations. With respect to the former point, Sleator and Tarjan prove that a sequence of m tree operations causes $O(m \log n)$ splices, and thus $O(\log n)$ splices amortized per `expose`.

THEOREM 35.1 *[15] Any sequence of m tree operations (including n `maketree`) requires $O(m)$ path operations and at most m `expose`. The exposes can be implemented with $O(m \log n)$ splices, each of which requires $O(1)$ path operations.*

35.2.2 Implementing Operations on Vertex-Disjoint Paths

We now describe how to represent solid paths in order to implement efficiently tree operations. Each solid path is represented by a binary tree whose nodes in symmetric order are the vertices in the path; each node x contains pointers to its parent $p(x)$, to its left child $l(x)$, and to its right child $r(x)$. We call the tree representing a solid path a *solid tree*. The vertex representing a solid path is the root of the corresponding solid tree, and thus the root of the solid tree contains a pointer to the successor of the path in the dynamic tree.

Vertex costs are represented as follows. Let $cost(x)$ be the cost of vertex x, and let $mincost(x)$ be the minimum cost among the descendants of x in its solid tree. Rather than storing these two values, in order to implement `addcost` operations efficiently, we store at x the incremental quantities $\Delta cost(x)$ and $\Delta min(x)$ defined as follows:

$$\Delta cost(x) = cost(x) - mincost(x)$$

$$\Delta min(x) = \begin{cases} mincost(x) & \textit{if } x \textit{ is a solid tree root} \\ mincost(x) - mincost(p(x)) & \textit{otherwise} \end{cases}$$

An example of this representation is given in Figure 35.3. Given $\Delta cost$ and Δmin, we can compute $mincost(x)$ by summing up Δmin for all vertices in the solid tree path from the root to x, and $cost(x)$ as $mincost(x) + \Delta cost(x)$. Moreover, note that $\Delta cost(x) = 0$ if and only if x is a minimum cost node in the subtree rooted at x. If this is not the case and the minimum cost node is in the right subtree, then $\Delta min(r(x)) = 0$; otherwise $\Delta min(l(x)) = 0$. With this representation, rotations can be still implemented in $O(1)$ time. The path operations can be carried out as follows:

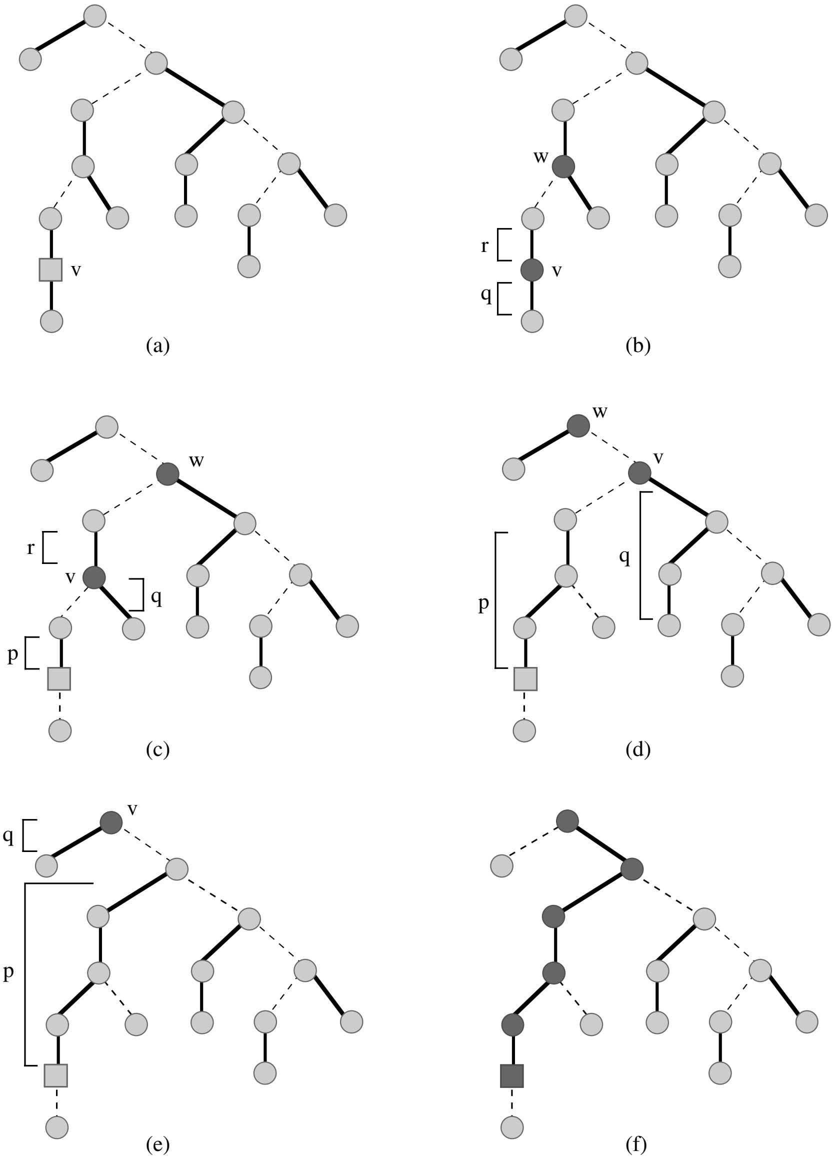


FIGURE 35.2: Effect of `expose`(v). (a) The original decomposition into solid paths; (b–e) vertices and paths after the execution of line 5 in the four consecutive iterations of the **while** loop; (f) the decomposition into solid paths after `expose`(v).

- `makepath`(v): initialize a binary tree of one vertex v with $\Delta min(v) = 0$ and $\Delta cost(v) = 0$.
- `findpath`(v): starting from v, follow parent pointers in v's solid tree until a node with no parent is found. Return this node.
- `findpathtail`(p): assuming that p is the root of a solid tree, follow right pointers and return the rightmost node in the solid tree.
- `findpathcost`(p): initialize v to p and repeat the following step until $\Delta cost(v) = 0$: if v has a right child and $\Delta min(r(v)) = 0$, replace v by $r(v)$; otherwise, replace

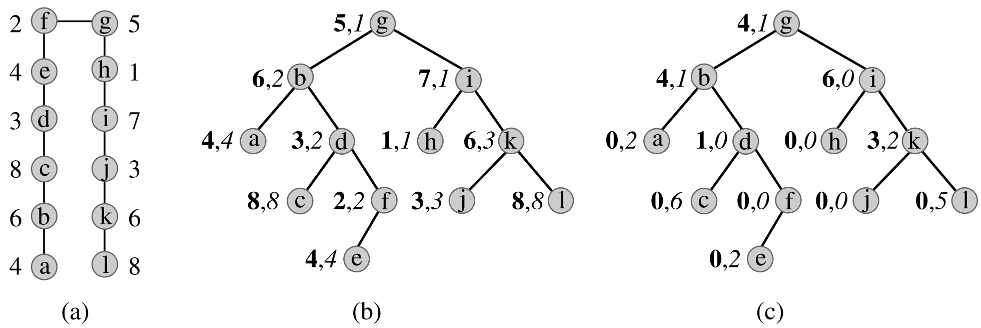


FIGURE 35.3: Representing solid paths with binary trees. (a) A solid path and its vertex costs; (b) solid tree with explicit costs (the bold value is $cost(v)$ and the italic value is $mincost(v)$); (c) corresponding solid tree with incremental costs (the bold value is $\Delta cost(v)$ and the italic value is $\Delta min(v)$).

v by $l(v)$. At the end, return v.

- `addpathcost`(p, δ): add δ to $\Delta min(p)$.
- `join`(p, v, q): join the solid trees with roots p, v and q.
- `split`(v): split the solid tree containing node v.

We observe that operations `findpath`, `findpathtail` and `findpathcost` are essentially a look up in a search tree, while `split` and `join` are exactly the same operations on search trees. If we represent solid paths by means of balanced search trees, the time per path operation becomes $O(\log n)$, and by Theorem 35.1 any sequence of m tree operations can be supported in $O(m(\log n)^2)$ time. Using self-adjusting binary search trees [16] to represent solid paths, together with a more careful analysis, yields a better bound:

THEOREM 35.2 *[15] Any sequence of m tree operations (including n* `maketree`*) requires $O(m \log n)$ time.*

Insights on the use of self-adjusting binary search trees in the implementation of path operations are given in Chapter 12. Using biased search trees [3], the $O(\log n)$ amortized bound given in Theorem 35.2 can be made worst-case. Details can be found in [15].

35.3 Topology Trees

Topology trees have been introduced by Frederickson [6] in order to maintain information about trees subject to insertions and deletions of edges and answer efficiently, e.g., tree membership queries. Similarly to the linking and cutting trees of Sleator and Tarjan [15] that we have discussed in Section 35.2, topology trees follow the idea of partitioning a tree into a set of vertex-disjoint paths. However, they are very different in how this partition is chosen, and in the data structures used to represent the paths inside the partition. Indeed, Sleator and Tarjan [15] use a simple partition of the trees based upon a careful choice of sophisticated data structures to represent paths. On the contrary, Frederickson [6] uses a more sophisticated partition that is based upon the topology of the tree; this implies more complicated algorithms but simpler data structures for representing paths.

The basic idea is to partition the tree into a suitable collection of subtrees, called *clusters*,

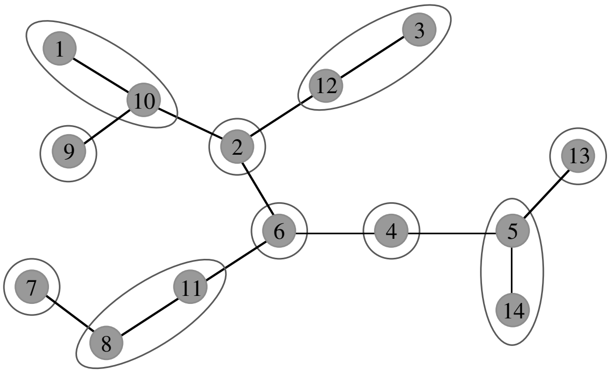


FIGURE 35.4: A restricted partition of order 2 of a tree T.

and to implement updates such that only a small number of such clusters is involved. The decomposition defined by the clusters is applied recursively to get faster update and query times.

In order to illustrate how such a recursive decomposition is computed, we assume that T has maximum vertex degree 3: this is without loss of generality, since a standard transformation can be applied if this is not the case [9]. Namely, each vertex v of degree $d > 3$ is replaced by new vertices $v_0, \ldots, v_{d-1}$; for each neighbor u_i of vertex v, $0 \leq i \leq d-1$, edge (v, u_i) is replaced by (v_i, u_i), and a new edge (v_i, v_{i+1}) is created if $i < d-1$.

Given a tree T of maximum degree 3, a *cluster* is any connected subgraph of T. The *cardinality* and the *external degree* of a cluster are the number of its vertices and the number of tree edges incident to it, respectively. We now define a partition of the vertices of T such that the resulting clusters possess certain useful properties. Let z be a positive integer.

DEFINITION 35.1 A *restricted partition* of order z w.r.t. T is a partition of the vertex set V into clusters of degree at most 3 such that:

(1) Each cluster of external degree 3 has cardinality 1.

(2) Each cluster of external degree < 3 has cardinality at most z.

(3) No two adjacent clusters can be combined and still satisfy the above.

A restricted partition of order 2 of a tree T is shown in Figure 35.4. There can be several restricted partitions for a given tree T, based upon different choices of the vertices to be unioned. For instance, vertex 8 in Figure 35.4 could be unioned with 7, instead of 11, and the partition would still be valid. It can be proved that a restricted partition of order z has $\Theta(n/z)$ clusters [6, 7].

We now show that the partition defined above can be applied recursively for $\Theta(\log n)$ levels. Such a recursive application yields a *restricted multilevel partition* [6, 7], from which the topology tree can be finally obtained.

DEFINITION 35.2 A *topology tree* is a hierarchical representation of a tree T such that each level of the topology tree partitions the vertices of T into clusters. Clusters at level 0 contain one vertex each. Clusters at level $\ell \geq 1$ form a restricted partition of order 2 of the vertices of the tree T' obtained by shrinking each cluster at level $\ell - 1$ into a single vertex.

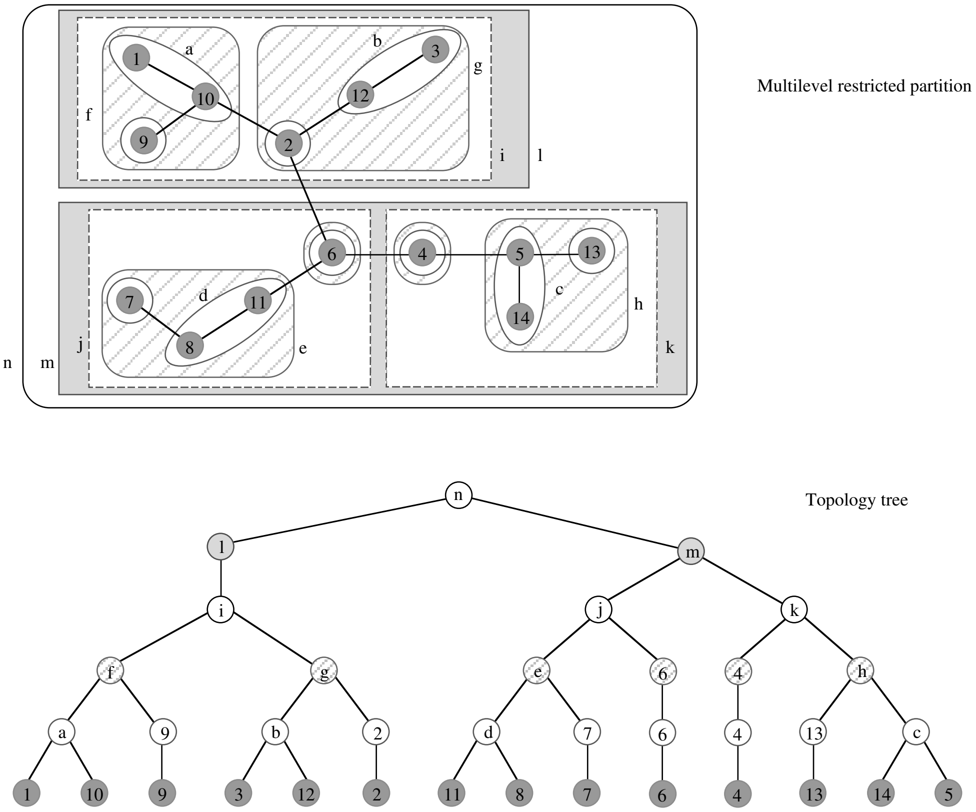


FIGURE 35.5: Restricted multilevel partition and corresponding topology tree.

As shown in Figure 35.5, level l of the restricted multilevel partition is obtained by computing a restricted partition of order 2 with respect to the tree resulting from viewing each cluster at level $l-1$ as a single vertex. Figure 35.5 also shows the topology tree corresponding to the restricted multilevel partition. Call any cluster of level $l-1$ *matched* if it is unioned with another cluster to give a cluster of level l: *unmatched* clusters have a unique child in the topology tree. It can be proved that, for any level $l > 0$ of a restricted multilevel partition, the number of matched clusters at level $l-1$ is at least $1/3$ of the total number of vertex clusters at level $l-1$. Since each pair of matched clusters is replaced by their union at level l, the number of clusters at level l is at most $5/6$ the number of clusters at level $l-1$. The number of levels of the topology tree is therefore $\Theta(\log n)$.

35.3.1 Construction

It is sufficient to show how to compute a restricted partition: the levels of the topology tree can be then built in a bottom up fashion by repeatedly applying the clustering algorithm as suggested by Definition 35.2. Because of property (3) in Definition 35.1, it is natural to compute a restricted partition according to a locally greedy heuristic, which does not always obtain the minimum number of clusters, but has the advantage of requiring only local adjustments during updates. The tree is first rooted at any vertex of degree 1 and the procedure `cluster` is called with the root as argument. At a generic step, procedure `cluster(v)` works as follows. It initializes the cluster $C(v)$ containing vertex v as $C(v) = \{v\}$. Then, for each child w of v, it recursively calls `cluster(w)`, that computes $C(w)$: if $C(w)$ can be unioned with $C(v)$ without violating the size and degree bounds in Definition 35.1, $C(v)$ is updated as $C(v) \cup C(w)$, otherwise $C(w)$ is output as a cluster. As an example, the restricted partition shown in Figure 35.4 is obtained by running procedure `cluster` on the tree rooted at vertex 7.

35.3.2 Updates

We first describe how to update the clusters of a restricted partition when an edge e is inserted in or deleted from the dynamic tree T: this operation is the crux of the update of the entire topology tree.

Update of a restricted partition. We start from edge deletion. First, removing an edge e splits T into two trees, say T_1 and T_2, which inherit all of the clusters of T, possibly with the following exceptions.

1. Edge e is entirely contained in a cluster: this cluster is no longer connected and therefore must be split. After the split, we must check whether each of the two resulting clusters is adjacent to a cluster of tree degree at most 2, and if these two adjacent clusters together have cardinality ≤ 2. If so, we combine these two clusters in order to maintain condition (3).
2. Edge e is between two clusters: in this case no split is needed. However, since the tree degree of the clusters containing the endpoints of e has been decreased, we must check if each cluster should be combined with an adjacent cluster, again because of condition (3).

Similar local manipulations can be applied to restore invariants (1) - (3) in Definition 35.1 in case of edge insertions. We now come to the update of the topology tree.

Update of the topology tree. Each level can be updated upon insertions and deletions of edges in tree T by applying few locally greedy adjustments similar to the ones described above. In particular, a constant number of basic clusters (corresponding to leaves in the topology tree) are examined: the changes in these basic clusters percolate up in the topology tree, possibly causing vertex clusters to be regrouped in different ways. The fact that only a constant amount of work has to be done on $O(\log n)$ topology tree nodes implies a logarithmic bound on the update time.

THEOREM 35.3 *[6, 7] The update of a topology tree because of an edge insertion or deletion in the dynamic tree T can be supported in $O(\log n)$ time, where n is the number of vertices of T.*

35.3.3 Applications

In the *fully dynamic tree membership* problem we would like to maintain a forest of unrooted trees under insertion of edges (which merge two trees into one), deletion of edges (which split one tree into two), and membership queries. Typical queries require to return the name of the tree containing a given vertex, or ask whether two vertices are in a same tree. Most of the solutions presented in the literature root each tree arbitrarily at one of its vertices; by keeping extra information at the root (such as the name of the tree), membership queries are equivalent to finding the tree root a vertex.

The dynamic tree clustering techniques of Frederickson have also found wide application in dynamic graph algorithms. Namely, topology trees have been originally designed in order to solve the *fully dynamic minimum spanning tree* problem [6], in which we wish to maintain a minimum spanning tree of a dynamic weighted undirected graph upon insertions/deletions of edges and edge cost changes. Let $G = (V, E)$ be the dynamic graph and let S be a designated spanning tree of G. As G is updated, edges in the spanning tree S may change: e.g., if the cost of an edge e is increased in G and e is in the spanning tree, we need to check for the existence of a replacement edge e' of smaller cost, and swap e' with e in S. The clustering approach proposed in [6, 7] consists of partitioning the vertex set V into subtrees connected in S, so that each subtree is only adjacent to a few other subtrees. A topology tree is used for representing this recursive partition of the spanning tree S. A generalization of topology trees, called *2-dimensional topology trees*, is also formed from pairs of nodes in the topology tree in order to maintain information about the edges in $E \setminus S$ [6]. Fully dynamic algorithms based only on a single level of clustering obtain typically time bounds of the order of $O(m^{2/3})$ (see for instance [8, 14]), where m is the number of edges of the graph. When the partition is applied recursively, better $O(m^{1/2})$ time bounds can be achieved by using 2-dimensional topology trees: we refer the interested reader to [6, 7] for details. As we will see in Section 36.5.2, Frederickson's algorithm is not optimal: the fully dynamic minimum spanning tree problem has been later solved in polylogarithmic time [11].

With the same technique, an $O(m^{1/2})$ time bound can be obtained also for *fully dynamic connectivity* and *2-edge connectivity* [6, 7]. For instance, [7] shows that edges and vertices can be inserted to or deleted from an undirected graph in $O(m^{1/2})$ time, and a query as to whether two vertices are in the same 2-edge-connected component can be answered in $O(\log n)$ time, n being the number of vertices. This result is based on the use of *ambivalent data structures* [7], a refinement of the clustering technique in which edges can belong to multiple groups, only one of which is actually selected depending on the topology of the given spanning tree.

35.4 Top Trees

Top trees have been introduced by Alstrup et al. [1] to maintain efficiently information about paths in trees, such as, e.g., the maximum weight on the path between any pair of vertices in a tree. The basic idea is taken from Frederickson's topology trees, but instead of partitioning vertices, top trees work by partitioning edges: the same vertex can then appear in more than one cluster. Top trees can be also seen as a natural generalization of standard balanced binary trees over dynamic collections of lists that may be concatenated and split, where each node of the balanced binary tree represents a segment of a list. As we will see, in the terminology of top trees this is just a special case of a cluster.

We follow here the presentation in [2]. Similarly to [6, 7], a *cluster* is a connected subtree of the dynamic tree T, with the additional constraint that at most two vertices, called *boundary vertices*, have edges out of the subtree. We will denote the boundary of a cluster C as δC. If the boundary contains two vertices u and v, we call *cluster path* of C the unique path between u and v in T and we denote it as $\pi(C)$. If $|\delta C < 2|$, then $\pi(C) = \emptyset$. Two clusters C_1 and C_2 are called *neighbors* if their intersection contains exactly one vertex: since clusters are connected and have no edges in common, the intersection vertex must be in $\delta C_1 \cap \delta C_2$. It is also possible to define a boundary δT, consisting of one or two vertices, for the entire tree T: we will call such vertices, if any, *external boundary vertices*. If external boundary vertices are defined, we have to extend the notion of boundary of a cluster: namely, if a cluster C contains an external boundary vertex v, then $v \in \delta C$ even if

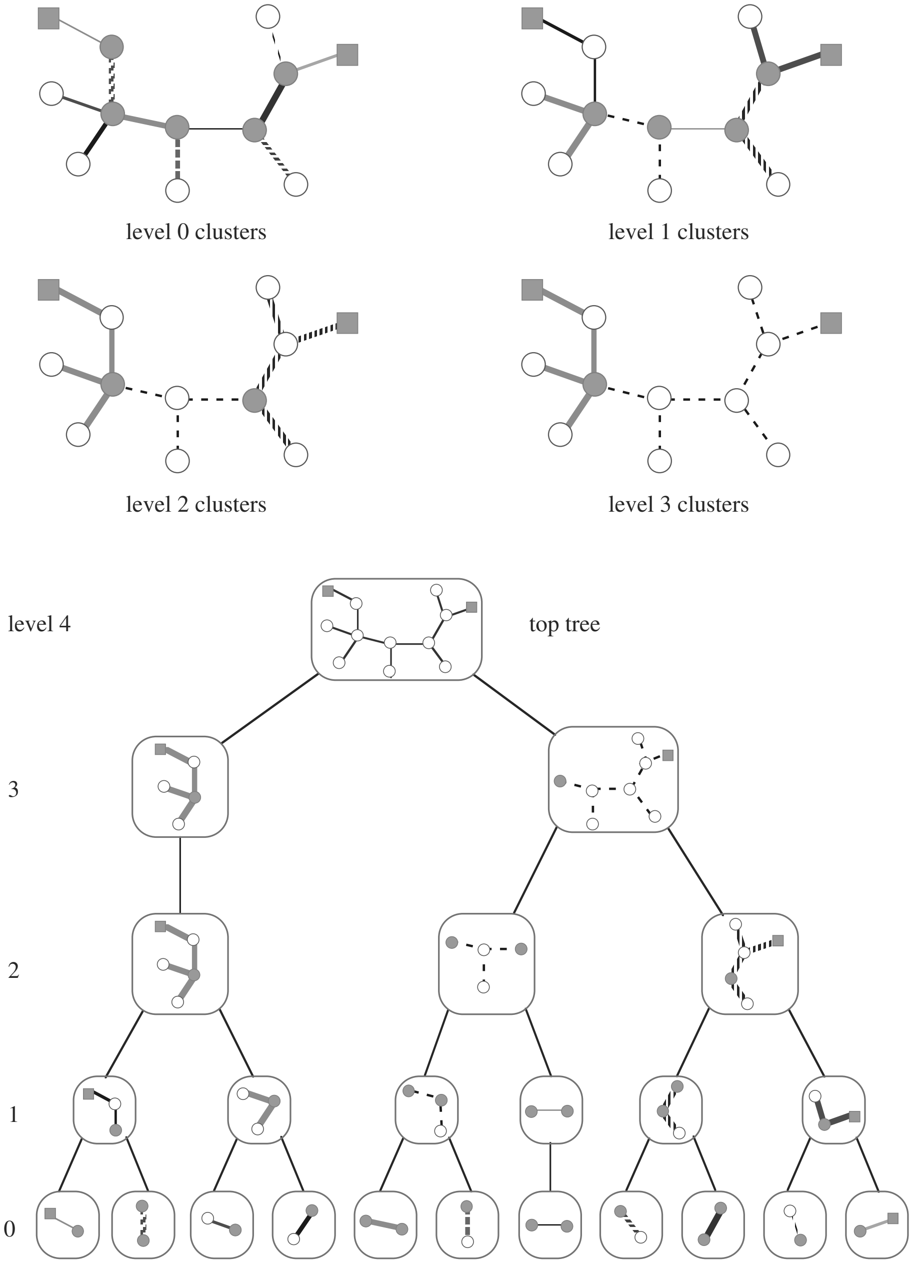


FIGURE 35.6: Clusters and top tree of a tree T. Edges with the same color, thickness, and pattern are in the same cluster. Boundary vertices are grey. External boundary vertices are squared.

v has no edge out of C.

DEFINITION 35.3 A *top tree* $\mathcal{T}$ over a pair $(T, \delta T)$ is a binary tree such that:

- The leaves of $\mathcal{T}$ are the edges of T.

- The internal nodes of $\mathcal{T}$ are clusters of T.
- The subtree represented by each internal node is the union of the subtrees represented by its two children, which must be neighbors.
- The root of $\mathcal{T}$ represents the entire tree T.
- The height of $\mathcal{T}$ is $O(\log n)$, where n is the number of vertices of T.

A tree with a single node has an empty top tree. Figure 35.6 shows the clusters at different levels of the recursive partition and the corresponding top tree. Note that a vertex can appear in many clusters, as many as $\Theta(n)$ in the worst case. However, it can be a non-boundary vertex only in $O(\log n)$ clusters. Indeed, for each vertex v which is neither an external boundary vertex nor a leaf in T, there exists a unique cluster C with children A and B such that $v \in \delta A$, $v \in \delta B$, and $v \notin \delta C$. Then v is non-boundary vertex only in cluster C and in all its ancestors in the top tree.

A locally greedy approach similar to the one described in Section 35.3.1 for topology trees can be used to build a top tree. The only modifications require to reason in terms of edges, instead of vertices, and to check the condition on the cardinality of the boundary before unioning any two neighboring clusters.

35.4.1 Updates

Given a dynamic forest, top trees over the trees of the forest are maintained under the following operations:

- `link(u,v)`, where u and v are in different trees T_u and T_v of the forest: link trees T_u and T_v by adding edge (u, v);
- `cut(e)`: remove edge e from the forest;
- `expose(u,v)`, where u and v are in the same tree T of the forest: make u and v the external boundary vertices of T and return the new root cluster of the top tree over T.

Top trees can be maintained under these operations by making use of two basic `merge` and `split` primitives:

- `merge`: it takes two top trees whose roots are neighbor clusters and joins them to form a unique top tree;
- `split`: this is the reverse operation, deleting the root of a given top tree.

The implementation of update operations starts with a sequence of `Split` of all ancestor clusters of edges whose boundary changes and finishes with a sequence of `Merge`. In case of insertion and deletions, since an end-point v of an edge has to be already boundary vertex of the edge if v is not a leaf, an update can change the boundary of at most two edges, excluding the edge being inserted/deleted. From [1, 2, 6] we have:

THEOREM 35.4 *[1, 2] For a dynamic forest we can maintain top trees of height $O(\log n)$ supporting each* `link`*,* `cut`*, and* `expose` *operation with a sequence of $O(\log n)$* `split` *and* `merge`*. The sequence itself is identified in $O(\log n)$ time. The space usage of top trees is linear in the size of the dynamic forest.*

35.4.2 Representation and Applications

Top trees are represented as standard binary trees, with pointers to parent and children for each node. With each leaf is associated the corresponding edge in T and with each internal node the at most two boundary vertices of the corresponding cluster. In addition, each vertex v of T has a pointer to the deepest cluster for which v is a non-boundary vertex, or to the root cluster containing v if v is an external boundary vertex. Given this representation, top trees can be used as a black box for maintaining different kinds of information. Typically, the user needs to attach extra information to the top tree nodes and to show how such information can be maintained upon `merge` and `split` operations.

A careful choice of the extra information makes it possible to maintain easily path properties of trees, such as the minimum weight of an edge on the path between any two vertices. In this example, the extra information w_C associated with a cluster C is the weight of the lightest edge on the cluster path $\pi(C)$. Before showing how to maintain it, note that if cluster A is a child of cluster C in the top tree and A contains an edge from $\pi(C)$, then $\pi(A) \subseteq \pi(C)$: we call A a *path child* of C. When a cluster is created by a `merge`, we store as extra information the minimum weight stored at its path children. In case of a `split`, we just discard the information. Now, in order to find the minimum weight between any two vertices u and v, we compute the root cluster C of the top tree in which u and v are external boundary vertices by calling `expose(u,v)`. Then $\pi(C)$ is the path between u and v and w_C is exactly the value we are looking for.

Top trees can be used quite easily if the property we wish to maintain is a *local* property, i.e., being satisfied by a vertex/edge in a tree implies that the property is also satisfied in all the subtrees containing the vertex/edge. Non-local properties appear to be more challenging. For general non-local searching the user has to supply a function `select` that can be used to guide a binary search towards a desired edge: given the root of a top tree, the function selects one of the two children according to the property to be maintained. Since the property is non-local, in general it is not possible to recurse directly on the selected child as is. However, Alstrup et al. [2] show that the top tree can be temporarily modified by means of a few `merge` operations so that `select` can be provided with the "right" input in the recursive call and guide the search to a correct solution.

LEMMA 35.1 Given a top tree, after $O(\log n)$ calls to `select`, `merge`, and `split`, there is a unique edge (u, v) contained in all clusters chosen by `select`, and then (u, v) is returned.

We refer the interested reader to [2] for the proof of Lemma 35.1, which shows how to modify the top tree in order to facilitate calls to `select`. We limit here to use the search as a black box in order to show how to maintain dynamically the *center* of a tree (i.e., a vertex which maximizes the distance from any other vertex) using top trees.

The extra information maintained for a cluster C with boundary vertices a and b are: the distance $dist(C)$ between a and b, and the lengths $\ell_a(C)$ and $\ell_b(C)$ of a longest path in C departing from a and b, respectively. Now we show how to compute the extra information for a cluster C obtained by merging two neighboring clusters A and B. Let c be a boundary vertex of cluster C and, w.l.o.g., let $c \in \delta A$. The longest path from c to a vertex in A has length $\ell_c(A)$. Instead, in order to get from c to a vertex in B, we must traverse a vertex, say x, such that $x \in \delta A \cap \delta B$: thus, the longest path from c to a vertex in B has length $dist(c, x) + \ell_x(B)$. This is equal to $\ell_c(B)$ if $c \in \delta B$ (i.e., if $x = c$), or to $dist(A) + \ell_x(B)$ if $c \notin \delta B$. Now, we set $\ell_c(C) = max\{\ell_c(A), dist(c, x) + \ell_x(B)\}$. We can compute $dist(C)$ similarly. Finally, function `select` can be implemented as follows. Given a cluster C with children A and B, let u be the vertex in the intersection of A and B: if $\ell_u(A) \geq \ell_u(B)$

`select` picks A, otherwise it picks B. The correctness of `select` depends on the fact that if $\ell_u(A) \geq \ell_u(B)$, then A contains all the centers of C. Using Lemma 35.1 we can finally conclude that the center of a tree can be maintained dynamically in $O(\log n)$ time.

We refer the interested reader to [1,2] for other sample applications of top trees, such as maintaining the median or the diameter of dynamic trees. As we will recall in Section 36.5.2, top trees are fundamental in the design of the fastest algorithm for the fully dynamic minimum spanning tree problem [11].

35.5 ET Trees

ET trees, first introduced in [18], have been later used to design algorithms for a variety of problems on dynamic graphs, such as fully dynamic connectivity and bipartiteness (see, e.g., [10,13]). They provide an implicit representation of dynamic forests whose vertices are associated with weighted or unweighted keys. In addition to arbitrary edge insertions and deletions, updates allow it to add or remove the weighted key associated to a vertex. Supported queries are the following:

- `connected`(u, v): tells whether vertices u and v are in the same tree.
- `size`(v): returns the number of vertices in the tree that contains v.
- `minkey`(v): returns a key of minimum weight in the tree that contains v; if keys are unweighted, an arbitrary key is returned.

The main concept for the definition of ET trees is that of Euler tour.

DEFINITION 35.4 An *Euler tour* of a tree T is a maximal closed walk over the graph obtained by replacing each edge of T by two directed edges with opposite direction.

The Euler tour of a tree can be easily computed in linear time by rooting the tree at an arbitrary vertex and running a depth first visit [4]. Each time a vertex v is encountered on the walk, we call this an *occurrence* of v and we denote it by $o(v)$. A vertex of degree Δ has exactly Δ occurrences, expect for the root which is visited $\Delta + 1$ times. Furthermore, the walk traverses each directed edge exactly once; hence, if T has n vertices, the Euler tour has length $2n - 2$. Given an n-vertex tree T, we encode it with a sequence of $2n - 1$ symbols given by an Euler tour. We refer to this encoding as $ET(T)$. For instance, the encoding of the tree given in Figure 35.7 derived from the Euler tour shown below the tree itself is $ET(T) = a\ b\ c\ b\ d\ b\ g\ f\ g\ e\ g\ b\ a$.

DEFINITION 35.5 An *ET tree* is a dynamic balanced d-ary tree over some Euler tour around T. Namely, the leaves of the ET tree are the nodes of the Euler tour, in the same order in which they appear (see Figure 35.7).

An ET tree has $O(n)$ nodes due to the linear length of the Euler tour and to properties of d-ary trees. However, since each vertex of T may occur several times in the Euler tour, an arbitrary occurrence is marked as *representative* of the vertex.

35.5.1 Updates

We first analyze how to update the encoding $ET(T)$ when T is subject to dynamic edge operations. If an edge $e = (u, v)$ is deleted from T, denote by T_u and T_v the two trees

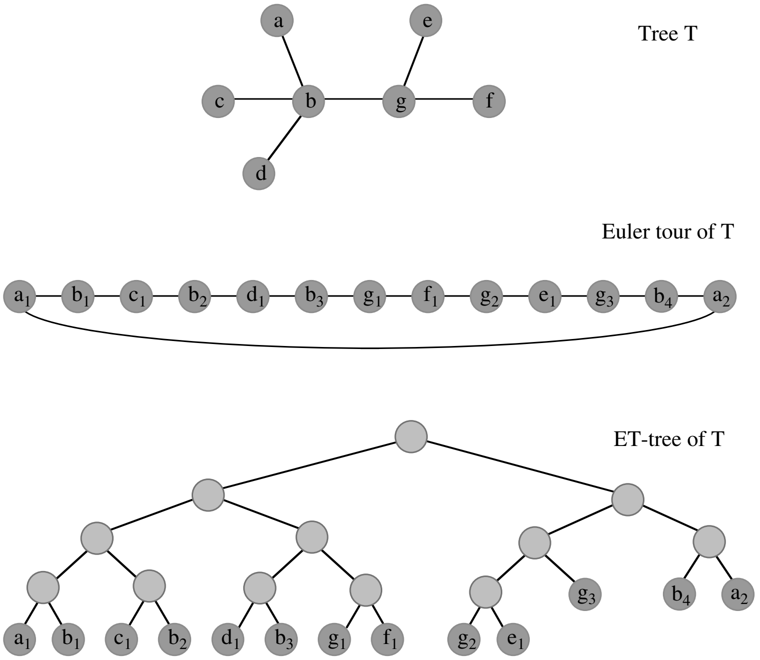

Tree T
a
e
c
b
g
f
d
Euler tour of T
a_1 b_1 c_1 b_2 d_1 b_3 g_1 f_1 g_2 e_1 g_3 b_4 a_2
ET-tree of T
g_3 b_4 a_2
a_1 b_1 c_1 b_2 d_1 b_3 g_1 f_1 g_2 e_1

FIGURE 35.7: A tree, an Euler tour, and the corresponding ET tree with $d = 2$.

obtained after the deletion, with $u \in T_u$ and $v \in T_v$. Let $o_1(u)$, $o_1(v)$, $o_2(u)$ and $o_2(v)$ be the occurrences of u and v encountered during the visit of (u, v). Without loss of generality assume that $o_1(u) < o_1(v) < o_2(v) < o_2(u)$ so that $ET(T) = \alpha o_1(u)\beta o_1(v)\gamma o_2(v)\delta o_2(u)\epsilon$. Then $ET(T_v)$ is given by the interval $o_1(v)\gamma o_2(v)$, and $ET(T_u)$ can be obtained by splicing out the interval from $ET(T)$, i.e., $ET(T_u) = \alpha o_1(u)\beta\delta o_2(u)\epsilon$.

If two trees T_1 and T_2 are joined in a new tree T because of a new edge $e = (u, v)$, with $u \in T_1$ and $v \in T_2$, we first reroot T_2 at v. Now, given $ET(T_1) = \alpha o_1(u)\beta$ and the computed encoding $ET(T_2) = o_1(v)\gamma o_2(v)$, we compute $ET(T) = \alpha o_1(u)o_1(v)\gamma o_2(v)o(u)\beta$, where $o(u)$ is a newly created occurrence of vertex u. To complete the description, we need to show how to change the root of a tree T from r to another vertex s. Let $ET(T) = o(r)\alpha o_1(s)\beta$, where $o_1(s)$ is any occurrence of s. Then, the new encoding will be $o_1(s)\beta\alpha o(s)$, where $o(s)$ is a newly created occurrence of s that is added at the end of the new sequence.

In summary, if trees in the forest are linked or cut, a constant number of the following operations is required: (i) splicing out an interval from the encoding, (ii) inserting an interval into the encoding, (iii) inserting or (iv) deleting a single occurrence from the encoding. If the encoding $ET(T)$ is stored in a balanced search tree of degree d, then one may perform each operation in time $O(d \log_d n)$ while maintaining the balance of the tree.

35.5.2 Applications

The query connected(u, v) can be easily supported in time $O(\log n/d)$ by finding the roots of the ET trees containing u and v and checking if they coincide. The same time is sufficient to check whether one element precedes another element in the ordering.

To support size and minkey queries, each node q of the ET tree maintains two additional values: the number $s(q)$ of representatives below it and the minimum weight key $k(q)$ attached to a representative below it. Such values can be easily maintained during updates and allow it to answer queries of the form `size`(v) and `minkey`(v) in $O(\log n/d)$ time for any vertex v of the forest: the root r of the ET tree containing v is found and values $s(r)$ and $k(r)$ are returned, respectively. We refer the interested reader to [10] for additional details of the method.

In Section 36.4 we will see how ET trees have been used [10, 11] to design a polylogarithmic algorithm for fully dynamic connectivity. Here we limit to observe that trees in a spanning forest of the dynamic graph are maintained using the Euler tour data structure, and therefore updates and connectivity queries within the forest can be supported in logarithmic time. The use of randomization and of a logarithmic decomposition makes it possible to maintain also non-tree edges in polylogarithmic time upon changes in the graph.

35.6 Reachability Trees

In this section we consider a tree data structure that has been widely used to solve dynamic path problems on directed graphs.

The first appearance of this tool dates back to 1981, when Even and Shiloach showed how to maintain a breadth-first tree of an undirected graph under any sequence of edge deletions [5]; they used this as a kernel for decremental connectivity on undirected graphs. Later on, Henzinger and King [10] showed how to adapt this data structure to fully dynamic transitive closure in directed graphs. King [12] designed an extension of this tree data structure to weighted directed graphs for solving fully dynamic transitive closure (see Section 36.6.1) and all pairs shortest paths (see Section 36.7.1).

In the unweighted directed version, the goal is to maintain information about breadth-first search (BFS) on a directed graph G undergoing deletions of edges. In particular, in the context of dynamic path problems, we are interested in maintaining BFS trees of depth up to d, with $d \leq n$. Given a directed graph $G = (V, E)$ and a vertex $r \in V$, we would like to support any intermixed sequence of the following operations:

`Delete`(x, y): delete edge (x, y) from G.

`Level`(u): return the level of vertex u in the BFS tree of depth d rooted at r (return $+\infty$ if u is not reachable from r within distance d).

In [12] it is shown how to maintain efficiently the BFS levels, supporting any `Level` operation in constant time and any sequence of `Delete` operations in $O(md)$ overall time:

LEMMA 35.2 (King [12]) Maintaining BFS levels up to depth d from a given root requires $O(md)$ time in the worst case throughout any sequence of edge deletions in a directed graph with m initial edges.

Lemma 35.2 implies that maintaining BFS levels requires d times the time needed for constructing them. Since $d \leq n$, we obtain a total bound of $O(mn)$ if there are no limits on the depth of the BFS levels.

As it was shown in [10, 12], it is possible to extend the BFS data structure presented in this section to deal with weighted directed graphs. In this case, a shortest path tree is maintained in place of BFS levels: after each edge deletion or edge weight increase, the tree is reconnected by essentially mimicking Dijkstra's algorithm rather than BFS. Details can be found in [12].

35.7 Conclusions

In this chapter we have surveyed data structures for maintaining properties of dynamically changing trees, focusing our attention on linking and cutting trees [15], topology trees [6], top trees [1], ET trees [10, 18], and reachability trees [5]. We have shown that these data structures typically support updates and many different kinds of queries in logarithmic (amortized or worst-case) time. Hence, problems such as tree membership, maintaining the center or diameter of a tree, finding the minimum cost on a given path can be solved in $O(\log n)$ time in a fully dynamic setting.

All the data structures for maintaining properties of dynamically changing trees are not only important and interesting on their own, but are often used as building blocks by many dynamic graph algorithms, as we will see in Chapter 36. Some of these data structures, such as the union find data structures and the linking and cutting trees of Sleator and Tarjan have myriads of applications in other problems, such as implementing property grammars, computational geometry problems, testing equivalence of finite state machines, computing the longest common subsequence of two sequences, performing unification in logic programming and theorem proving, finding minimum spanning trees, and maximum flow algorithms. Since all these problems are outside the scope of this survey, we have not mentioned these applications and have restricted our attention to the applications of these data structures to dynamic tree and graph algorithms only.

Acknowledgments

This work has been supported in part by the IST Programmes of the EU under contract numbers IST-1999-14186 (ALCOM-FT) and IST-2001-33555 (COSIN), and by the Italian Ministry of University and Scientific Research (Project “ALINWEB: Algorithmics for Internet and the Web”).

References

[1] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Minimizing diameters of dynamic trees. In *Proc. 24th Int. Colloquium on Automata, Languages and Programming (ICALP 97)*, LNCS 1256, pages 270–280, 1997.

[2] S. Alstrup, J. Holm, and M. Thorup. Maintaining center and median in dynamic trees. In *Proc. 7th Scandinavian Workshop on Algorithm Theory (SWAT 00)*, pages 46–56, 2000.

[3] S. W. Bent, D. D. Sleator, and R. E. Tarjan. Biased search trees. *SIAM J. Comput.*, 14(3):545–568, 1985.

[4] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*, Second Edition. The MIT Press, 2001.

[5] S. Even and Y. Shiloach. An on-line edge deletion problem. *J. Assoc. Comput. Mach.*, 28:1–4, 1981.

[6] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees. *SIAM J. Comput.*, 14:781–798, 1985.

[7] G. N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM J. Comput.*, 26(2): 484–538, 1997.

[8] Z. Galil and G. F. Italiano. Fully-dynamic algorithms for 2-edge connectivity. *SIAM J. Comput.*, 21:1047–1069, 1992.

[9] F. Harary. *Graph Theory*. Addison-Wesley, Reading, MA, 1969.

[10] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. Assoc. Comput. Mach.*, 46(4):502–536, 1999.

[11] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. Assoc. Comput. Mach.*, 48(4):723–760, 2001.

[12] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proc. 40-th Symposium on Foundations of Computer Science (FOCS 99)*, 1999.

[13] P. B. Miltersen, S. Subramanian, J. S. Vitter, and R. Tamassia. Complexity models for incremental computation. *Theoret. Comput. Science*, 130:203–236, 1994.

[14] M. Rauch. Fully dynamic biconnectivity in graphs. *Algorithmica*, 13:503-538, 1995.

[15] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comp. Syst. Sci.*, 24:362–381, 1983.

[16] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. Assoc. Comput. Mach.*, 32:652–686, 1985.

[17] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *J. Assoc. Comput. Mach.*, 31:245–281, 1984.

[18] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. on Computing*, 14:862–874, 1985.

36

Dynamic Graphs

Camil Demetrescu
Università di Roma

Irene Finocchi
Università di Roma

Giuseppe F. Italiano
Università di Roma

36.1 Introduction

In many applications of graph algorithms, including communication networks, VLSI design, graphics, and assembly planning, graphs are subject to discrete changes, such as insertions or deletions of vertices or edges. In the last two decades there has been a growing interest in such dynamically changing graphs, and a whole body of algorithmic techniques and data structures for dynamic graphs has been discovered. This chapter is intended as an overview of this field.

An *update on a graph* is an operation that inserts or deletes edges or vertices of the graph or changes attributes associated with edges or vertices, such as cost or color. Throughout this chapter by *dynamic graph* we denote a graph that is subject to a sequence of updates. In a typical dynamic graph problem one would like to answer queries on dynamic graphs, such as, for instance, whether the graph is connected or which is the shortest path between any two vertices. The goal of a dynamic graph algorithm is to update efficiently the solution of a problem after dynamic changes, rather than having to recompute it from scratch each time. Given their powerful versatility, it is not surprising that dynamic algorithms and dynamic data structures are often more difficult to design and to analyze than their static counterparts.

We can classify dynamic graph problems according to the types of updates allowed. In particular, a dynamic graph problem is said to be *fully dynamic* if the update operations include unrestricted insertions and deletions of edges or vertices. A dynamic graph problem is said to be *partially dynamic* if only one type of update, either insertions or deletions, is allowed. More specifically, a dynamic graph problem is said to be *incremental* if only insertions are allowed, while it is said to be *decremental* if only deletions are allowed.

In the first part of this chapter we will present the main algorithmic techniques used to solve dynamic problems on both *undirected* and *directed* graphs. In the second part of the chapter we will deal with dynamic problems on graphs, and we will investigate as paradigmatic problems the dynamic maintenance of minimum spanning trees, connectivity, transitive closure and shortest paths. Interestingly enough, dynamic problems on directed graphs seem much harder to solve than their counterparts on undirected graphs, and require completely different techniques and tools.

36.2 Techniques for Undirected Graphs

Many of the algorithms proposed in the literature use the same general techniques, and hence we begin by describing these techniques. In this section we focus on undirected graphs, while techniques for directed graphs will be discussed in Section 36.3. Typically, most of these techniques use some sort of graph decomposition, and partition either the vertices or the edges of the graph to be maintained. Moreover, data structures that maintain properties of dynamically changing trees, such as the ones described in Chapter 35 (linking and cutting trees, topology trees, and Euler tour trees), are often used as building blocks by many dynamic graph algorithms.

36.2.1 Clustering

The clustering technique has been introduced by Frederickson [13] and is based upon partitioning the graph into a suitable collection of connected subgraphs, called *clusters*, such that each update involves only a small number of such clusters. Typically, the decomposition defined by the clusters is applied recursively and the information about the subgraphs is combined with the topology trees described in Section 35.3. A refinement of the clustering technique appears in the idea of *ambivalent data structures* [14], in which edges can belong to multiple groups, only one of which is actually selected depending on the topology of the given spanning tree.

As an example, we briefly describe the application of clustering to the problem of maintaining a minimum spanning forest [13]. Let $G = (V, E)$ be a graph with a designated spanning tree S. Clustering is used for partitioning the vertex set V into subtrees connected in S, so that each subtree is only adjacent to a few other subtrees. A topology tree, as described in Section 35.3, is then used for representing a recursive partition of the tree S. Finally, a generalization of topology trees, called *2-dimensional topology trees*, are formed from pairs of nodes in the topology tree and allow it to maintain information about the edges in $E \setminus S$ [13].

Fully dynamic algorithms based only on a single level of clustering obtain typically time bounds of the order of $O(m^{2/3})$ (see for instance [17, 32]). When the partition can be applied recursively, better $O(m^{1/2})$ time bounds can be achieved by using 2-dimensional topology trees (see, for instance, [13, 14]).

THEOREM 36.1 *(Frederickson [13]) The minimum spanning forest of an undirected graph can be maintained in time $O(\sqrt{m})$ per update, where m is the current number of edges in the graph.*

We refer the interested reader to [13, 14] for details about Frederickson's algorithm. With the same technique, an $O(\sqrt{m})$ time bound can be obtained also for fully dynamic connectivity and 2-edge connectivity [13, 14]

The type of clustering used can very problem-dependent, however, and makes this technique difficult to be used as a black box.

36.2.2 Sparsification

Sparsification is a general technique due to Eppstein *et al.* [10] that can be used as a black box (without having to know the internal details) in order to design and dynamize graph algorithms. It is a divide-and-conquer technique that allows it to reduce the dependence on the number of edges in a graph, so that the time bounds for maintaining some property of the graph match the times for computing in sparse graphs. More precisely, when the technique is applicable, it speeds up a $T(n, m)$ time bound for a graph with n vertices and m edges to $T(n, O(n))$, i.e., to the time needed if the graph were sparse. For instance, if $T(n, m) = O(\sqrt{m}\,)$, we get a better bound of $O(\sqrt{n}\,)$. The technique itself is quite simple. A key concept is the notion of certificate.

DEFINITION 36.1 For any graph property P and graph G, a *certificate* for G is a graph G' such that G has property P if and only if G' has the property.

Let G be a graph with m edges and n vertices. We partition the edges of G into a collection of $O(m/n)$ sparse subgraphs, i.e., subgraphs with n vertices and $O(n)$ edges. The information relevant for each subgraph can be summarized in a sparse certificate. Certificates are then merged in pairs, producing larger subgraphs which are made sparse by again computing their certificate. The result is a balanced binary tree in which each node is represented by a sparse certificate. Each update involves $O(\log(m/n))^*$ graphs with $O(n)$ edges each, instead of one graph with m edges.

There exist two variants of sparsification. The first variant is used in situations where no previous fully dynamic algorithm is known. A static algorithm is used for recomputing a sparse certificate in each tree node affected by an edge update. If the certificates can be found in time $O(m + n)$, this variant gives time bounds of $O(n)$ per update.

In the second variant, certificates are maintained using a dynamic data structure. For this to work, a *stability* property of certificates is needed, to ensure that a small change in the input graph does not lead to a large change in the certificates. We refer the interested reader to [10] for a precise definition of stability. This variant transforms time bounds of the form $O(m^p)$ into $O(n^p)$.

DEFINITION 36.2 A time bound $T(n)$ is *well-behaved* if, for some $c < 1$, $T(n/2) < cT(n)$. Well-behavedness eliminates strange situations in which a time bound fluctuates wildly with n. For instance, all polynomials are well-behaved.

THEOREM 36.2 *(Eppstein et al. [10]) Let P be a property for which we can find sparse*

*Throughout this chapter, we assume that $\log x$ stands for $\max\{1, \log_2 x\}$, so $\log(m/n)$ is never smaller than 1 even if $m < 2n$.

certificates in time $f(n,m)$ for some well-behaved f, and such that we can construct a data structure for testing property P in time $g(n,m)$ which can answer queries in time $q(n,m)$. Then there is a fully dynamic data structure for testing whether a graph has property P, for which edge insertions and deletions can be performed in time $O(f(n,O(n))) + g(n,O(n))$, and for which the query time is $q(n,O(n))$.

THEOREM 36.3 *(Eppstein et al. [10]) Let P be a property for which stable sparse certificates can be maintained in time $f(n,m)$ per update, where f is well-behaved, and for which there is a data structure for property P with update time $g(n,m)$ and query time $q(n,m)$. Then P can be maintained in time $O(f(n,O(n))) + g(n,O(n))$ per update, with query time $q(n,O(n))$.*

Basically, the first version of sparsification (Theorem 36.2) can be used to dynamize static algorithms, in which case we only need to *compute* efficiently *sparse* certificates, while the second version (Theorem 36.3) can be used to speed up existing fully dynamic algorithms, in which case we need to *maintain* efficiently *stable sparse* certificates.

Sparsification applies to a wide variety of dynamic graph problems, including minimum spanning forests, edge and vertex connectivity. As an example, for the fully dynamic minimum spanning tree problem, it reduces the update time from $O(\sqrt{m}\,)$ [13, 14] to $O(\sqrt{n}\,)$ [10].

Since sparsification works on top of a given algorithm, we need not to know the internal details of this algorithm. Consequently, it can be applied orthogonally to other data structuring techniques: in a large number of situations both clustering and sparsification have been combined to produce an efficient dynamic graph algorithm.

36.2.3 Randomization

Clustering and sparsification allow one to design efficient deterministic algorithms for fully dynamic problems. The last technique we present in this section is due to Henzinger and King [20], and allows one to achieve faster update times for some problems by exploiting the power of randomization.

We now sketch how the randomization technique works taking the fully dynamic connectivity problem as an example. Let $G = (V,E)$ be a graph to be maintained dynamically, and let F be a spanning forest of G. We call edges in F *tree edges*, and edges in $E \setminus F$ *non-tree edges.* The algorithm by Henzinger and King is based on the following ingredients.

Maintaining Spanning Forests

Trees are maintained using the Euler Tours data structure (ET trees) described in Section 35.5: this allows one to obtain logarithmic updates and queries within the forest.

Random Sampling

Another key idea is the following: when e is deleted from a tree T, use random sampling among the non-tree edges incident to T, in order to find quickly a replacement edge for e, if any.

Graph Decomposition

The last key idea is to combine randomization with a suitable graph decomposition. We maintain an edge decomposition of the current graph G using $O(\log n)$ edge disjoint subgraphs $G_i = (V,E_i)$. These subgraphs are hierarchically ordered. The lower levels contain tightly-connected portions of G (i.e., dense edge cuts), while the higher levels contain loosely-connected portions of G (i.e., sparse cuts). For each level i, a spanning forest for the graph defined by all the edges in levels i or below is also maintained.

Note that the hard operation in this problem is the deletion of a tree edge: indeed, a spanning forest is easily maintained with the help of the linking and cutting trees described in Section 35.2 throughout edge insertions, and deleting a non-tree edge does not change the forest.

The goal is an update time of $O(\log^3 n)$: after an edge deletion, in the quest for a replacement edge, we can afford a number of sampled edges of $O(\log^2 n)$. However, if the candidate set of edge e is a small fraction of all non-tree edges which are adjacent to T, it is unlikely to find a replacement edge for e among this small sample. If we found no candidate among the sampled edges, we must check explicitly all the non-tree edges adjacent to T. After random sampling has failed to produce a replacement edge, we need to perform this check explicitly, otherwise we would not be guaranteed to provide correct answers to the queries. Since there might be a lot of edges which are adjacent to T, this explicit check could be an expensive operation, so it should be made a low probability event for the randomized algorithm. This can produce pathological updates, however, since deleting all edges in a relatively small candidate set, reinserting them, deleting them again, and so on will almost surely produce many of those unfortunate events.

The graph decomposition is used to prevent the undesirable behavior described above. If a spanning forest edge e is deleted from a tree at some level i, random sampling is used to quickly find a replacement for e at that level. If random sampling succeeds, the tree is reconnected at level i. If random sampling fails, the edges that can replace e in level i form with high probability a sparse cut. These edges are moved to level $i+1$ and the same procedure is applied recursively on level $i+1$.

THEOREM 36.4 *(Henzinger and King [20]) Let G be a graph with m_0 edges and n vertices subject to edge deletions only. A spanning forest of G can be maintained in $O(\log^3 n)$ expected amortized time per deletion, if there are at least $\Omega(m_0)$ deletions. The time per query is $O(\log n)$.*

36.3 Techniques for Directed Graphs

In this section we discuss the main techniques used to solve dynamic path problems on directed graphs. We first address combinatorial and algebraic properties, and then we consider some efficient data structures, which are used as building blocks in designing dynamic algorithms for transitive closure and shortest paths. Similarly to the case of undirected graphs seen in Section 36.2, data structures that maintain properties of dynamically changing trees, such as the ones described in Chapter 35 (reachability trees), are often used as building blocks by many dynamic graph algorithms.

36.3.1 Kleene Closures

Path problems such as transitive closure and shortest paths are tightly related to matrix sum and matrix multiplication over a closed semiring (see [5] for more details). In particular, the transitive closure of a directed graph can be obtained from the adjacency matrix of the graph via operations on the semiring of Boolean matrices, that we denote by $\{+, \cdot, 0, 1\}$. In this case, $+$ and $\cdot$ denote the usual sum and multiplication over Boolean matrices.

LEMMA 36.1 Let $G = (V, E)$ be a directed graph and let $TC(G)$ be the (reflexive) transitive closure of G. If X is the Boolean adjacency matrix of G, then the Boolean adjacency matrix of $TC(G)$ is the Kleene closure of X on the $\{+, \cdot, 0, 1\}$ Boolean semiring:

$$X^* = \sum_{i=0}^{n-1} X^i.$$

Similarly, shortest path distances in a directed graph with real-valued edge weights can be obtained from the weight matrix of the graph via operations on the semiring of real-valued matrices, that we denote by $\{\oplus, \odot, \mathcal{R}\}$, or more simply by $\{\min, +\}$. Here, $\mathcal{R}$ is the set of real values and $\oplus$ and $\odot$ are defined as follows. Given two real-valued matrices A and B, $C = A \odot B$ is the matrix product such that $C[x, y] = \min_{1 \le z \le n}\{A[x, z] + B[z, y]\}$ and $D = A \oplus B$ is the matrix sum such that $D[x, y] = \min\{A[x, y], B[x, y]\}$. We also denote by AB the product $A \odot B$ and by $AB[x, y]$ entry (x, y) of matrix AB.

LEMMA 36.2 Let $G = (V, E)$ be a weighted directed graph with no negative-length cycles. If X is a weight matrix such that $X[x, y]$ is the weight of edge (x, y) in G, then the distance matrix of G is the Kleene closure of X on the $\{\oplus, \odot, \mathcal{R}\}$ semiring:

$$X^* = \bigoplus_{i=0}^{n-1} X^i.$$

We now briefly recall two well-known methods for computing the Kleene closure X^* of X. In the following, we assume that X is an $n \times n$ matrix.

Logarithmic Decomposition

A simple method to compute X^*, based on repeated squaring, requires $O(n^\mu \cdot \log n)$ worst-case time, where $O(n^\mu)$ is the time required for computing the product of two matrices over a closed semiring. This method performs $\log_2 n$ sums and products of the form $X_{i+1} = X_i + X_i^2$, where $X = X_0$ and $X^* = X_{\log_2 n}$.

Recursive Decomposition

Another method, due to Munro [28], is based on a Divide and Conquer strategy and computes X^* in $O(n^\mu)$ worst-case time. Munro observed that, if we partition X in four submatrices A, B, D, C of size $n/2 \times n/2$ (considered in clockwise order), and X^* similarly in four submatrices E, F, H, G of size $n/2 \times n/2$, then X^* is defined recursively according to the following equations:

$$E = (A + BD^*C)^* \quad | \quad F = EBD^* \quad | \quad G = D^*CE \quad | \quad H = D^* + D^*CEBD^*$$

Surprisingly, using this decomposition the cost of computing X^* starting from X is asymptotically the same as the cost of multiplying two matrices over a closed semiring.

36.3.2 Long Paths

In this section we recall an intuitive combinatorial property of long paths in a graph. Namely, if we pick a subset S of vertices at random from a graph G, then a sufficiently long path will intersect S with high probability. This can be very useful in finding a long path by using short searches.

To the best of our knowledge, the long paths property was first given in [18], and later on it has been used many times in designing efficient algorithms for transitive closure and shortest paths (see e.g., [7, 24, 37, 38]). The following theorem is from [37].

THEOREM 36.5 *(Ullman and Yannakakis [37]) Let $S \subseteq V$ be a set of vertices chosen uniformly at random. Then the probability that a given simple path has a sequence of more than $(cn \log n)/|S|$ vertices, none of which are from S, for any $c > 0$, is, for sufficiently large n, bounded by $2^{-\alpha c}$ for some positive α.*

Zwick [38] showed it is possible to choose set S deterministically by a reduction to a hitting set problem [3, 27]. King used a similar idea for maintaining fully dynamic shortest paths [24].

36.3.3 Locality

Recently, Demetrescu and Italiano [8] proposed a new approach to dynamic path problems based on maintaining classes of paths characterized by local properties, i.e., properties that hold for all proper subpaths, even if they may not hold for the entire paths. They showed that this approach can play a crucial role in the dynamic maintenance of shortest paths. For instance, they considered a class of paths defined as follows:

DEFINITION 36.3 A path π in a graph is *locally shortest* if and only if every proper subpath of π is a shortest path.

This definition is inspired by the optimal-substructure property of shortest paths: all subpaths of a shortest path are shortest. However, a locally shortest path may not be shortest.

The fact that locally shortest paths include shortest paths as a special case makes them an useful tool for computing and maintaining distances in a graph. Indeed, paths defined locally have interesting combinatorial properties in dynamically changing graphs. For example, it is not difficult to prove that the number of locally shortest paths that may change due to an edge weight update is $O(n^2)$ if updates are partially dynamic, i.e., increase-only or decrease-only:

THEOREM 36.6 *Let G be a graph subject to a sequence of increase-only or decrease-only edge weight updates. Then the amortized number of paths that start or stop being locally shortest at each update is $O(n^2)$.*

Unfortunately, Theorem 36.6 does not hold if updates are fully dynamic, i.e., increases and decreases of edge weights are intermixed. To cope with pathological sequences, a possible solution is to retain information about the history of a dynamic graph, considering the following class of paths:

DEFINITION 36.4 A *historical shortest path* (in short, *historical path*) is a path that has been shortest at least once since it was last updated.

Here, we assume that a path is updated when the weight of one of its edges is changed. Applying the locality technique to historical paths, we derive locally historical paths:

DEFINITION 36.5 A path π in a graph is *locally historical* if and only if every proper subpath of π is historical.

Like locally shortest paths, also locally historical paths include shortest paths, and this makes them another useful tool for maintaining distances in a graph:

LEMMA 36.3 If we denote by SP, LSP, and LHP respectively the sets of shortest paths, locally shortest paths, and locally historical paths in a graph, then at any time the following inclusions hold: $SP \subseteq LSP \subseteq LHP$.

Differently from locally shortest paths, locally historical paths exhibit interesting combinatorial properties in graphs subject to fully dynamic updates. In particular, it is possible to prove that the number of paths that become locally historical in a graph at each edge weight update depends on the number of historical paths in the graph.

THEOREM 36.7 *(Demetrescu and Italiano [8]) Let G be a graph subject to a sequence of update operations. If at any time throughout the sequence of updates there are at most $O(h)$ historical paths in the graph, then the amortized number of paths that become locally historical at each update is $O(h)$.*

To keep changes in locally historical paths small, it is then desirable to have as few historical paths as possible. Indeed, it is possible to transform every update sequence into a slightly longer equivalent sequence that generates only a few historical paths. In particular, there exists a simple *smoothing* strategy that, given any update sequence Σ of length k, produces an operationally equivalent sequence $F(\Sigma)$ of length $O(k \log k)$ that yields only $O(\log k)$ historical shortest paths between each pair of vertices in the graph. We refer the interested reader to [8] for a detailed description of this smoothing strategy. According to Theorem 36.7, this technique implies that only $O(n^2 \log k)$ locally historical paths change at each edge weight update in the smoothed sequence $F(\Sigma)$.

As elaborated in [8], locally historical paths can be maintained very efficiently. Since by Lemma 36.3 locally historical paths include shortest paths, this yields the fastest known algorithm for fully dynamic all pairs shortest paths (see Section 36.7.2).

36.3.4 Matrices

Another useful data structure for keeping information about paths in dynamic directed graphs is based on matrices subject to dynamic changes. As we have seen in Section 36.3.1, Kleene closures can be constructed by evaluating polynomials over matrices. It is therefore natural to consider data structures for maintaining polynomials of matrices subject to updates of entries, like the one introduced in [6].

In the case of Boolean matrices, the problem can be stated as follows. Let P be a polynomial over $n \times n$ Boolean matrices with constant degree, constant number of terms, and variables $X_1 \ldots X_k$. We wish to maintain a data structure for P subject to any intermixed sequence of update and query operations of the following kind:

`SetRow`$(i, \Delta X, X_b)$: sets to one the entries in the i-th row of variable X_b of polynomial

P corresponding to one-valued entries in the i-th row of matrix ΔX.

`SetCol`$(i, \Delta X, X_b)$: sets to one the entries in the i-th column of variable X_b of polynomial P corresponding to one-valued entries in the i-th column of matrix ΔX.

`Reset`$(\Delta X, X_b)$: resets to zero the entries of variable X_b of polynomial P corresponding to one-valued entries in matrix ΔX.

`Lookup`$()$: returns the maintained value of P.

We add to the previous four operations a further update operation especially designed for maintaining path problems:

`LazySet`$(\Delta X, X_b)$: sets to 1 the entries of variable X_b of P corresponding to one-valued entries in matrix ΔX. However, the maintained value of P might not be immediately affected by this operation.

Let C_P be the correct value of P that we would have by recomputing it from scratch after each update, and let M_P be the actual value that we maintain. If no `LazySet` operation is ever performed, then always $M_P = C_P$. Otherwise, M_P is not necessarily equal to C_P, and we guarantee the following weaker property on M_P: if $C_P[u, v]$ flips from 0 to 1 due to a `SetRow`/`SetCol` operation on a variable X_b, then $M_P[u, v]$ flips from 0 to 1 as well. This means that `SetRow` and `SetCol` always correctly reveal new 1's in the maintained value of P, possibly taking into account the 1's inserted through previous `LazySet` operations. This property is crucial for dynamic path problems, since the appearance of new paths in a graph after an edge insertion, which corresponds to setting a bit to one in its adjacency matrix, is always correctly recorded in the data structure.

LEMMA 36.4 (Demetrescu and Italiano [6]) Let P be a polynomial with constant degree of matrices over the Boolean semiring. Any `SetRow`, `SetCol`, `LazySet`, and `Reset` operation on a polynomial P can be supported in $O(n^2)$ amortized time. `Lookup` queries are answered in optimal time.

Similar data structures can be given for settings different from the semiring of Boolean matrices. In particular, in [7] the problem of maintaining polynomials of matrices over the $\{\min, +\}$ semiring is addressed.

The running time of operations for maintaining polynomials in this semiring is given below.

THEOREM 36.8 *(Demetrescu and Italiano [7]) Let P be a polynomial with constant degree of matrices over the $\{\min, +\}$ semiring. Any* `SetRow`*,* `SetCol`*,* `LazySet`*, and* `Reset` *operation on variables of P can be supported in $O(D \cdot n^2)$ amortized time, where D is the maximum number of different values assumed by entries of variables during the sequence of operations.* `Lookup` *queries are answered in optimal time.*

36.4 Connectivity

In this section and in Section 36.5 we consider dynamic problems on undirected graphs, showing how to deploy some of the techniques and data structures presented in Section 36.2 to obtain efficient algorithms. These algorithms maintain efficiently some property of an undirected graph that is undergoing structural changes defined by insertion and deletion of edges, and/or updates of edge costs. To check the graph property throughout a sequence of these updates, the algorithms must be prepared to answer queries on the graph property efficiently. In particular, in this section we address the *fully dynamic connectivity* problem, where we are interested in algorithms that are capable of inserting edges, deleting edges, and answering a query on whether the graph is connected, or whether two vertices are in the same connected component. We recall that the goal of a dynamic algorithm is to minimize the amount of recomputation required after each update. All the dynamic algorithms that we describe are able to maintain dynamically the graph property at a cost (per update operation) which is significantly smaller than the cost of recomputing the graph property from scratch.

36.4.1 Updates in $O(\log^2 n)$ Time

In this section we give a high level description of the fastest deterministic algorithm for the fully dynamic connectivity problem in undirected graphs [22]: the algorithm, due to Holm, de Lichtenberg and Thorup, answers connectivity queries in $O(\log n/\log\log n)$ worst-case running time while supporting edge insertions and deletions in $O(\log^2 n)$ amortized time.

Similarly to the randomized algorithm in [20], the deterministic algorithm in [22] maintains a spanning forest F of the dynamically changing graph G. As above, we will refer to the edges in F as *tree edges.* Let e be a tree edge of forest F, and let T be the tree of F containing it. When e is deleted, the two trees T_1 and T_2 obtained from T after the deletion of e can be reconnected if and only if there is a non-tree edge in G with one endpoint in T_1 and the other endpoint in T_2. We call such an edge a *replacement edge* for e. In other words, if there is a replacement edge for e, T is reconnected via this replacement edge; otherwise, the deletion of e creates a new connected component in G.

To accommodate systematic search for replacement edges, the algorithm associates to each edge e a level $\ell(e)$ and, based on edge levels, maintains a set of sub-forests of the spanning forest F: for each level i, forest F_i is the sub-forest induced by tree edges of level $\geq i$. If we denote by L denotes the maximum edge level, we have that:

$$F = F_0 \supseteq F_1 \supseteq F_2 \supseteq \ldots \supseteq F_L,$$

Initially, all edges have level 0; levels are then progressively increased, but never decreased. The changes of edge levels are accomplished so as to maintain the following invariants, which obviously hold at the beginning.

Invariant (1): F is a maximum spanning forest of G if edge levels are interpreted as weights.

Invariant (2): The number of nodes in each tree of F_i is at most $n/2^i$.

Invariant (1) should be interpreted as follows. Let (u, v) be a non-tree edge of level $\ell(u, v)$ and let $u \cdots v$ be the unique path between u and v in F (such a path exists since F is a spanning forest of G). Let e be any edge in $u \cdots v$ and let $\ell(e)$ be its level. Due to (1), $\ell(e) \geq \ell(u, v)$. Since this holds for each edge in the path, and by construction $F_{\ell(u,v)}$ contains all the tree edges of level $\geq \ell(u, v)$, the entire path is contained in $F_{\ell(u,v)}$, i.e., u and v are connected in $F_{\ell(u,v)}$.

Invariant (2) implies that the maximum number of levels is $L \leq \lfloor \log_2 n \rfloor$.

Note that when a new edge is inserted, it is given level 0. Its level can be then increased at most $\lfloor \log_2 n \rfloor$ times as a consequence of edge deletions. When a tree edge $e = (v, w)$ of level $\ell(e)$ is deleted, the algorithm looks for a replacement edge at the highest possible level, if any. Due to invariant (1), such a replacement edge has level $\ell \leq \ell(e)$. Hence, a replacement subroutine `Replace(`(u, w)`,`$\ell(e)$`)` is called with parameters e and $\ell(e)$. We now sketch the operations performed by this subroutine.

> `Replace(`(u, w)`,`ℓ`)` finds a replacement edge of the highest level $\leq \ell$, if any. If such a replacement does not exist in level ℓ, we have two cases: if $\ell > 0$, we recurse on level $\ell - 1$; otherwise, $\ell = 0$, and we can conclude that the deletion of (v, w) disconnects v and w in G.

During the search at level ℓ, suitably chosen tree and non-tree edges may be promoted at higher levels as follows. Let T_v and T_w be the trees of forest F_ℓ obtained after deleting (v, w) and let, w.l.o.g., T_v be smaller than T_w. Then T_v contains at most $n/2^{\ell+1}$ vertices, since $T_v \cup T_w \cup \{(v, w)\}$ was a tree at level ℓ and due to invariant (2). Thus, edges in T_v of level ℓ can be promoted at level $\ell+1$ by maintaining the invariants. Non-tree edges incident to T_v are finally visited one by one: if an edge does connect T_v and T_w, a replacement edge has been found and the search stops, otherwise its level is increased by 1.

We maintain an ET-tree, as described in Section 35.5, for each tree of each forest. Consequently, all the basic operations needed to implement edge insertions and deletions can be supported in $O(\log n)$ time. In addition to inserting and deleting edges from a forest, ET-trees must also support operations such as finding the tree of a forest that contains a given vertex, computing the size of a tree, and, more importantly, finding tree edges of level ℓ in T_v and non-tree edges of level ℓ incident to T_v. This can be done by augmenting the ET-trees with a constant amount of information per node: we refer the interested reader to [22] for details.

Using an amortization argument based on level changes, the claimed $O(\log^2 n)$ bound on the update time can be finally proved. Namely, inserting an edge costs $O(\log n)$, as well as increasing its level. Since this can happen $O(\log n)$ times, the total amortized insertion cost, inclusive of level increases, is $O(\log^2 n)$. With respect to edge deletions, cutting and linking $O(\log n)$ forest has a total cost $O(\log^2 n)$; moreover, there are $O(\log n)$ recursive calls to `Replace`, each of cost $O(\log n)$ plus the cost amortized over level increases. The ET-trees over $F_0 = F$ allows it to answer connectivity queries in $O(\log n)$ worst-case time. As shown in [22], this can be reduced to $O(\log n / \log \log n)$ by using a $\Theta(\log n)$-ary version of ET-trees.

THEOREM 36.9 *(Holm et al. [22]) A dynamic graph G with n vertices can be maintained upon insertions and deletions of edges using $O(\log^2 n)$ amortized time per update and answering connectivity queries in $O(\log n / \log \log n)$ worst-case running time.*

36.5 Minimum Spanning Tree

One of the most studied dynamic problem on undirected graphs is the *fully dynamic minimum spanning tree* problem, which consists of maintaining a minimum spanning forest of a graph during insertions of edges, deletions of edges, and edge cost changes. In this section, we show that a few simple changes to the connectivity algorithm presented in Section 36.4 are sufficient to maintain a minimum spanning forest of a weighted undirected graph upon deletions of edges [22]. A general reduction from [19] can then be applied to make the deletions-only algorithm fully dynamic.

36.5.1 Deletions in $O(\log^2 n)$ Time

In addition to starting from a *minimum* spanning forest, the only change concerns function `Replace`, that should be implemented so as to consider candidate replacement edges of level ℓ in order of increasing weight, and not in arbitrary order. To do so, the ET-trees described in Section 35.5 can be augmented so that each node maintains the minimum weight of a non-tree edge incident to the Euler tour segment below it. All the operations can still be supported in $O(\log n)$ time, yielding the same time bounds as for connectivity.

We now discuss the correctness of the algorithm. In particular, function `Replace` returns a replacement edge of minimum weight on the highest possible level: it is not immediate that such a replacement edge has the minimum weight among all levels. This can be proved by first showing that the following invariant, proved in [22], is maintained by the algorithm:

Invariant (3): Every cycle $\mathcal{C}$ has a non-tree edge of maximum weight and minimum level among all the edges in $\mathcal{C}$.

Invariant (3) can be used to prove that, among all the replacement edges, the lightest edge is on the maximum level. Let e_1 and e_2 be two replacement edges with $w(e_1) < w(e_2)$, and let $\mathcal{C}_i$ be the cycle induced by e_i in F, $i = 1, 2$. Since F is a minimum spanning forest, e_i has maximum weight among all the edges in $\mathcal{C}_i$. In particular, since by hypothesis $w(e_1) < w(e_2)$, e_2 is also the heaviest edge in cycle $\mathcal{C} = (\mathcal{C}_1 \cup \mathcal{C}_2) \setminus (\mathcal{C}_1 \cap \mathcal{C}_2)$. Thanks to Invariant (3), e_2 has minimum level in $\mathcal{C}$, proving that $\ell(e_2) \leq \ell(e_1)$. Thus, considering non-tree edges from higher to lower levels is correct.

LEMMA 36.5 (Holm et al. [22]) There exists a deletions-only minimum spanning forest algorithm that can be initialized on a graph with n vertices and m edges and supports any sequence of edge deletions in $O(m \log^2 n)$ total time.

36.5.2 Updates in $O(\log^4 n)$ Time

The reduction used to obtain a fully dynamic algorithm, which involves the top tree data structure discussed in Section 35.4, is a slight generalization of the construction proposed by Henzinger and King [19], and works as follows.

LEMMA 36.6 [19, 22] Suppose we have a deletions-only minimum spanning tree algorithm that, for any k and l, can be initialized on a graph with k vertices and l edges and supports any sequence of $\Omega(l)$ deletions in total time $O(l \cdot t(k, l))$, where t is a non-decreasing function. Then there exists a fully-dynamic minimum spanning tree algorithm for a graph with n nodes starting with no edges, that, for m edges, supports updates in time

$$O\left(\log^3 n + \sum_{i=1}^{3+\log_2 m} \sum_{j=1}^{i} t(min\{n, 2^j\}, 2^j)\right)$$

We refer the interested reader to references [19] and [22] for the description of the construction that proves Lemma 36.6. From Lemma 36.5 we get $t(k, l) = O(\log^2 k)$. Hence, combining Lemmas 36.5 and 36.6, we get the claimed result.

THEOREM 36.10 *(Holm et al. [22]) There exists a fully-dynamic minimum spanning* *forest algorithm that, for a graph with n vertices, starting with no edges, maintains a minimum spanning forest in $O(\log^4 n)$ amortized time per edge insertion or deletion.*

36.6 Transitive Closure

In the rest of this chapter we survey the newest results for dynamic problems on directed graphs. In particular, we focus on two of the most fundamental problems: transitive closure and shortest paths. These problems play a crucial role in many applications, including network optimization and routing, traffic information systems, databases, compilers, garbage collection, interactive verification systems, industrial robotics, dataflow analysis, and document formatting. In this section we consider the best known algorithms for fully dynamic transitive closure. Given a directed graph G with n vertices and m edges, the problem consists of supporting any intermixed sequence of operations of the following kind:

`Insert`(u,v): insert edge (u,v) in G;

`Delete`(u,v): delete edge (u,v) from G;

`Query`(x,y): answer a reachability query by returning "yes" if there is a path from vertex x to vertex y in G, and "no" otherwise;

A simple-minded solution to this problem consists of maintaining the graph under insertions and deletions, searching if y is reachable from x at any query operation. This yields $O(1)$ time per update (`Insert` and `Delete`), and $O(m)$ time per query, where m is the current number of edges in the maintained graph.

Another simple-minded solution would be to maintain the Kleene closure of the adjacency matrix of the graph, rebuilding it from scratch after each update operation. Using the recursive decomposition of Munro [28] discussed in Section 36.3.1 and fast matrix multiplication [4], this takes constant time per reachability query and $O(n^\omega)$ time per update, where $\omega < 2.38$ is the current best exponent for matrix multiplication.

Despite many years of research in this topic, no better solution to this problem was known until 1995, when Henzinger and King [20] proposed a randomized Monte Carlo algorithm with one-sided error supporting a query time of $O(n/\log n)$ and an amortized update time of $O(n\widehat{m}^{0.58}\log^2 n)$, where $\widehat{m}$ is the average number of edges in the graph throughout the whole update sequence. Since $\widehat{m}$ can be as high as $O(n^2)$, their update time is $O(n^{2.16}\log^2 n)$. Khanna, Motwani and Wilson [23] proved that, when a lookahead of $\Theta(n^{0.18})$ in the updates is permitted, a deterministic update bound of $O(n^{2.18})$ can be achieved.

King and Sagert [25] showed how to support queries in $O(1)$ time and updates in $O(n^{2.26})$ time for general directed graphs and $O(n^2)$ time for directed acyclic graphs; their algorithm is randomized with one-sided error. The bounds of King and Sagert were further improved by King [24], who exhibited a deterministic algorithm on general digraphs with $O(1)$ query time and $O(n^2\log n)$ amortized time per update operations, where updates are insertions of a set of edges incident to the same vertex and deletions of an arbitrary subset of edges. Using a different framework, in 2000 Demetrescu and Italiano [6] obtained a deterministic fully dynamic algorithm that achieves $O(n^2)$ amortized time per update for general directed graphs. We note that each update might change a portion of the transitive closure as large as $\Omega(n^2)$. Thus, if the transitive closure has to be maintained explicitly after each update so that queries can be answered with one lookup, $O(n^2)$ is the best update bound one could hope for.

If one is willing to pay more for queries, Demetrescu and Italiano [6] showed how to break the $O(n^2)$ barrier on the single-operation complexity of fully dynamic transitive closure: building on a previous path counting technique introduced by King and Sagert [25], they devised a randomized algorithm with one-sided error for directed acyclic graphs that achieves $O(n^{1.58})$ worst-case time per update and $O(n^{0.58})$ worst-case time per query. Other recent results for dynamic transitive closure appear in [34, 35].

36.6.1 Updates in $O(n^2 \log n)$ Time

In this section we address the algorithm by King [24], who devised the first deterministic near-quadratic update algorithm for fully dynamic transitive closure. The algorithm is based on the reachability tree data structure considered in Section 35.6 and on the logarithmic decomposition discussed in Section 36.3.1.

It maintains explicitly the transitive closure of a graph G in $O(n^2 \log n)$ amortized time per update, and supports inserting and deleting several edges of the graph with just one operation. Insertion of a bunch of edges incident to a vertex and deletion of any subset of edges in the graph require asymptotically the same time of inserting/deleting just one edge.

The algorithm maintains $\log n + 1$ levels: level i, $0 \leq i \leq \log n$, maintains a graph G_i whose edges represent paths of length up to 2^i in the original graph G. Thus, $G_0 = G$ and $G_{\log n}$ is the transitive closure of G.

Each level graph G_i is built on top of the previous level graph G_{i-1} by keeping two trees of depth ≤ 2 rooted at each vertex v: an out-tree $OUT_i(v)$ maintaining vertices reachable from v by traversing at most two edges in G_{i-1}, and an in-tree $IN_i(v)$ maintaining vertices that reach v by traversing at most two edges in G_{i-1}. Trees $IN_i(v)$ can be constructed by considering the orientation of edges in G_{i-1} reversed. An edge (x, y) will be in G_i if and only if $x \in IN_i(v)$ and $y \in OUT_i(v)$ for some v. In/out trees are maintained with the deletions-only reachability tree data structure considered in Section 35.6.

To update the levels after an insertion of edges around a vertex v in G, the algorithm simply rebuilds $IN_i(v)$ and $OUT_i(v)$ for each i, $1 \leq i \leq \log n$, while other trees are not touched. This means that some trees might not be up to date after an insertion operation. Nevertheless, any path in G is represented in at least the in/out trees rooted at the latest updated vertex in the path, so the reachability information is correctly maintained. This idea is the key ingredient of King's algorithm.

When an edge is deleted from G_i, it is also deleted from any data structures $IN_i(v)$ and $OUT_i(v)$ that contain it. The interested reader can find further details in [24].

36.6.2 Updates in $O(n^2)$ Time

In this section we address the algorithm by Demetrescu and Italiano [6]. The algorithm is based on the matrix data structure considered in Section 36.3.4 and on the recursive decomposition discussed in Section 36.3.1.

It maintains explicitly the transitive closure of a graph in $O(n^2)$ amortized time per update, supporting the same generalized update operations of King's algorithm, i.e., insertion of a bunch of edges incident to a vertex and deletion of any subset of edges in the graph with just one operation. This is the best known update bound for fully dynamic transitive closure with constant query time.

The algorithm maintains the Kleene closure X^* of the $n \times n$ adjacency matrix X of the graph as the sum of two matrices X_1 and X_2. Let V_1 be the subset of vertices of the graph corresponding to the first half of indices of X, and let V_2 contain the remaining vertices. Both matrices X_1 and X_2 are defined according to Munro's equations of Section 36.3.1, but in such a way that paths appearing due to an insertion of edges around a vertex in V_1 are correctly recorded in X_1, while paths that appear due to an insertion of edges around a vertex in V_2 are correctly recorded in X_2. Thus, neither X_1 nor X_2 encode complete information about X^*, but their sum does. In more detail, assuming that X is decomposed in sub-matrices A, B, C, D as explained in Section 36.3.1, and that X_1, and X_2 are similarly decomposed in sub-matrices E_1, F_1, G_1, H_1 and E_2, F_2, G_2, H_2, the algorithm maintains X_1 and X_2 with the following 8 polynomials using the data structure discussed in Section 36.3.4:

$Q = A + BP^2C$	$E_2 = E_1BH_2^2CE_1$
$F_1 = E_1^2BP$	$F_2 = E_1BH_2^2$
$G_1 = PCE_1^2$	$G_2 = H_2^2CE_1$
$H_1 = PCE_1^2BP$	$R = D + CE_1^2B$

where $P = D^*$, $E_1 = Q^*$, and $H_2 = R^*$ are Kleene closures maintained recursively as smaller instances of the problem of size $n/2 \times n/2$.

To support an insertion of edges around a vertex in V_1, strict updates are performed on polynomials Q, F_1, G_1, and H_1 using `SetRow` and `SetCol`, while E_2, F_2, G_2, and R are updated with `LazySet`. Insertions around V_2 are performed symmetrically, while deletions are supported via `Reset` operations on each polynomial in the recursive decomposition. Finally, P, E_1, and H_2 are updated recursively. The interested reader can find the low-level details of the method in [6].

36.7 All-Pairs Shortest Paths

In this section we survey the best known algorithms for fully dynamic all pairs shortest paths (in short APSP). Given a weighted directed graph G with n vertices and m edges, the problem consists of supporting any intermixed sequence of operations of the following kind:

`Update`(u, v, w): updates the weight of edge (u, v) in G to the new value w (if $w = +\infty$ this corresponds to edge deletion);

`Query`(x, y): returns the distance from vertex x to vertex y in G, or $+\infty$ if no path between them exists;

The dynamic maintenance of shortest paths has a remarkably long history, as the first papers date back to 35 years ago [26, 29, 33]. After that, many dynamic shortest paths algorithms have been proposed (see, e.g., [11, 15, 16, 30, 31, 36]), but their running times in the worst case were comparable to recomputing APSP from scratch.

The first dynamic shortest path algorithms which are provably faster than recomputing APSP from scratch, only worked on graphs with small integer weights. In particular, Ausiello *et al.* [1] proposed a decrease-only shortest path algorithm for directed graphs having positive integer weights less than C: the amortized running time of their algorithm is $O(Cn \log n)$ per edge insertion. Henzinger *et al.* [21] designed a fully dynamic algorithm for APSP on planar graphs with integer weights, with a running time of $O(n^{4/3} \log(nC))$ per operation. Fakcharoemphol and Rao in [12] designed a fully dynamic algorithm for single-source shortest paths in planar directed graphs that supports both queries and edge weight updates in $O(n^{4/5} \log^{13/5} n)$ amortized time per edge operation.

The first big step on general graphs and integer weights was made by King [24], who presented a fully dynamic algorithm for maintaining all pairs shortest paths in directed graphs with positive integer weights less than C: the running time of her algorithm is $O(n^{2.5}\sqrt{C \log n}\,)$ per update.

Demetrescu and Italiano [7] gave the first algorithm for fully dynamic APSP on general directed graphs with real weights assuming that each edge weight can attain a limited number S of different *real* values throughout the sequence of updates. In particular, the algorithm supports each update in $O(n^{2.5}\sqrt{S\log^3 n}\,)$ amortized time and each query in $O(1)$ worst-case time. The same authors discovered the first algorithm that solves the fully dynamic all pairs shortest paths problem in its generality [8]. The algorithm maintains explicitly information about shortest paths supporting any edge weight update in $O(n^2\log^3 n)$ amortized time per operation in directed graphs with non-negative real edge weights. Distance queries are answered with one lookup and actual shortest paths can be reconstructed in optimal time. We note that each update might change a portion of the distance matrix as large as $\Omega(n^2)$. Thus, if the distance matrix has to be maintained explicitly after each update so that queries can be answered with one lookup, $O(n^2)$ is the best update bound one could hope for. Other deletions-only algorithms for APSP, in the simpler case of unweighted graphs, are presented in [2].

36.7.1 Updates in $O(n^{2.5}\sqrt{C\log n}\,)$ Time

In this section we consider the dynamic shortest paths algorithm by King [24]. The algorithm is based on the long paths property discussed in Section 36.3.2 and on the reachability tree data structure Section 35.6.

Similarly to the transitive closure algorithms described in Section 36.6 generalized update operations are supported within the same bounds, i.e., insertion (or weight decrease) of a bunch of edges incident to a vertex, and deletion (or weight increase) of any subset of edges in the graph with just one operation.

The main idea of the algorithm is to maintain dynamically all pairs shortest paths up to a distance d, and to recompute longer shortest paths from scratch at each update by stitching together shortest paths of length $\leq d$. For the sake of simplicity, we only consider the case of unweighted graphs: an extension to deal with positive integer weights less than C is described in [24].

To maintain shortest paths up to distance d, similarly to the transitive closure algorithm by King described in Section 36.6, the algorithm keeps a pair of in/out shortest paths trees $IN(v)$ and $OUT(v)$ of depth $\leq d$ rooted at each vertex v. Trees $IN(v)$ and $OUT(v)$ are maintained with the decremental data structure mentioned in Chapter 35. It is easy to prove that, if the distance d_{xy} between any pair of vertices x and y is at most d, then d_{xy} is equal to the minimum of $d_{xv}+d_{vy}$ over all vertices v such that $x \in IN(v)$ and $y \in OUT(v)$. To support updates, insertions of edges around a vertex v are handled by rebuilding only $IN(v)$ and $OUT(v)$, while edge deletions are performed via operations on any trees that contain them. The amortized cost of such updates is $O(n^2 d)$ per operation.

To maintain shortest paths longer than d, the algorithm exploits the long paths property of Theorem 36.5: in particular, it hinges on the observation that, if H is a random subset of $\Theta((n\log n)/d)$ vertices in the graph, then the probability of finding more than d consecutive vertices in a path, none of which are from H, is very small. Thus, if we look at vertices in H as "hubs", then any shortest path from x to y of length $\geq d$ can be obtained by stitching together shortest subpaths of length $\leq d$ that first go from x to a vertex in H, then jump between vertices in H, and eventually reach y from a vertex in H. This can be done by first computing shortest paths only between vertices in H using any cubic-time static all-pairs shortest paths algorithm, and then by extending them at both endpoints with shortest paths of length $\leq d$ to reach all other vertices. This stitching operation requires $O(n^2|H|) = O((n^3\log n)/d)$ time.

Choosing $d = \sqrt{n\log n}$ yields an $O(n^{2.5}\sqrt{\log n})$ amortized update time. As mentioned in Section 36.3.2, since H can be computed deterministically, the algorithm can be derandomized. The interested reader can find further details on the algorithm in [24].

36.7.2 Updates in $O(n^2 \log^3 n)$ Time

In this section we address the algorithm by Demetrescu and Italiano [8], who devised the first deterministic near-quadratic update algorithm for fully dynamic all-pairs shortest paths. This algorithm is also the first solution to the problem in its generality. The algorithm is based on the notions of historical paths and locally historical paths in a graph subject to a sequence of updates, as discussed in Section 36.3.3.

The main idea is to maintain dynamically the locally historical paths of the graph in a data structure. Since by Lemma 36.3 shortest paths are locally historical, this guarantees that information about shortest paths is maintained as well.

To support an edge weight update operation, the algorithm implements the smoothing strategy mentioned in Section 36.3.3 and works in two phases. It first removes from the data structure all maintained paths that contain the updated edge: this is correct since historical shortest paths, in view of their definition, are immediately invalidated as soon as they are touched by an update. This means that also locally historical paths that contain them are invalidated and have to be removed from the data structure. As a second phase, the algorithm runs an all-pairs modification of Dijkstra's algorithm [9], where at each step a shortest path with minimum weight is extracted from a priority queue and it is combined with existing historical shortest paths to form new locally historical paths. At the end of this phase, paths that become locally historical after the update are correctly inserted in the data structure.

The update algorithm spends constant time for each of the $O(zn^2)$ new locally historical path (see Theorem 36.7). Since the smoothing strategy lets $z = O(\log n)$ and increases the length of the sequence of updates by an additional $O(\log n)$ factor, this yields $O(n^2 \log^3 n)$ amortized time per update. The interested reader can find further details about the algorithm in [8].

36.8 Conclusions

In this chapter we have surveyed the algorithmic techniques underlying the fastest known dynamic graph algorithms for several problems, both on undirected and on directed graphs. Most of the algorithms that we have presented achieve bounds that are close to the best possible. In particular, we have presented fully dynamic algorithms with polylogarithmic amortized time bounds for connectivity and minimum spanning trees [22] on undirected graphs. It remains an interesting open problem to show whether polylogarithmic update bounds can be achieved also in the worst case: we recall that for both problems the current best worst-case bound is $O(\sqrt{n})$ per update, and it is obtained with the sparsification technique [10] described in Section 36.2.

For directed graphs, we have shown how to achieve constant-time query bounds and nearly-quadratic update bounds for transitive closure and all pairs shortest paths. These bounds are close to optimal in the sense that one update can make as many as $\Omega(n^2)$ changes to the transitive closure and to the all pairs shortest paths matrices. If one is willing to pay more for queries, Demetrescu and Italiano [6] have shown how to break the $O(n^2)$ barrier on the single-operation complexity of fully dynamic transitive closure for directed acyclic graphs. This also yields the first efficient update algorithm that maintains reachability in acyclic directed graphs between two fixed vertices s and t, or from s to all other vertices. However, in the case of general graphs or shortest paths, no solution better that the static is known for these problems. In general, it remains an interesting open problem to show whether effective query/update tradeoffs can be achieved for general graphs and for shortest paths problems.

Finally, dynamic algorithms for other fundamental problems such as matching and flow problems deserve further investigation.

Acknowledgment

This work has been supported in part by the IST Programmes of the EU under contract numbers IST-1999-14186 (ALCOM-FT) and IST-2001-33555 (COSIN), and by the Italian Ministry of University and Scientific Research (Project "ALINWEB: Algorithmics for Internet and the Web").

References

[1] G. Ausiello, G. F. Italiano, A. Marchetti-Spaccamela, and U. Nanni. Incremental algorithms for minimal length paths. *Journal of Algorithms*, 12(4):615–38, 1991.

[2] S. Baswana, R. Hariharan, and S. Sen. Improved decremental algorithms for transitive closure and all-pairs shortest paths. In *Proc. 34th ACM Symposium on Theory of Computing (STOC'02)*, pages 117–123, 2002.

[3] V. Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[4] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9, 1990.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, Second Edition. The MIT Press, 2001.

[6] C. Demetrescu and G. F. Italiano. Fully dynamic transitive closure: Breaking through the $O(n^2)$ barrier. In *Proc. of the 41st IEEE Annual Symposium on Foundations of Computer Science (FOCS'00)*, pages 381–389, 2000.

[7] C. Demetrescu and G. F. Italiano. Fully dynamic all pairs shortest paths with real edge weights. In *Proc. of the 42nd IEEE Annual Symposium on Foundations of Computer Science (FOCS'01), Las Vegas, Nevada*, pages 260–267, 2001.

[8] C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. In *Proc. 35th Symp. on Theory of Computing* (STOC'03), 2003.

[9] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[10] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification – A technique for speeding up dynamic graph algorithms. *J. Assoc. Comput. Mach.*, 44:669–696, 1997.

[11] S. Even and H. Gazit. Updating distances in dynamic graphs. *Methods of Operations Research*, 49:371–387, 1985.

[12] J. Fakcharoemphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. In *Proc. of the 42nd IEEE Annual Symposium on Foundations of Computer Science (FOCS'01), Las Vegas, Nevada*, pages 232–241, 2001.

[13] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees. *SIAM J. Comput.*, 14:781–798, 1985.

[14] G. N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM J. Comput.*, 26(2): 484–538, 1997.

[15] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Semi-dynamic algorithms for maintaining single source shortest paths trees. *Algorithmica*, 22(3):250–274, 1998.

[16] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, 34:251–281, 2000.

[17] Z. Galil and G. F. Italiano. Fully-dynamic algorithms for 2-edge connectivity. *SIAM J. Comput.*, 21:1047–1069, 1992.

[18] D. H. Greene and D. E. Knuth. *Mathematics for the analysis of algorithms.* Birkhäuser, 1982.

[19] M. R. Henzinger and V. King. Maintaining minimum spanning trees in dynamic graphs. In *Proc. 24th Int. Colloquium on Automata, Languages and Programming (ICALP 97)*, pages 594–604, 1997.

[20] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. Assoc. Comput. Mach.*, 46(4):502–536, 1999.

[21] M. R. Henzinger, P. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3–23, August 1997.

[22] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. Assoc. Comput. Mach.*, 48(4):723–760, 2001.

[23] S. Khanna, R. Motwani, and R. H. Wilson. On certificates and lookahead on dynamic graph problems. In *Algorithmica* 21(4): 377-394 (1998).

[24] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proc. 40-th Symposium on Foundations of Computer Science (FOCS 99)*, 1999.

[25] V. King and G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. In *Proc. 31-st ACM Symposium on Theory of Computing (STOC 99)*, pages 492–498, 1999.

[26] P. Loubal. A network evaluation procedure. *Highway Research Record 205*, pages 96–109, 1967.

[27] L. Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13:383–390, 1975.

[28] I. Munro. Efficient determination of the transitive closure of a directed graph. *Information Processing Letters*, 1(2):56–58, 1971.

[29] J. Murchland. The effect of increasing or decreasing the length of a single arc on all shortest distances in a graph. Technical report, LBS-TNT-26, London Business School, Transport Network Theory Unit, London, UK, 1967.

[30] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest path problem. *Journal of Algorithms*, 21:267–305, 1996.

[31] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158:233–277, 1996.

[32] M. Rauch. Fully dynamic biconnectivity in graphs. In *Algorithmica*, 13:503-538, 1995.

[33] V. Rodionov. The parametric problem of shortest distances. *U.S.S.R. Computational Math. and Math. Phys.*, 8(5):336–343, 1968.

[34] L. Roditty. A faster and simpler fully dynamic transitive closure. In *Proceedings of the 14th ACM-SIAM Symposium on Discrete Algorithms (SODA'03)*, pages 404–412, Baltimore, Maryland, USA, January 12-14, 2003.

[35] L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. In *Proceedings of the 43th Annual IEEE Symposium on Foundations of Computer Science (FOCS'02)*, p. 679, Vancouver, Canada, 2002.

[36] H. Rohnert. A dynamization of the all-pairs least cost problem. In *Proc. 2nd Annual Symposium on Theoretical Aspects of Computer Science, (STACS 85), LNCS 182*, pages 279–286, 1985.

[37] J. D. Ullman and M. Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM Journal on Computing*, 20(1):100–125, 1991.

[38] U. Zwick. All pairs shortest paths in weighted directed graphs - exact and almost exact algorithms. In *Proc. of the 39th IEEE Annual Symposium on Foundations of Computer Science (FOCS'98)*, pages 310–319, Los Alamitos, CA, November 8–11 1998.

37

Succinct Representation of Data Structures

J. Ian Munro
University of Waterloo

S. Srinivasa Rao
University of Waterloo

37.1 Introduction

Although computer memories, at all levels of the hierarchy, have grown dramatically over the past few years, increased problem sizes continues to outstrip this growth. Minimizing space is crucial not only in keeping data in the fastest memory possible, but also in moving it from one level to another, be it from main memory to cache or from a web site around the world. Standard data compression, say Huffman code or grammar based code, applied to a large text file reduces space dramatically, but basic operations on the text require that it be fully decoded.

In this chapter we focus on representations that are not only terse but also permit the basic operations one would expect on the underlying data type to be performed quickly. Jacobson [33] seems to have been the first to apply the term *succinct* to such structures; the goal is to use the information-theoretic minimum number of bits and to support the expected operations on the data type in optimal time. Our archetypical example (discussed in Section 37.4) is the representation of a binary tree. Suppose, we would like to support the operations of navigating through a binary tree moving to either child or the parent of the current node, asking the size of the subtree rooted at the current node or giving the unique 'number' of the node so that data can be stored in that position of an array. At $\lg n$ bits per reference, this adds up to at least $5n \lg n$ bits. However, there are only

$\binom{2n+1}{n}/(2n+1)$ binary trees, so the information-theoretic minimum space is fewer than $2n$ bits. Our archetypical data structure is a $2n + o(n)$-bit representation that supports the operations noted above, and others, in constant time.

We consider a variety of abstract data types, or combinatorial objects, with the goal of producing such succinct data structures. Most, though not all, of the structures we consider are static. In most cases the construction of a succinct data structure from the standard representation is fairly straightforward in linear time.

Memory Model: We study the problems under the RAM model with word size $\Theta(\lg n)$, where n is the input size of the problem under consideration. This supports arithmetic (addition, subtraction, multiplication and division), indexing and bit-wise boolean operations (AND, OR, NOT, XOR etc.) on words, and reading/writing of words from/to the memory in constant time.

37.2 Bitvector

A bitvector provides a simple way to represent a set from any universe that is easily mapped onto $[m]$ *. Membership queries (checking whether a given element from the universe is present in the set) can be answered in constant time (in fact a single bit probe) using a bitvector. Furthermore, one can easily support updates (inserting and deleting elements) in constant time. The most interesting twist on the bitvector came with Jacobson [33] considering two more operations:

- rank(i) : return the number of 1s before the position i, and
- select(i) : return the position of the i-th 1.

As we shall see, these operations are crucial to a number of more complex structures supporting a variety of data types. An immediate use is to support the queries:

- predecessor(x) : find the largest element $y \leq x$ in the set S,
- successor(x) : find the smallest element $y \geq x$ in the set S.

Given a bitvector of length m, Jacobson [33] gave a structure that takes $o(m)$ bits of additional space and supports rank and select operations by making $O(\lg m)$ bit probes to the structure. On a RAM with word size $\Theta(\lg m)$ bits, the structure given by Munro [40] enhanced this structure and the algorithms to support the operations in $O(1)$ time, without increasing the space bound. We briefly describe the details of this structure.

The structure for computing rank, the rank directory, consists of the following:

- Conceptually break the bitvector into blocks of length $\lceil \lg^2 m \rceil$. Keep a table containing the number of **1**s up to the last position in each block. This takes $O(m/\lg m)$ bits of space.
- Conceptually break each block into sub-blocks of length $\lceil \frac{1}{2} \lg m \rceil$. Keep a table containing the number of **1**s within the block up to the last position in each sub-block. This takes $O(m \lg \lg m/\lg m)$ bits.
- Keep a precomputed table giving the number of **1**s up to every possible position in every possible distinct sub-block. Since there $O(\sqrt{m})$ distinct possible blocks, and $O(\lg m)$ positions in each, this takes $O(\sqrt{m} \lg m \lg \lg m)$ bits.

*for positive integers m, $[m]$ denotes the set $\{0, 1, \ldots, m-1\}$

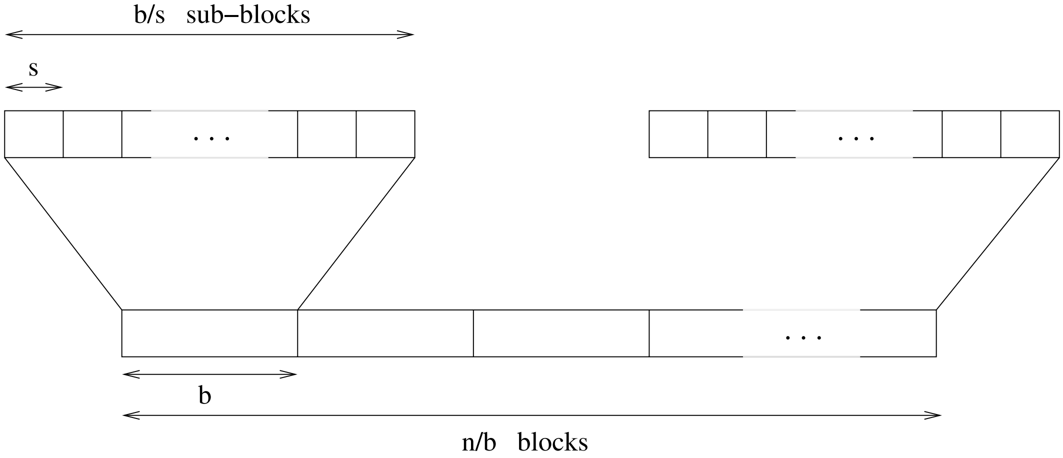


FIGURE 37.1: Two-level rank directory.

Thus, the total space occupied by this auxiliary structure is $o(m)$ bits. The rank of an element is, then, simply the sum of three values, one from each table.

The structure for computing select uses three levels of directories and is more complex. The first one records the position of every $(\lg m \lg\lg m)$-th 1 bit in the bitvector. This takes $O(m/\lg\lg m)$ bits. Let r be the subrange between two values in the first directory, and consider the sub-directory for this range. If $r \geq (\lg m \lg\lg m)^2$ then explicitly store the positions of all ones, which requires $O(r/\lg\lg m)$ bits. Otherwise, subdivide the range and store the position (relative to the beginning of this range) of every $(\lg r \lg\lg m)$-th one bit in the second level directory. This takes $O(r/\lg\lg m)$ bits for each range of size r, and hence $O(m/\lg\lg m)$ bits over the entire bitvector. After one more level of similar range subdivision, the range size will reduce to at most $(\lg\lg m)^4$. Computing select on these small ranges is performed using a precomputed table. The total space occupied by this auxiliary structure is $o(m)$ bits. The query algorithm is straightforward. See [9, 44] for details.

This 'indexable bitvector' is used as a substructure in several succinct data structures. To represent a bitvector of length m, it takes $m + o(m)$ bits of space. In general, if nothing is known about the bitvector then any representation needs at least m bits to distinguish between all possible bitvectors, and hence this is close to the optimal space. But if we also know the density (the number of ones) of the bitvector, then the space bound is no longer optimal, in general. The 'fully indexable dictionary' described in Section 37.3.2 gives a solution that takes nearly optimal space.

Using the ideas involved in constructing rank and select directories, one can also support the following generalizations of these two operations, using $o(m)$ bits of extra space: Given a bitvector of length m, and a fixed binary pattern p of length up to $(1-\epsilon)\lg m$, for some fixed constant $0 < \epsilon < 1$

- $\mathsf{rank}_p(i)$: return the number of (possibly overlapping) occurrences of p before the position i, and
- $\mathsf{select}_p(i)$: return the i-th occurrence of the pattern p.

One can extend the ideas of rank and select directories to support indexing into a fixed or variable length encoded text (e.g. Huffman coding, prefix-free encoding etc.) in constant time, using negligible extra space. See [33, 44] for some examples.

37.3 Succinct Dictionaries

The (static) dictionary problem is to store a subset S of size n so that membership queries can be answered efficiently. In our case, the universe is taken to be the set $[m]$. This problem has been widely studied and various solutions have been proposed to support membership queries in constant time.

As we have seen in the last section, a bitvector is a simple way of representing a set from a given universe. But this requires m bits of space. Since there are $\binom{m}{n}$ sets of size n from a universe of size m, one would require only $\mathcal{B} \equiv \lg \binom{m}{n}$ ($\approx n(\lg m - \lg n + \lg e)$, when $n = o(m)$) bits to store a canonical representation of any such set. Thus a bitvector is quite wasteful of space when the set is sparse. A sorted array is another simple representation, but it requires $\Theta(\lg n)$ time to answer queries. A *fusion tree* (see Chapter 39) also takes linear space and supports membership queries in $\Theta(\lg n / \lg\lg n)$ time. In this section, we consider representations of sets whose space complexity is close to the information theoretic minimum and support queries in constant time. (As all the structures outlined below support membership queries in worst case constant time, we do not mention the query complexity explicitly.)

Fredman, Komlós and Szemerédi [19] gave the first linear space structure for the static dictionary problem. This takes $n \lg m + O(n\sqrt{\lg n} + \lg\lg m)$ bits of space. The lower order term was later improved by Schmidt and Siegel [55] to $O(n + \lg\lg m)$. This structure uses a universe reduction function followed by a two-level hash function to hash the given subset one-to-one onto the set $[n]$, and stores the elements of subset in a hash table (in the order determined by the hash function). The hash table takes $n \lg m$ bits and a clever encoding of the hash function takes $O(n + \lg\lg m)$ bits of space. We refer to this as the FKS hashing scheme. Note that the space required for this structure is $\Theta(n \lg n)$ bits more than the optimal bound of $\mathcal{B}$ bits.

Brodnik and Munro [5] gave a static dictionary representation that takes $\mathcal{B} + o(\mathcal{B})$ bits of space. It uses two different solutions depending on the relative values of n and m. When the set is relatively sparse (namely, when $n \leq m/(\lg m)^{\lg\lg m}$), it partitions the elements into buckets based on the first $\lg n - \lg\lg m$ bits of their bit representations, and store explicit pointers to refer to the representations of individual buckets. Each bucket is represented by storing all the elements that fall into it in a perfect hash table for that bucket. Otherwise, when the set is dense, it uses two levels of bucketing (at each level splitting the universe into a number of equal-range buckets, depending only on the universe size) after which the range of these buckets reduces to $\Theta(\lg n)$. These small buckets are stored (almost) optimally by storing pointers into a precomputed table that contains all possible small buckets. In either case the space occupancy can be shown to be $\mathcal{B} + o(\mathcal{B})$ bits.

Pagh [46] observed that each bucket of the hash table may be resolved with respect to the part of the universe hashing to that bucket. Thus, one can save space by compressing the hash table, storing only the *quotient* information, rather than the element itself. From the FKS hash function, one can obtain a quotienting function that takes $\lg(m/n) + O(1)$ bits for each element. Using this idea one can obtain a dictionary structure that takes $n \lg(m/n) + O(n + \lg\lg m)$ bits of space, which is only $\Theta(n)$ bits more than the information-theoretic lower bound (except for the $O(\lg\lg m)$ term). Pagh has also given a dictionary structure that takes only $\mathcal{B} + o(n) + O(\lg\lg m)$ bits of space.

37.3.1 Indexable Dictionary

One useful feature of the sorted array representation of a set is that, given an index i, the i-th smallest element in the set can be retrieved in constant time. Furthermore, when we locate an element in the array, we immediately know its rank (the number of elements in the set which are less than the given element). On the other hand, hashing based schemes support membership in constant time, but typically do not maintain the ordering information. In this section we look at a structure that combines the good features of both these approaches.

An *indexable dictionary* is a structure representing a set S of size n from the universe $[m]$ to support the following queries in constant time:

rank(x, S): Given $x \in [m]$, return -1 if $x \notin S$, and $|\{y \in S | y < x\}|$ otherwise, and

select(i, S): Given $i \in \{1, \ldots n\}$, return the i-th smallest element in S.

Here the rank operation is only supported for the elements present in the set S. Ajtai [1] showed that the more general problem of supporting rank for every element in the universe has a query lower bound of $\Omega(\lg \lg n)$, even if the space used is polynomial in n. As a consequence, we emphasize the need for handling both S and its complement in the next section.

A dictionary that supports rank operation [52], as well as an indexable dictionary is very useful in representing trees [50] (see Section 37.4.3).

Elias [15] considered the indexable dictionary problem and gave a representation that takes $n \lg m - n \lg n + O(n)$ bits and supports the queries in $O(1)$ time, though he only considered the average case time complexity of the queries. Raman et al. [50] have given an indexable dictionary structure that takes $\mathcal{B} + o(n) + O(\lg \lg m)$ bits. The main idea here is, again, to partition the elements into buckets based on their most significant bits, as in the static dictionary structure of Brodnik and Munro [5]. The difference is that instead of storing explicit pointers to the bucket representations, they store the bucket sizes using a succinct representation that supports partial sum queries (see Section 37.8) in constant time. This not only saves a significant amount of space, but also provides the extra functionality needed for supporting rank and select.

Using similar ideas, one can also represent multisets and collections of sets using almost optimal space. See [50] for details.

37.3.2 Fully Indexable Dictionary

Given a set $S \subseteq [m]$, a *fully indexable dictionary* (FID) of S is a representation that supports rank and select operations on both S and its complement $\bar{S} = [m] \setminus S$ in constant time [50].

It is easy to see that the bitvector representation of a set, with auxiliary structures to support rank and select on both the bits as mentioned in Section 37.2, is an FID. But this requires $m + o(m)$ bits, where m is the size of the universe. Here we look at an FID representation that takes $\mathcal{B} + o(m)$ bits of space. Note that when the set is reasonably sparse (namely when $n = m/\omega(\lg m)$) $\mathcal{B} = o(m)$, and hence it improves the space complexity of the bitvector representation.

Let $S \subseteq [m]$ be a given set of size n. Divide $[m]$ into blocks of consecutive elements, with block size $u = \lfloor \frac{1}{2} \lg m \rfloor$. Let S_i be the subset of S that falls into the i-th block. Each of the S_i's is represented by storing an index into a table that contains the characteristic bitvectors of all possible subsets of a particular size from the universe $[u]$. As a consequence, the space occupied by these representations together with all the precomputed tables can be shown to be $\mathcal{B} + o(m)$ bits. To enable fast access to the representations of these subsets, we store the partial sums of the sizes of the subsets, and also the partial sums of the lengths of the representations of these subsets, which take $O(m \lg\lg m/\lg m)$ bits. This can be used to support rank in $O(1)$ time.

To support select, we first store the positions of every $(\lg^2 m)$-th element explicitly in an array, which takes $O(m/\lg m)$ bits. Call the part the universe that lies between two successive elements in this array a *segment*. If the size of a segment is more than $\lg^4 m$, then we explicitly store all the $\lg^2 m$ elements of S that belong to this segment in sorted order. This takes $\lg^3 m$ bits for every such 'sparse' segment, and hence at most $m/\lg m$ bits, over all the sparse segments. Dense segments are handled by constructing a complete tree with branching factor $\sqrt{\lg m}$, and so constant height, whose leaves are the blocks that constitute this segment, and storing some additional information to navigate this tree efficiently (see the *searchable partial sum* structure in Section 37.8).

To support rank and select on $\bar{S}$, first observe that an implicit representation of a set over a given universe is also an implicit representation of its complement. Thus, we need not store the implicit representations of $\bar{S}_i$ again. Except for this, we repeat the above construction with S_i's replaced by $\bar{S}_i$'s.

The overall space requirement of the structure is $\mathcal{B} + O(m \lg\lg m/\lg m)$ bits, and rank and select are supported on both S and $\bar{S}$ in $O(1)$ time. See [50] for details.

37.3.3 Dynamic Dictionary

We have looked at several succinct structures for static dictionaries. We now briefly consider the dynamic dictionary problem where one can add and delete elements from the set while supporting the membership queries.

Model: The model of memory allocation is very important in dynamic data structures. One widely used model [4, 45, 50] is to assume the existence of a 'system' memory manager that would allocate and free memory in variable-sized chunks. In this model, the space complexity of a structure is counted as the total size of all the blocks allocated for that structure, and hence this approach does not account for the space wastage due to *external fragmentation*.

Fundamentally, memory is most easily viewed as a large array. If we are to use the storage, we must manage it. Therefore a simple view is to count all the fragmentation we may cause and count the memory usage as the difference between the addresses of the first and last locations used by the structure. While more complex scenarios may be more realistic in certain cases, we take this simple address difference model as our focus. The methods we discuss are equivalent under either model up to constant factors.

A balanced tree can be used to support all the dynamic dictionary operations in $O(\lg n)$ time using $n \lg m + O(n \lg n)$ bits, where n is the current size of the set. Using the ideas of the FKS dictionary, Dietzfelbinger et al. [14] gave a dynamic dictionary structure that supports membership in $O(1)$ time and updates (insert/delete) in $O(1)$ expected amortized time. This structure takes $O(n \lg m)$ bits of space. There have been several improvements, lowering the space complexity close to the information theoretic-minimum, culminating in a structure that takes $\mathcal{B} + o(\mathcal{B})$ bits with the same query complexity as above. See [5, 18, 46, 47, 51] and the references therein.

All these structures also support associating satellite information with the elements, so that whenever an element is found to be in the set, we can also retrieve the satellite information associated with it in constant time.

37.4 Tree Representations

Trees are one of the most fundamental objects in computer science. We consider the problem of representing large trees succinctly. Storing a tree with a pointer per child as well as other structural information can account for the dominant storage cost. For example, standard representations of a binary tree on n nodes, using pointers, take $O(n \lg n)$ bits of space. Since there are only $\binom{2n+1}{n}/(2n+1)$ different binary trees on n nodes, less than $2n$ bits suffice to distinguish between them. We look at some binary tree representations that take $2n + o(n)$ bits and support the basic navigational operations in constant time.

37.4.1 Binary Trees

First, if the tree is a complete binary tree (i.e., a binary tree in which every level, except possibly the deepest, is completely filled, and the last level is filled from the left as far as required), then there is a unique tree of a given size and we require no additional space to store the tree structure. In fact, by numbering the nodes from 1 to n in the 'heap order' [59] (left-to-right level-order traversal of the tree), one can support navigational operations on the tree by observing that the parent of a node numbered i is the node numbered $\lfloor i/2 \rfloor$, and the left and right children of node i are $2i$ and $2i + 1$ respectively. But this property does not hold when the tree is not complete.

If the tree is not complete, one could extend it to a complete binary tree with the same height and store a bit vector indicating which nodes are present in the tree (in the heap order of the complete tree) to support the operations efficiently. But this takes space exponential in the number of nodes, in the worst case.

To save space, one can use the following compressed representation due to Jacobson [33]: First, mark all the nodes of the tree with 1 bits. Then add external nodes to the tree, and mark them with 0 bits. Construct a bitvector by reading off the bits that are marking the nodes in left-to-right level-order. (See Figure 37.2.) It is easy to see that the original tree can be reconstructed from this bitvector. For a binary tree with n nodes, this bitvector representation takes $2n + 1$ bits. Moving between parent and child is just a slight twist on the method used in a heap. By storing the rank and select directories for this bitvector, one can support the navigational operations in constant time using the following equations:

$$\mathsf{parent}(i) = \mathsf{select}(\lfloor i/2 \rfloor); \quad \mathsf{leftchild}(i) = 2 \cdot \mathsf{rank}(i); \quad \mathsf{rightchild}(i) = 2 \cdot \mathsf{rank}(i) + 1.$$

37.4.2 Ordinal Trees

Now, consider optimal representations of trees of higher degree, of which there are two different notions.

An *ordinal tree* is a rooted tree of arbitrary degree in which the children of each node are ordered. Ordinal trees on n nodes are in one to one correspondence with binary trees on n nodes. Hence about $2n$ bits are necessary to represent an arbitrary ordinal tree on n nodes. A *cardinal tree* of degree k is a rooted tree in which each node has k positions for an edge to a child. Hence, a binary tree is a cardinal tree of degree 2. There are $C_n^k \equiv \binom{kn+1}{n}/(kn+1)$ cardinal trees of degree k on n nodes [25]. Hence we need roughly $(\lg(k-1) + k \lg \frac{k}{k-1})n$ bits to represent an arbitrary such tree.

The basic operations we would like to support on tree representations are: given a node, finding its parent, i-th child, degree and the size of the subtree rooted at that node (subtree size). For the cardinal trees we also need to support the additional operation of finding a child with a given label.

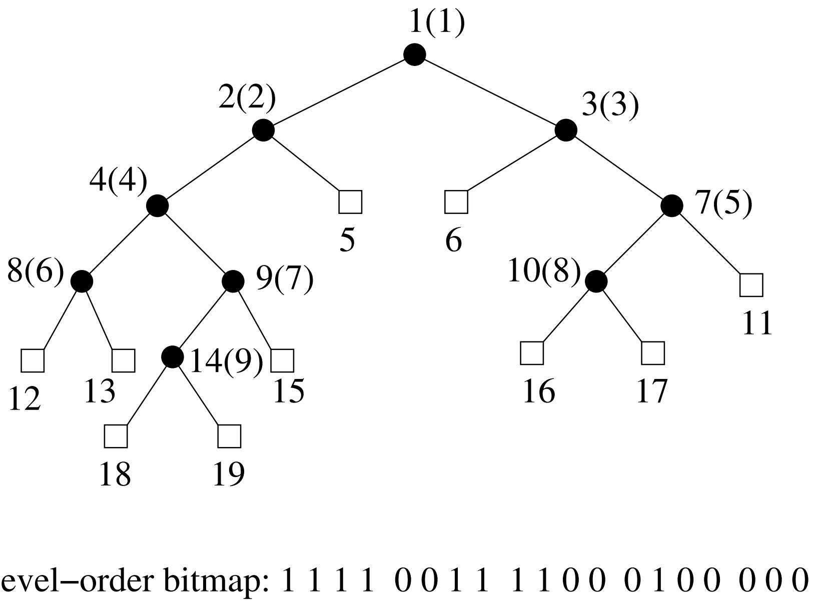


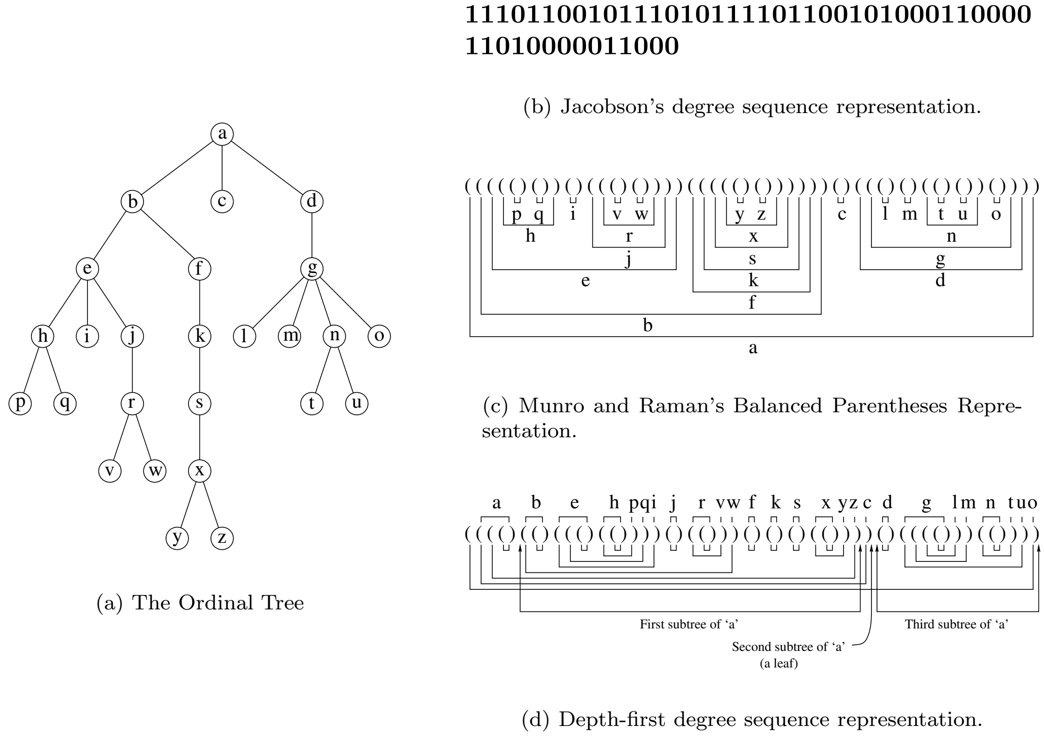
FIGURE 37.2: Level-order bitmap representation of a binary tree.

We outline three different representations of an ordinal tree. All the three representations map the n nodes of the tree onto the integers $1, \ldots, n$, and hence all are appropriate for applications in which data is to associated with nodes or leaves.

Level-order unary degree sequence representation: A rooted ordered tree can be represented by storing its degree sequence in any of a number of standard orderings of the nodes. The ordinal tree encoding of Jacobson [33] represents a node of degree d as a string of d **1**s followed by a **0**. Thus the degree of a node is represented by a binary prefix code. These prefix codes are then written in a level-order traversal of the entire tree. Using auxiliary structures to support rank and select operations on this sequence, one can support finding the parent, the i-th child and the degree of any node in constant time. Thus, it gives a representation that takes $2n + o(n)$ bits of space and supports the above three operations in constant time, for an ordered tree on n nodes.

Balanced parenthesis representation: The tree encoding of Munro and Raman [41] uses a balanced sequence of parentheses to represent an ordinal tree. This balanced representation is derived from the depth-first traversal of the tree, writing an open parenthesis on the way down and a close parenthesis on the way up. Thus, a tree on n nodes can be represented by a balanced parenthesis sequence of length $2n$. Extending the ideas of Jacobson, they showed how to support the following operations in $O(1)$ time, using negligible extra space ($o(n)$ bits):

- findopen/findclose(i): find the position of the open/close parenthesis matching the given close/open parenthesis in position i.
- excess(i): find the difference between the number of open and closing parentheses before the position i.
- enclose(i): given a parenthesis pair whose open parenthesis is in position i, find the open parenthesis corresponding to its closest enclosing matching parenthesis pair.



(a) The Ordinal Tree

(b) Jacobson's degree sequence representation.

(c) Munro and Raman's Balanced Parentheses Representation.

(d) Depth-first degree sequence representation.

FIGURE 37.3: Three encodings of an ordinal tree.

The parent of a node can be found in constant time using the enclose operation. In the parenthesis representation, the nodes of a subtree are stored together, which enables us to support the operation of finding the size of the subtree rooted at a given node in constant time. The problem with this representation is that finding the i-th child takes $\Theta(i)$ time.

Depth-first unary degree sequence representation: Jacobson's representation allows access to the i-th child in constant time, whereas Munro and Raman's representation supports subtree size operation in constant time. To combine the virtues of these two representations, Benoit et al. [2] used a representation that writes the unary degree sequence of each node in the depth-first traversal order of the tree. The representation of each node contains essentially the same information as in Jacobson's level-order degree sequence, but written in a different order. Thus, it gives another $2n$ bit encoding of a tree on n nodes. Replacing the **0**'s and **1**'s by open and close parentheses respectively, and adding an extra open parenthesis at the beginning, creates a string of balanced parentheses. Using auxiliary structures to support rank and select operations on this bit string and also the operations on balanced parenthesis sequences defined above, one can support finding the parent, i-th child, degree and subtree size of a given node in constant time.

Other operations: Sadakane [54] has shown that the parenthesis representation of an ordinal tree can be used to support least common ancestor queries in $O(1)$ time using a $o(n)$-bit auxiliary structure. Munro and Rao [42] have shown that one can also support the level ancestor queries in $O(1)$ time, using an additional $o(n)$ bit auxiliary structure by storing the parenthesis representation. Geary et al. [23] obtained another structure that takes $2n + o(n)$ bits and supports level-ancestor queries, in addition to all the other navigational operations mentioned above in $O(1)$ time.

37.4.3 Cardinal Trees

A simple cardinal tree encoding can be obtained as follows: Encode each node of a k-ary tree by k bits, where the ith bit specifies whether child i is present. These can be written in any fixed ordering of the tree nodes, such as level order or depth-first order, to obtain the tree encoding. By storing the rank and select directories for this bitvector encoding, one can support parent, i-th child and degree queries in constant time. This encoding has the major disadvantage of taking kn bits, far from the lower bound of roughly $(\lg k + \lg e)n$ bits, as there are $\mathcal{C}_n^k \equiv \binom{kn+1}{n}/(kn+1)$ k-ary cardinal trees on n nodes.

Using some probabilistic assumptions, Darragh et al. [11] have implemented a structure that takes $\lg k + O(1)$ bits per node, though the implementation treats $\lg \lg n$ as 'a constant' (indeed 5). This structure supports the navigational operations in constant expected time and also supports updates 'efficiently' (compared with other linear space representations), and was also shown to perform well in practice.

To achieve a better space bound with good worst-case performance, one can use the ordinal tree encoding to store the underlying tree, and store some additional information about which children are present at each node. The ordinal information (using the depth-first unary degree sequence representation) can be used to support the parent, i-th child, degree and subtree size queries in constant time.

Let $S_x = \{i_1, i_2, \ldots, i_d\}$ be the child labels of a node x with degree d in the cardinal tree. To find the child labeled j of node x, it suffices to find $i = \mathsf{rank}(j)$ in the set S_x, if $j \in S_x$. If $i = -1$ (i.e., $j \notin S_x$), then there is no child labeled j at node x, otherwise the i-th child of x is the child labeled j of node x. The i-th child can be found using the ordinal information. Storing each of these sets S_x using the indexable dictionary representation of Section 37.3.1, which takes $d \lg k + o(d) + O(\lg \lg k)$ bits for each S_x, requires $n \lg k + o(n) + O(n \lg \lg k)$ bits in the worst case. Using a representation that stores a collection of indexable dictionaries efficiently [50], one can reduce the space consumption to $n \lg k + o(n) + O(\lg \lg k)$ bits.

Thus, this structure uses $2n + o(n)$ bits to represent the underlying ordinal tree, $n \lg k + o(n + \lg k)$ bits to represent the labels of the children at each node, and supports all the navigational operations and the subtree size operation in $O(1)$ time.

Using the succinct indexable dictionary structure mentioned in Section 37.3, Raman et al. [50] obtained an optimal space cardinal tree representation. The main idea is to store the set of all pairs, $\langle i, j \rangle$ such that the i-th node, in the level-order of the nodes, has a child labeled j, using an indexable dictionary representation. (See Figure 37.4 for an example.) Since this set is of size n from the universe $[nk]$, it requires $\lg \binom{nk}{n} + o(n + \lg k) = \mathcal{C}_n^k + o(n + \lg k)$ bits to store an indexable dictionary for this set. One can easily map the navigational operations on the tree to the operations on this set, to support them in constant time. But this structure does not support the subtree size operation efficiently.

37.4.4 Dynamic Binary Trees

All the tree representations mentioned so far are static. Even to make a minor modification to the tree, such as adding a leaf, the entire structure has to be reconstructed (in the worst case). In this section we look at some representations that are more efficient in supporting updates to the tree.

Munro et al. [45] gave a binary tree representation that takes $2n + o(n)$ bits, and supports parent, left child, right child and subtree size operations in $O(1)$ time. Updating the tree (adding a leaf or adding a node along an edge) requires $O(\lg^c n)$ time, for some constant

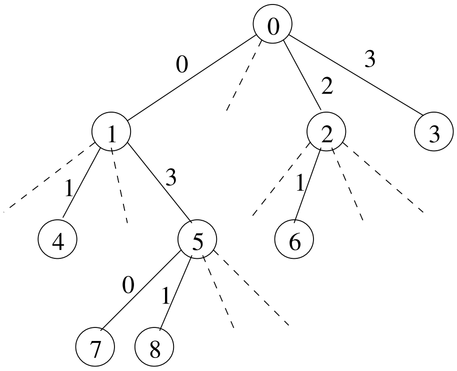


FIGURE 37.4: The tree is represented by storing an indexable dictionary of the set $\{\langle 0,0\rangle, \langle 0,2\rangle, \langle 0,3\rangle, \langle 1,1\rangle, \langle 1,3\rangle, \langle 2,1\rangle, \langle 5,0\rangle, \langle 5,1\rangle\}$.

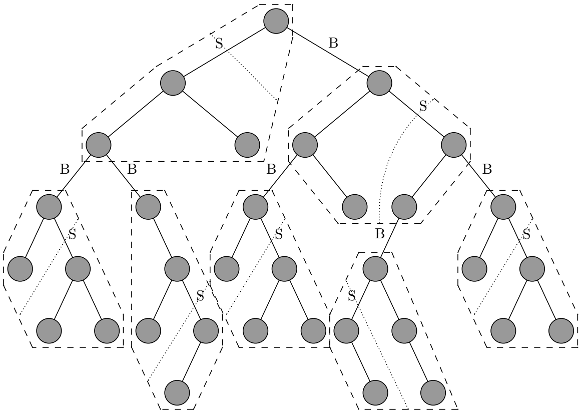


FIGURE 37.5: Dynamic binary tree representation. B denotes an inter-block pointer and S denotes an inter-subblock pointer.

$c \geq 1$ which depends on the size of the data associated with the nodes. Extending some of the ideas involved in this, Raman and Rao [51] improved the update time to $O((\lg\lg n)^\epsilon)$, for any fixed $\epsilon > 0$, while maintaining the other time and space bounds.

We briefly outline the key issues involved in the construction of these structures. First, we divide the tree into blocks of size $\Theta(\lg^c n)$, for some $c \geq 2$, and each block in turn into sub-blocks of size $\epsilon \lg n$, for some fixed $\epsilon < 1$. The sub-blocks are stored using an implicit representation and are operated upon using precomputed tables. The block structure of the tree is stored using explicit pointers. Since there are only $\Theta(\lg^{c-1} n)$ sub-blocks in each block, we can store the sub-block structure within a block explicitly using $\Theta(\lg\lg n)$ sized pointers. Each block stores its parent block and the size, using a constant number of words. Thus, the overall block structure of the tree is easily handled by conventional means (storing explicit pointers) and only takes $O(n/\lg n)$ bits. The blocks and sub-blocks tolerate some slack in their sizes and are moved to appropriate sized areas to avoid wasting space. Ultimately, the key issues boil down to the memory management.

To support subtree size, we maintain the the subtree sizes of the roots of all blocks and sub-blocks. Since each update changes the subtree sizes of several nodes, it is not possible to update all the effected blocks and sub-blocks in constant time, in general. For this reason, we assume that the navigation through the tree begins at the root and may end at any point (or at the root, to achieve worst-case constant time for updates), and navigates the tree by moving from a node only to either its parent or one of its children. Hence, updates to a node higher in the tree regarding the insertions and deletions to descendants are made on return to that node.

37.5 Graph Representations

In this section, we briefly describe some space efficient representations of graphs. In particular, we consider representations that take close to the information theoretic minimum space and support degree and adjacency queries efficiently. A degree query asks for the degree of a given node in the graph, and an adjacency query asks whether two given vertices are adjacent or not in the graph. In addition, we would also like to support listing all the vertices adjacent to a given vertex.

Turán [57] gave a linear time constructible representation of an arbitrary planar graph that takes at most 12 bits per node. Though this gives a space efficient representation of planar graphs, it does not support the queries efficiently. Kannan et al. [34] have given an *implicit* (linear space) graph representation that supports adjacency queries using $O(\lg n)$ bit probes.

Jacobson [33] gave a representation that takes $O(n)$ bits of space to represent a planar graph on n nodes and supports degree and adjacency queries in constant time. It uses a simple mapping of one-page graphs to sequences of balanced parentheses, and the fact that a planar graph always has a 4-page embedding. By storing auxiliary structures to support some natural operations on these sequences (see Section 37.4.2), one can also support the navigational operations in optimal time.

Munro and Raman [41] improved the space to $8n + 2m + o(n)$ bits, for a planar graph on n vertices with m edges, still supporting the queries in constant time. In general, their representation takes $2kn + 2m + o(nk + m)$ bits to store a k page graph on n vertices and m edges and supports degree and adjacency queries in $O(k)$ time.

There have been several improvements [7, 8, 29, 37, 38], improving the space close to the information theoretic-lower bound, simultaneously expanding the class of graphs for which the scheme works. In particular, Lu [38] gave an optimal space (within lower-order terms) representation that can be constructed in linear time. This supports degree and adjacency queries in $O(1)$ time for constant-genus graphs.

The main idea is to partition the given graph G on n vertices into $o(n/\lg n)$ disjoint subgraphs of size $O(\lg^6 n)$ by removing a subgraph H of size $o(n/\lg n)$. This is done using a 'planarization algorithm' for bounded genus graphs, and an algorithm to construct a 'separator decomposition tree' of a planar graph. The representation of G is obtained by storing a rerepresentation of H, and recursing on each of the smaller subgraphs upto a constant number of levels, after which one can use a precomputed table to operate on the small subgraphs. See [38] for details.

37.6 Succinct Structures for Indexing

A *text index* is a data structure storing a *text* (a string or a set of strings) and supporting string matching queries: given a pattern P, find all the occurrences of P in the text. Two well-known and widely used index structures are the suffix trees and suffix arrays. In this section we briefly describe some succinct data structures for these two.

A *suffix tree* for a text is the *compressed digital trie* of all the suffixes of the text [39, 58]. A suffix tree for a text of length n has n leaves and at most $n-1$ internal nodes. The space bound is a consequence of skipping nodes with only one child, hence there are precisely $n-1$ internal nodes if we use a binary trie. Each leaf points to the position in the text of the corresponding suffix it represents uniquely. The edges are labeled by substrings of the text, which are usually represented by storing a position in the text where the substring starts and its length. Thus, a standard representation of a suffix tree for a text of length n takes $O(n \lg n)$ bits. Searching for an occurrence of a pattern of length m using a suffix tree takes $O(m)$ time.

The *suffix array* of a text is an array storing pointers to the suffixes of the text in their lexicographic order. Thus, a suffix array for a text of length n takes $n\lceil \lg n\rceil$ bits. Note that the leaf labels of a suffix tree written from left to right form the suffix array, if the children of each node are arranged in lexicographic order of their edge labels. Searching for an occurrence of a pattern of length m using a suffix array takes $O(m + \lg n)$ time.

We now briefly sketch the ideas involved in representing a suffix tree (and hence also a suffix array) using $O(n)$ bits. We first convert the trie into binary by using a fixed length encoding of the characters of the alphabet. We then store the parenthesis representation of the underlying tree structure (see Section 37.4.2). The edge labels of a suffix tree can be omitted, as this can be determined by finding the longest common prefix of the leftmost and rightmost leaves of the parent node (of the edge). The parenthesis representation of an ordinal tree can be augmented with $o(n)$-bit additional structure to support finding the leftmost and rightmost leaves of a given node in constant time. Thus, one can use this tree representation to store the tree structure of a suffix tree, and store the leaf pointers (suffix array) explicitly. This gives a suffix tree representation that takes $n \lg n + O(n)$ bits of space and supports indexing queries in optimal time. See [44] for details.

The above structure uses $n\lceil \lg n\rceil$ bits to represent the pointers to the text or the suffix array. Grossi and Vitter [26] obtained a suffix array structure that takes $O(n)$ bits and supports finding the i-th element in the suffix array (lookup queries) in $O(\lg^{\epsilon} n)$ time, for any fixed $\epsilon > 0$. Using this structure they also obtained a suffix tree representation that takes $O(n)$ bits of space and supports finding all the s occurrences of a given pattern of length m in $O(m + s \lg^{\epsilon} n)$ time. The structure given by Rao [53] generalizes the suffix array structure of Grossi and Vitter, which takes $O(nt(\lg n)^{1/(t+1)})$ bits and supports lookup in $O(t)$ time, for any parameter $1 \leq t \leq \lg\lg n$. Using this structure, one can get an index structure that takes $o(n \lg n)$ bits and supports finding all the s occurrences of a given pattern of length m in $O(m + s + \lg^{\epsilon} n)$ time.

Ferragina and Manzini [16] presented an *opportunistic* data structure taking $O(nH_k(n)) + o(n)$ bits of space, where $H_k(n)$ denotes the k-th order entropy of the given text of length n. This supports finding all the occurrences of a pattern of length m in $O((m + s)\lg^{\epsilon} n)$ time, where s is the number of occurrences of the pattern. They also presented its practical performance [17].

Sadakane [54] gave a data structure that takes $O(n \cdot (1 + H_0) + O(|\Sigma| \lg |\Sigma|))$ bits for a text of length n over an alphabet Σ, where $H_0 \leq \lg |\Sigma|$ is the order-0 entropy for the text. This supports finding all the s occurrences of a given pattern P in $O(|P| \lg n + s \lg^{\epsilon} n)$ time, and *decompress* a portion of the text of length l in $O(l + \lg^{\epsilon} n)$ time, for any fixed $\epsilon > 0$.

Grossi et al. [27] gave another index structure that takes $nH_k(n) + O(n \lg\lg n \lg |\Sigma| / \lg n)$ bits for a text of length n over an alphabet Σ. Finding an occurrence of a pattern of length m using this structure takes $O(m \lg |\Sigma| + polylog(n))$ time. This is also shown to perform well, in terms of space as well as query times, in practice [28]

37.7 Permutations and Functions

37.7.1 Permutations

Permutations are fundamental in computer science and have been the focus of extensive study. Here we consider the problem of representing permutations succinctly to support computing $\pi^k(i)$ for any integer k, where $\pi^0(i) = i$ for all i; $\pi^k(i) = \pi(\pi^{k-1}(i))$ when $k > 0$ and $\pi^k(i) = \pi^{-1}(\pi^{k+1}(i))$ when $k < 0$.

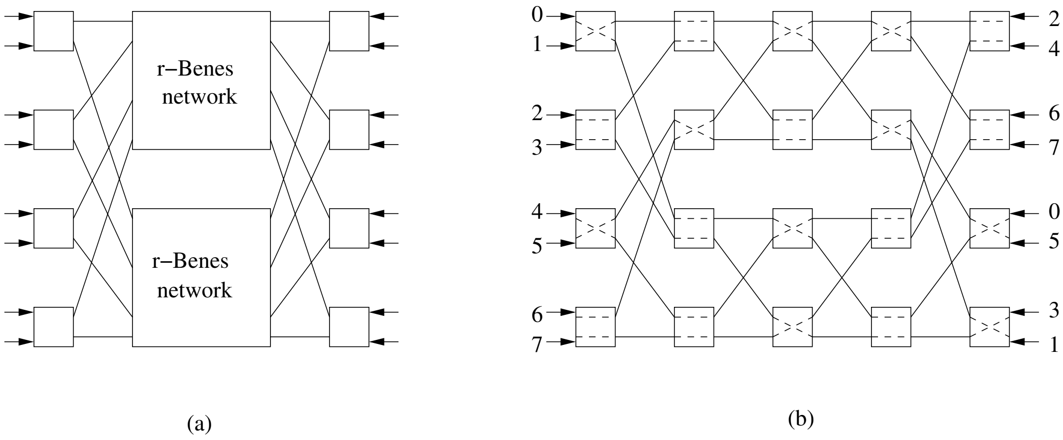
The most obvious way of representing an arbitrary permutation, π, of the integers $\{0, 1, \ldots, n-1\}$ is to store the sequence $\pi(0), \pi(1), \ldots, \pi(n-1)$. This takes $n \lceil \lg n \rceil$ bits, which is $\Theta(n)$ bits more than the information-theoretic lower bound of $\lg(n!) \approx n \lg n - n \lg e$ bits. This representation can be used to find $\pi(i)$ in $O(1)$ time, but finding $\pi^{-1}(i)$ takes $O(n)$ time in the worst case, for $0 \leq i \leq n-1$. Using this representation, one can easily compute $\pi^k(i)$ in k steps, for $k \geq 1$. To facilitate the computation in constant time, one could store $\pi^k(i)$ for all i and k ($|k| \leq n$, along with its cycle length), but that would require $\Theta(n^2 \lg n)$ bits. The most natural compromise is to retain $\pi^k(i)$ with $|k| \leq n$ a power of 2. Unfortunately, this $n \lceil \lg n \rceil^2$ bit representation leaves us with a logarithmic time evaluation scheme and a factor of $\lg n$ from the minimal space representation.

We first show how to augment the standard representation to support π^{-1} queries efficiently, while avoiding most of the extra storage cost one would expect. In addition to storing the standard representation, we trace the cycle structure of the permutation, and for every cycle whose length is at least t, we store a shortcut pointer with the elements which are at a distance of a multiple of t steps from an arbitrary starting point. The shortcut pointer points to the element which is t steps before it in the cycle of the permutation. This *shortcut representation* of a permutation can be stored using $(1 + 1/t)n \lg n + o(n)$ bits, and it supports π queries in $O(1)$ time and π^{-1} queries in $O(t)$ time, for any parameter $1 \leq t \leq n$.

Consider the cycle representation of a permutation π over $\{0, 1, \ldots, n-1\}$, which is a collection of disjoint cycles of π (where the cycles are ordered arbitrarily). Let σ be this permutation, i.e., the standard representation of σ is a cycle representation of π. Let B be a bit vector of length n that has a **1** corresponding to the starting position of each cycle of π and **0** everywhere else, together with its rank and select directories with respect to both bits. Let S be a representation of σ that supports $\sigma(i)$ and $\sigma^{-1}(i)$ queries efficiently. Then to find $\pi^k(i)$, first find the index j of the cycle to which $\sigma^{-1}(i)$ belongs, using B and S. Find the length l of the j-th cycle and the number p of elements up to (but not including) the j-th cycle. Then, one can verify that $\pi^k(i) = \sigma(p + (i - p + k \bmod l))$. Combining this with the shortcut representation, one can get a representation taking $(1 + \epsilon)n \lg n + O(1)$ bits that supports computing arbitrary powers in $O(1)$ time.

Benes network: A *Benes network* [36] is a communication network composed of a number of switches. Each switch has 2 inputs x_0 and x_1 and 2 outputs y_0 and y_1 and can be configured either so that x_0 is connected to y_0 (i.e. a packet that is input along x_0 comes out of y_0) and x_1 is connected to y_1, or the other way around. An r-Benes network has 2^r inputs and outputs, and is defined as follows. For $r = 1$, the Benes network is a single switch with 2 inputs and 2 outputs. An $(r+1)$-Benes network is composed of 2^{r+1} switches and two r-Benes networks, connected as as shown in Fig. 37.6(a). A particular setting of the switches of a Benes network *realizes* a permutation π if a packet introduced at input i comes out at output $\pi(i)$, for all i. See Fig. 37.6(b) for an example.

Clearly, a Benes network may be used to represent a permutation. For example, if $n = 2^r$, a representation of a permutation π on $[n]$ may be obtained by configuring an r-Benes network to realize π and then listing the settings of the switches in some canonical



(a)

(b)

FIGURE 37.6: Benes network: (a) construction of $(r+1)$-Benes network (b) Benes network realizing the permutation $(4, 7, 0, 6, 1, 5, 2, 3)$.

order (e.g. level-order). This represents π using $r2^r - 2^{r-1} = n \lg n - n/2$ bits. Given i, one can trace the path taken by a packet at input i by inspecting the appropriate bits in this representation, and thereby calculate $\pi(i)$ in $O(\lg n)$ time (indeed, in $O(\lg n)$ bit-probes). In fact, by tracing the path back from output i we can also compute $\pi^{-1}(i)$ in $O(\lg n)$ time.

One can compress the middle levels of a Benes network by storing an implicit representation of the permutation represented by the middle $O(\lg n/ \lg \lg n)$ levels. This reduces the space to $\lg(n!) + o(n)$ bits. One can also group the remaining bits of this Benes network into words of size $\Theta(\lg n)$ bits (by taking $O(\lg \lg n)$ consecutive levels and $O(\lg \lg n)$ appropriate rows). This enables us to traverse $\Theta(\lg \lg n)$ levels in a Benes network in $O(1)$ time. Thus, it gives a representation that takes the optimal $\lceil \lg(n!) \rceil + o(n)$ bits, and supports computing arbitrary powers in $O(\lg n/ \lg \lg n)$ time.

One can obtain a structure with same time and space bounds even when n is not a power of 2. See [43] for details.

37.7.2 Functions

Now consider the problem of representing arbitrary functions $f : [n] \rightarrow [n]$, so that queries for $f^k(i)$, for any integer k can be answered efficiently. Here $f^0(i) = i$ and for any $k > 0$, $f^k(i) = f(f^{k-1}(i))$ and $f^{-k}(i) = \{j | f^k(j) = i\}$, for all i. This is a generalization of the problem considered in the previous section. Since there are n^n functions from $[n]$ to $[n]$, any representation scheme takes at least $\lceil n \lg n \rceil$ bits to store an arbitrary function.

A standard way of representing a function is to store the sequence $f(i)$, for $i = 0, \ldots, n-1$. This representation does not support the efficient evaluation of $f^k(i)$ for $k >> 1$. We look at a representation that takes $(1+\epsilon) n \lg n + O(1)$ bits of space to store a function $f : [n] \rightarrow [n]$ and supports computing arbitrary positive powers in constant time and negative powers $f^{-k}(i)$, in $O(1 + |f^{-k}(i)|)$ time.

Given an arbitrary function $f : [n] \rightarrow [n]$, consider the directed graph, $G_f = (V, E)$, obtained from it, where $V = [n]$ and $E = \{(i, j) : f(i) = j\}$. In general this directed graph consists of a disjoint set of subgraphs, each of which is a directed cycle with trees rooted at the nodes on the cycle and edges directed towards the roots. See Figure 37.7 for an example.

The main idea of the solution is as follows: in each directed cycle, we re-order the nodes

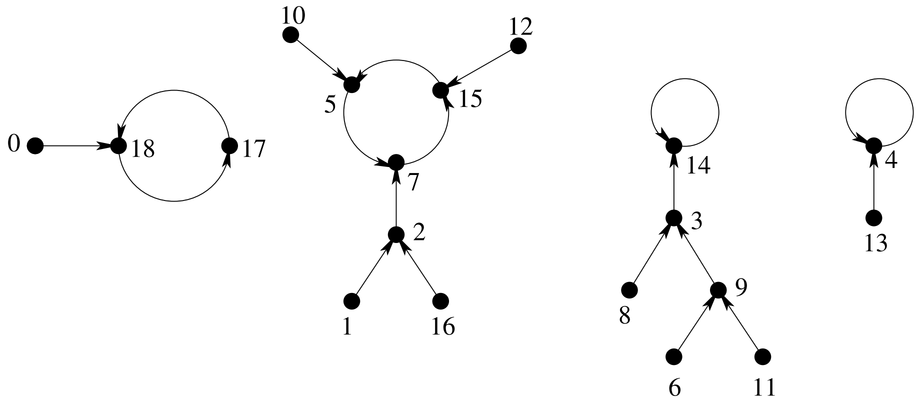


FIGURE 37.7: Graph representation of the function $f(x) = (x^2 + 2x - 1) \bmod 19$, for $0 \leq x \leq 18$.

of each tree such that the leftmost path of any subtree is the longest path in that subtree. This enables finding a node at a given depth from any internal node, if it exists, in constant time using the parenthesis representation. We then preprocess each of the trees and store auxiliary structures to support level-ancestor queries on them in constant time (see Section 37.4.2). Observe that finding $f^k(i)$, for $k > 0$, can be translated to finding the ancestor of node i which is k levels above it, if i is at a depth at least k in its tree T. Otherwise, we have to traverse the cycle to which the root of T belongs, to find the required answer. This can be done by storing these cycles as a permutation.

When i belongs to one of the trees in a subgraph, one can answer $f^k(i)$ queries for $k < 0$ in optimal time by finding all the nodes that are at the k-th level in the subtree rooted at i. Otherwise, if i is part of the cycle in the subgraph, we store an auxiliary structure that, for any given i and k, outputs all the trees in the subgraph containing i that have an answer in time proportional to the number of such nodes. From this, one can easily find the required answer in optimal time. The auxiliary structure takes $O(m)$ bits for a subgraph with m nodes, and hence $O(n)$ bits overall. See [42] for details.

For functions from $[n] \rightarrow [m]$ one can show the following: If there is a representation of a permutation that takes $P(n)$ space to represent a permutation on $[n]$ and supports forward in t_1 time and inverse in t_2 time, then there is a representation of a function from $[n]$ to $[m]$, $m \leq n$ that takes $(n - m) \lg m + P(m) + O(m)$ bits, and supports $f^k(i)$ in $O(t_1 + t_2)$ time, for any positive integer k and for any $i \in [n]$. When $m > n$, larger powers are not defined in general. In this case, we can have a structure that takes $n \lg m + P(n) + O(n)$ bits of space and answers queries for positive powers (returns the power if defined or returns -1 otherwise) in $O(t_1 + t_2)$ time.

37.8 Partial Sums

Let $a_1, a_2, \ldots, a_n$ be a sequence of n non-negative k-bit numbers. The *partial sums* problem maintains the sequence under the following operations:

- $\mathsf{sum}(i)$: return $\sum_{j=1}^{i} a_j$,
- $\mathsf{update}(i, \delta)$: set $a_i \leftarrow a_i + \delta$, for some integer δ such that $0 \leq a_i + \delta \leq 2^k - 1$. Our later solutions have the additional constraint that $|\delta| \leq \lg^{O(1)} n$.

Dietz [13] gave a structure for the partial sum problem that supports sum and update in $O(\lg n/\lg\lg n)$ time using $O(n\lg n)$ bits of extra space, for the case when $k = \Theta(\lg n)$ and no constraints on δ. The time bounds are optimal due to the lower bound of Fredman and Saks [20]. As the information-theoretic space lower bound is kn bits, this structure uses space within a constant factor of the optimal.

The main idea of this structure is to store the elements at the leaves of a complete tree with branching factor $O(\lg^{\epsilon} n)$ for some $\epsilon < 1$. The operations are performed by traversing a path from a leaf to the root, querying/updating the nodes along the path.

The *searchable partial sums* problem is an extension of the partial sums problem that also supports the following operation:

- search(j): find the smallest i such that sum(i) $\geq j$.

When $k = 1$ (i.e., each element is a bit), the special case is commonly known as the *dynamic bit vector* problem, which maintains a bit vector of length n under rank, select and flip (update) operations.

For the searchable partial sums problem there is a structure that supports all operations in $O(\lg n/\lg\lg n)$ time, and uses $kn + o(kn)$ bits of space [49]. For $k = O(\lg\lg n)$, one can also obtain a structure that again takes $kn + o(kn)$ bits and supports sum and search in $O(\log_b n)$ time and update in $O(b)$ amortized time, for any parameter $b \geq \lg n/\lg\lg n$ [30]. For the partial sums problem, one can support the above trade-off for $k = O(\lg n)$ [49], and the time bounds can be shown to be optimal [20].

For the dynamic bit vector problem, one can support rank and select in $O(\log_b n)$ time and flip in $O(b)$ (worst-case) time, for any parameter $\lg n/\lg\lg n \leq b \leq n$, using $o(n)$ bits of extra space. One can also extend the above trade-off for $k = O(\lg\lg n)$, using $kn + o(kn)$ bits of space.

See [49] and [30] for details.

37.9 Arrays

37.9.1 Resizable Arrays

A basic problem that arises in many applications is accumulating elements into a list when the number of elements is unknown ahead of time. The operations needed from such a structure are the ability to append elements to the end of the structure, removing the last element from the structure (in applications such as implementing a stack) and some method of accessing the elements currently in the structure.

One simple solution is a linked list which can easily grow and shrink, and supports sequential access. But this does not support random access to the elements. Moreover, its space overhead is $O(n)$ pointers to store n elements.

Another standard solution is the *doubling technique* [3]. Here the elements are stored in an array. Whenever the array becomes full, an array of double its size allocated and all the elements are copied to it. Similarly, whenever the array shrinks so that it is only one-fourth full, an array of half its size is allocated and all the elements are copied to it. The advantage of this solution over the linked lists is that random access to the elements takes only $O(1)$ time (as opposed to $O(n)$ for linked lists). The amortized update time is $O(1)$, though the worst-case update time is $O(n)$. The space overhead of this solution is $O(n)$.

Sitarski [56] has proposed a solution whose space overhead is only $O(\sqrt{n})$. The idea is to divide the given list of n elements into sublists of size $\lceil\sqrt{n}\rceil$, store them in separate arrays, and store an array (of length $O(\sqrt{n})$) of pointers to these sublists (in order). Whenever

$\lceil\sqrt{n}\rceil$ changes, the entire structure is reconstructed with the new size. Thus the amortized update time is $O(1)$ (though the worst-case time is $O(n)$). This also supports random access in $O(1)$ time.

Brodnik et al. [4] gave a structure that takes $O(\sqrt{n})$ extra locations, where n is the current size of the array, and supports the operations in $O(1)$ time. One advantage of this structure is that elements are never re-allocated. They have also shown that any such structure requires $\Omega(\sqrt{n})$ extra locations even if there are no constraints on the access time.

37.9.2 Dynamic Arrays

A resizable array supports adding/deleting elements only at the end of the list, but does not support insertion/deletion of elements at arbitrary positions in the array. A *dynamic array* is data structure that maintains a sequence of records under the following operations:

- access(i): return the i-th record in the sequence,
- insert(r, i): insert the record r at position i in the sequence, and
- delete(i): delete the i-th record in the sequence.

A standard way of implementing a dynamic array is to store the records in an array and maintain it using the doubling technique. This supports access in $O(1)$ but requires $O(n)$ time to support insert and delete operations.

Goodrich and Kloss [24] gave a structure, the *tiered vector* , that takes $n + O(\sqrt{n})$ words of space to represent a sequence of length n, where each record fits in a word. This structure supports access in $O(1)$ time and updates in $O(\sqrt{n})$ amortized time. The major component of a tiered vector is a set of *indexable circular deques.* A deque is a linear list which provides constant time insert and delete operations at either the head or the tail of the list [35]. A circular deque is a list which is stored in a sequential section of memory of fixed size. An indexable circular deque maintains pointers h and t, which reference the index in memory of the head and tail of this list. A tiered vector is a set of indexable circular deques. Insertions and deletions in an arbitrary indexable circular deque require time linear in its size, but inserting/deleting at either the head or the tail of the list takes $O(1)$ time.

Thus, by maintaining the given sequence of n elements using $O(\sqrt{n})$ indexable circular deques each of size $O(\sqrt{n})$, one can support access in $O(1)$ time and updates in $O(\sqrt{n})$ amortized time. One can easily generalize this structure to one that supports access in $O(1/\epsilon)$ time and updates in $O(n^{\epsilon})$ time, for any parameter $0 < \epsilon \leq 1$.

Using this structure to represent a block of $O(\lg^{O(1)} n)$ records, Raman et al. [49] gave a structure that supports access and updates in $O(\lg n/\lg\lg n)$ amortized time, using $o(n)$ bits of extra space. The main idea is to divide the given list of length n into sublists of length between $\frac{1}{2}\lg^4 n$ and $2\lg^4 n$, and store the sublists using the above dynamic array structure. One can maintain these sublists as the leaves of a weight-balanced B-tree with branching factor $O(\sqrt{\lg n})$, and hence height $O(\lg n/\lg\lg n)$.

By restricting the length of the array, Raman and Rao [51] obtained a dynamic array structure that maintains a sequence of $l = O(w^{O(1)})$ records of $r = O(w)$ bits each, where w is the word size. This structure supports *access* in $O(1)$ time and updates in $O(1 + lr/kw)$ amortized time, and uses $lr + O(k \lg l)$ bits, for any parameter $k \leq l$. The data structure also requires a precomputed table of size $O(2^{\epsilon w})$ bits, for any fixed $\epsilon > 0$. The main idea is to store the newly added elements separately from the existing elements, and store a structure to indicate all the positions of the 'updated' elements. The structure is rebuilt after every k updates.

37.10 Conclusions

We looked at several succinct data structures that achieve almost optimal space while supporting the required operations efficiently. Apart from being of theoretical interest, succinct data structures will also have many practical applications due to the enormous growth in the amount of data that needs to be stored in a structured fashion.

Most of the succinct data structures we presented here can be constructed in linear time from the standard representation. But this method requires more space than necessary during the construction. Developing algorithms that directly construct the succinct representations without using more space during construction, preferably in optimal time, is an important open problem. See [31] for an example.

Another aspect, that is more of theoretical significance, is to study the cell probe (in particular, bit probe) complexity of succinct data structures [6, 22, 48]. For most problems, no bounds other than the straightforward translations from the bounds on the RAM model are known. It is also interesting to consider the time-space trade-offs of these structures.

References

[1] Miklós Ajtai. A Lower Bound for Finding Predecessors in Yao's Cell Probe Model. *Combinatorica*, 8(3): 235-247, 1988.

[2] David Benoit, Erik D. Demaine, J. Ian Munro, and Venkatesh Raman. Representing Trees of Higher Degree. *Proceedings of the Workshop on Algorithms and Data Structures*, LNCS 1663: 169–180, 1999.

[3] John Boyer. Algorithm Alley: Resizable Data Structures. *Dr. Dobb's Journal*, 23(1), January 1998.

[4] Andrej Brodnik, Svante Carlsson, Erik D. Demaine, J. Ian Munro, and Robert Sedgewick. Resizable Arrays in Optimal Time and Space. *Proceedings of the Workshop on Algorithms and Data Structures*, LNCS 1663: 37–48, 1999.

[5] Andrej Brodnik and J. Ian Munro. Membership in Constant Time and Almost Minimum Space. *SIAM Journal on Computing*, 28(5): 1627–1640, 1999.

[6] Harry Buhrman, Peter Bro Miltersen, Jaikumar Radhakrishnan, and Srinivasan Venkatesh. Are Bitvectors Optimal? *Proceedings of the ACM Symposium on Theory of Computing*, 449–458, 2000.

[7] Y.-T. Chiang, C.-C. Lin, and Hsueh-I Lu. Orderly Spanning Trees with Applications to Graph Drawing and Graph Encoding. *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 506–515, 2001.

[8] Richie C. Chuang, Ashim Garg, Xin He, Ming-Yang Kao, and Hsueh-I Lu. Compact Encodings of Planar Graphs via Canonical Orderings and Multiple Parentheses. *Proceedings of the International Colloquium on Automata, Languages and Programming*, LNCS 1443: 118–129, 1998.

[9] David R. Clark. Compact Pat Trees. Ph.D. Thesis, University of Waterloo, 1996.

[10] David R. Clark and J. Ian Munro. Efficient Suffix Trees on Secondary Storage. *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 383–391, 1996.

[11] John J. Darragh, John G. Cleary, and Ian H. Witten. Bonsai: a Compact Representation of Trees. *Software Practice and Experience*, 23(3): 277–291, 1993.

[12] Erik D. Demaine. Algorithm Alley: Fast and Small Resizable Arrays. *Dr. Dobb's Journal*, 326: 132-134, July 2001.

[13] Paul F. Dietz. Optimal Algorithms for List Indexing and List Ranking. *Proceedings of the Workshop on Algorithms and Data Structures*, LNCS 382: 39–46, 1989.

[14] Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der

Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic Perfect Hashing: Upper and Lower Bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

[15] Peter Elias. Efficient Storage and Retrieval by Content and Address of Static Files. *Journal of the ACM*, 21(2): 246-260 (1974)

[16] Paolo Ferragina and Giovanni Manzini. Opportunistic Data Structures with Applications. *Proceedings of the Annual Symposium on Foundations of Computer Science*, 390–398, 2000.

[17] Paolo Ferragina and Giovanni Manzini. An Experimental Study of an Opportunistic Index. *Proceedings of the Annual Symposium on Foundations of Computer Science*, 269-278, 2001.

[18] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space Efficient Hash Tables with Worst Case Constant Access Time. *Proceedings of Symposium on Theoretical Aspects of Computer Science*, LNCS 2607: 271–282, 2003.

[19] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a Sparse Table with $O(1)$ Worst Case Access Time. *Journal of the ACM*, 31(3): 538–544, 1984.

[20] Michael L. Fredman and Michael E. Saks. The Cell Probe Complexity of Dynamic Data Structures. *Proceedings of the ACM Symposium on Theory of Computing*, 345–354, 1989.

[21] Michael L. Fredman and Dan E. Willard. Trans-Dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths. *Journal of Computer and System Sciences*, 48(3): 533–551, 1994.

[22] Anna Gál and Peter Bro Miltersen. The Cell Probe Complexity of Succinct Data Structures. *Proceedings of the International Colloquium on Automata, Languages and Programming*, LNCS 2719: 332–344, 2003.

[23] Richard F. Geary, Rajeev Raman, and Venkatesh Raman. Succinct Ordinal Trees with Level-ancestor Queries. *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 1–10, 2004.

[24] Michael T. Goodrich and John G. Kloss II. Tiered Vectors: Efficient Dynamic Array for JDSL. *Proceedings of the Workshop on Algorithms and Data Structures*, LNCS 1663: 205–216, 1999.

[25] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*, Addison-Wesley, 1989.

[26] Roberto Grossi and Jeffrey S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *Proceedings of the ACM Symposium on Theory of Computing*, 397–406, 2000.

[27] Roberto Grossi, Ankur Gupta, and Jeffrey S. Vitter. High-order Entropy-compressed Text Indexes. *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 841–850, 2003.

[28] Roberto Grossi, Ankur Gupta, and Jeffrey S. Vitter. When Indexing Equals Compression: Experiments with Compressing Suffix Arrays and Applications. *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 629–638, 2004.

[29] Xin He, Ming-Yang Kao, and Hsueh-I Lu. Linear-time Succinct Encodings of Planar Graphs via Canonical Orderings. *SIAM Journal on Discrete Mathematics*, 12: 317–325, 1999.

[30] Wing-Kai Hon, Kunihiko Sadakane, Wing-Kin Sung. Succinct Data Structures for Searchable Partial Sums. *Proceedings of the International Symposium on Algorithms and Computation*, LNCS 2906, 505–516, 2003.

[31] Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Breaking a Time-and-Space Barrier in Constructing Full-Text Indices. *Proceedings of the Annual Symposium on Foundations of Computer Science*, 251–260, 2003.

[32] Guy Jacobson. Space-efficient Static Trees and Graphs. *Proceedings of the Annual Symposium on Foundations of Computer Science*, 549–554, 1989.

[33] Guy Jacobson. Succinct Static Data Structures. Ph.D. thesis, Carnegie Mellon University, 1988.

[34] Sampath Kannan, Moni Naor, Steven Rudich. Implicit Representation of Graphs. *Proceedings of the ACM Symposium on Theory of Computing*, 334–343, 1988.

[35] Donald E. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. Addison-Wesley, Third edition, 1997.

[36] Frank T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees and Hypercubes*. Computer Science and Information Processing, Morgan Kauffman, 1992.

[37] Hsueh-I Lu. Linear-time Compression of Bounded-genus Graphs into Information-theoretically Optimal Number of Bits. *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 223–224, 2002.

[38] Hsueh-I Lu. Linear-Time Information-theoretically Optimal Encodings Supporting Constant-time Queries for Constant-genus Graphs. Manuscript, 2002.

[39] Edward M. McCreight. A Space-economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2): 262–272, 1976.

[40] J. Ian Munro. Tables. *Proceedings of the Conference on Foundations of Software Technology and Theoretical Computer Science*, LNCS 1180: 37–42, 1996.

[41] J. Ian Munro and Venkatesh Raman. Succinct Representation of Balanced Parentheses and Static Trees. *SIAM Journal on Computing*, 31(3): 762–776, 2001.

[42] J. Ian Munro and S. Srinivasa Rao. Succinct Representations of Functions. *Proceedings of the International Colloquium on Automata, Languages and programming*, LNCS 3142, 1006–1015, 2004.

[43] J. Ian Munro, Venkatesh Raman, Rajeev Raman, and S. Srinivasa Rao. Succinct Representations of Permutations. *Proceedings of the International Colloquium on Automata, Languages and Programming*, LNCS 2719: 345–356, 2003.

[44] J. Ian Munro, Venkatesh Raman, and S. Srinivasa Rao. Space Efficient Suffix trees. *Journal of Algorithms* 39(2): 205–222, 2001.

[45] J. Ian Munro, Venkatesh Raman, and Adam Storm. Representing Dynamic Binary Trees Succinctly. *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 529–536, 2001.

[46] Rasmus Pagh. Low Redundancy in Static Dictionaries with Constant Query Time. *SIAM Journal on Computing* 31(2): 353–363, 2001.

[47] Rasmus Pagh and Flemming F. Rodler. Cuckoo Hashing. *Proceedings of the European Symposium on Algorithms*, LNCS 2161: 121–133, 2001.

[48] Jaikumar Radhakrishnan, Venkatesh Raman, and S. Srinivasa Rao. Explicit Deterministic Constructions for Membership in the Bitprobe Model. *Proceedings of the European Symposium on Algorithms*, LNCS 2161: 290–299, 2001.

[49] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct Dynamic Data Structures. *Proceedings of the Workshop on Algorithms and Data Structures*, LNCS 2125, 426–437, 2001.

[50] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees and Multisets. *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 233–242, 2002.

[51] Rajeev Raman and S. Srinivasa Rao. Dynamic Dictionaries and Binary Trees in Near-minimum space. *Proceedings of the International Colloquium on Automata, Languages and Programming*, LNCS 2719: 357–368, 2003.

[52] Venkatesh Raman and S. Srinivasa Rao. Static Dictionaries Supporting Rank. *Pro-*

ceedings of the International Symposium on Algorithms and Computation, LNCS 1741: 18–26, 1999

[53] S. Srinivasa Rao. Time Space Tradeoffs for Compressed Suffix Arrays. *Information Processing Letters*, 82(6): 307–311, 2002.

[54] Kunihiko Sadakane. Succinct Representations of lcp Information and Improvements in the Compressed Suffix Arrays. *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 225-232, 2002.

[55] Jeanette P. Schmidt and Alan Siegel. The Spatial Complexity of Oblivious k-probe Hash Functions. *SIAM Journal on Computing*, 19(5): 775-786, 1990.

[56] Edward Sitarski. Algorithm Alley: HATs: Hashed Array Tables. *Dr. Dobb's Journal*, 21(11), September 1996.

[57] Gyorgy Turán. Succinct Representations of Graphs. *Discrete Applied Mathematics*, 8: 289–294, 1984.

[58] Peter Weiner. Linear Pattern Matching Algorithm. *Proceedings of the IEEE Symposium on Switching and Automata Theory*, 1–11, 1973.

[59] J. W. J. Williams. Algorithm 232: Heap Sort. *Communications of the ACM*, 7(6): 347–348, 1964.

38

Randomized Graph Data-Structures for Approximate Shortest Paths

Surender Baswana
Indian Institute of Technology, Delhi

Sandeep Sen
Indian Institute of Technology, Delhi

38.1 Introduction

Let $G = (V, E)$ be an undirected weighted graph on $n = |V|$ vertices and $m = |E|$ edges. Length of a path between two vertices is the sum of the weights of all the edges of the path. The shortest path between a pair of vertices is the path of least length among all possible paths between the two vertices in the graph. The length of the shortest path between two vertices is also called the distance between the two vertices. An α-approximate shortest path between two vertices is a path of length at-most α times the length of the shortest path.

Computing all-pairs exact or approximate distances in G is one of the most fundamental graph algorithmic problem. In this chapter, we present two randomized graph data-structures for all-pairs approximate shortest paths (APASP) problem in static and dynamic environments. Both the data-structures are hierarchical data-structures and their construction involves random sampling of vertices or edges of the given graph.

The first data-structure is a randomized data-structure designed for efficiently computing APASP in a given static graph. In order to answer a distance query in constant time, most of the existing algorithms for APASP problem output a data-structure which is an $n \times n$ matrix that stores the exact/approximate distance between each pair of vertices explicitly. Recently a remarkable data-structure of $o(n^2)$ size has been designed for reporting all-pairs approximate distances in undirected graph. This data-structure is called *approximate distance oracle* because of its ability to answer a distance query in constant time in spite of its sub-quadratic size. We present the details of this novel data-structure and an efficient algorithm to build it.

The second data-structure is a dynamic data-structure designed for efficiently maintaining APASP in a graph that is undergoing deletion of edges. For a given graph $G = (V, E)$ and a distance parameter $d \leq n$, this data-structure provides the first $o(nd)$ update time algorithm for maintaining α-approximate shortest paths for all pairs of vertices separated by distance $\leq d$ in the graph.

38.2 A Randomized Data-Structure for Static APASP : Approximate Distance Oracles

There exist classical algorithms that require $O(mn \log n)$ time for solving all-pairs shortest paths (APSP) problem. There also exist algorithms based on fast matrix multiplication that achieve sub-cubic time. However, there is still no combinatorial algorithm that could achieve $O(n^{3-\epsilon})$ running time for APSP problem. In recent past, many simple combinatorial algorithms have been designed that compute all-pairs approximate shortest paths (APASP) for undirected graphs. These algorithms achieve significant improvement in the running time compared to those designed for APSP, but the distance reported has some additive or/and multiplicative error. An algorithm is said to compute all pairs α-approximate shortest paths, if for each pair of vertices $u, v \in V$, the distance reported is bounded by $\alpha\delta(u, v)$, where $\delta(u, v)$ denotes the actual distance between u and v.

Among all the data-structures and algorithms designed for computing all-pairs approximate shortest paths, the approximate distance oracles are unique in the sense that they achieves simultaneous improvement in running time (sub-cubic) as well as space (sub-quadratic), and still answers any approximate distance query in constant time. For any $k \geq 1$, it takes $O(kmn^{1/k})$ time to compute $(2k-1)$-approximate distance oracle of size $O(kn^{1+1/k})$ that would answer any $(2k-1)$-approximate distance query in $O(k)$ time.

38.2.1 3-Approximate Distance Oracle

For a given undirected graph, storing distance information from each vertex to all the vertices requires $\theta(n^2)$ space. To achieve sub-quadratic space, the following simple idea comes to mind.

$\mathcal{I}$: *From each vertex, if we store distance information to a* small *number of vertices, can we still be able to report distance between any pair of vertices ?*

The above idea can indeed be realized using a simple random sampling technique, but at the expense of reporting approximate, instead of exact, distance as an answer to a distance query. We describe the construction of 3-approximate distance oracle as follows.

1. Let $R \subset V$ be a subset of vertices formed by picking each vertex randomly independently with probability γ (the value of γ will be fixed later on).
2. For each vertex $u \in V$, store the distances to all the vertices of the sample set R.
3. For each vertex $u \in V$, let $p(u)$ be the vertex nearest to u among all the sampled vertices, and let S_u be the set of all the vertices of the graph G that lie closer to u than the vertex $p(u)$. Store the vertices of set S_u along with their distance from u.

For each vertex $u \in V$, storing distance to vertices S_u helps in answering distance query to vertices in locality of u, whereas storing distance from all the vertices of the graph to all

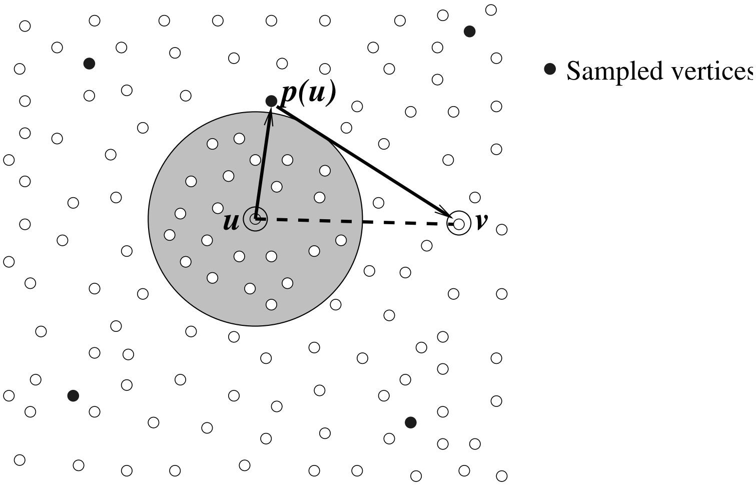


FIGURE 38.1: v is farther to u than $p(u)$, bounding $\delta(p(u), v)$ using triangle inequality.

the sampled vertices will be required (as shown below) to answer distance query for vertices that are not present in locality of each other. In order to extract distance information in constant time, for each vertex $u \in V$, we use two *hash tables* (see [4], chapter 12), for storing distances from u to vertices of sets S_u and R respectively. The size of each hash-table is of the order of the size of corresponding set (S_u or R). A typical hash table would require $O(1)$ expected time to determine whether $w \in S_u$, and if so, report the distance $\delta(u, w)$. In order to achieve $O(1)$ worst case time, the $2 - \mathit{level\ hash\ table}$ (see Fredman, Komlos, Szemeredi, [8]) of optimal size is employed.

The collection of these hash-tables (two tables per vertex) constitute a data-structure that we call approximate distance oracle. Let $u, v \in V$ be any two vertices whose intermediate distance is to be determined approximately. If either u or v belong to set R, we can report exact distance between the two. Otherwise also exact distance $\delta(u, v)$ will be reported if v lies in S_u or vice versa. The only case, that is left, is when neither $v \in S_u$ nor $u \in S_v$. In this case, we report $\delta(u, p(u)) + \delta(v, p(u))$ as approximate distance between u and v. This distance is bounded by $3\delta(u, v)$ as shown below.

$$\begin{aligned}
\delta(u, p(u)) + \delta(v, p(u)) &\leq \delta(u, p(u)) + (\delta(v, u) + \delta(u, p(u))) \ \{\text{using triangle inequality }\} \\
&= 2\delta(u, p(u)) + \delta(u, v) \ \{\text{since graph is undirected }\} \\
&\leq 2\delta(u, v) + \delta(u, v) \\
&\qquad \{\text{since } v \text{ lies farther to } u \text{ than } p(u)\text{, see Figure 38.1}\} \\
&= 3\delta(u, v)
\end{aligned}$$

Hence distance reported by the approximate distance oracle described above is no more than three times the actual distance between the two vertices. In other words, the oracle is a 3-approximate distance oracle. Now, we shall bound the expected size of the oracle. Using linearity of expectation, the expected size of the sample set R is $n\gamma$. Hence storing the distance from each vertex to all the vertices of sample set will take a total of $O(n^2\gamma)$ space. The following lemma gives a bound on the expected size of the sets $S_u, u \in V$.

LEMMA 38.1 Given a graph $G = (V, E)$, let $R \subset V$ be a set formed by picking each vertex independently with probability γ. For a vertex $u \in V$, the expected number of vertices in the set S_u is bounded by $1/\gamma$.

Proof Let $\{v_1, v_2, \cdots, v_{n-1}\}$ be the sequence of vertices of set $V\backslash\{u\}$ arranged in non-decreasing order of their distance from u. The set S_u consists of all those vertices of the set $V\backslash\{u\}$ that lie closer to u than any vertex of set R. Note that the vertex v_i belongs to S_u if none of the vertices of set $\{v_1, v_2, \cdots, v_{i-1}\}$ (i.e., the vertices preceding v_i in the sequence above) is picked in the sample R. Since each vertex is picked independently with probability p, therefore the probability that vertex v_i belongs to set S_u is $(1-\gamma)^{i-1}$. Using linearity of expectation, the expected number of vertices lying closer to u than any sampled vertex is

$$\sum_{i=1}^{n-1}(1-\gamma)^{i-1} \leq \frac{1}{\gamma}$$

Hence the expected number of vertices in the set S_u is no more than $1/\gamma$.

So the total expected size of the 3-approximate distance oracle is $O(n^2\gamma + n/\gamma)$. Choosing $\gamma = 1/\sqrt{n}$ to minimize the size, we conclude that there is a 3-approximate distance oracle of expected size $n^{3/2}$.

38.2.2 Preliminaries

In the previous subsection, 3-approximate distance oracle was presented based on the idea $\mathcal{I}$. The $(2k-1)$-approximate distance oracle is a k-level hierarchical data-structure. An important construct of the data-structure is $Ball(\cdot)$ defined as follows.

DEFINITION 38.1 For a vertex u, and subsets $X, Y \subset V$, the set $Ball(u, X, Y)$ is the set consisting of all those vertices of the set X that lie closer to u than any vertex from set Y. (see Figure 38.2)

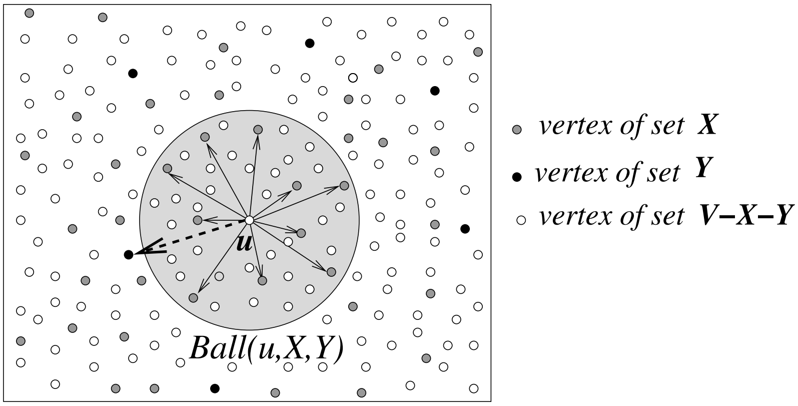


FIGURE 38.2: The vertices pointed by solid-arrows constitute $Ball(u, X, Y)$.

It follows from the definition given above that $Ball(u, X, \emptyset)$ is the set X itself, whereas

$Ball(u, X, X) = \emptyset$. It can also be seen that the 3-approximate distance oracle described in the previous subsection stores $Ball(u, V, R)$ and $Ball(u, R, \emptyset)$ for each vertex $u \in V$.

If the set Y is formed by picking each vertex of set X independently with probability γ, it follows from Lemma 38.1 that the expected size of $Ball(u, X, Y)$ is bounded by $1/\gamma$.

LEMMA 38.2 Let $G = (V, E)$ be a weighted graph, and $X \subset V$ be a set of vertices. If $Y \subset X$ is formed by selecting each vertex independently with probability γ, the expected number of vertices in $Ball(u, X, Y)$ for any vertex $u \in V$ is at-most $1/\gamma$.

38.2.3 $(2k-1)$-Approximate Distance Oracle

In this subsection we shall give the construction of a $(2k-1)$-approximate distance oracle which is also based on the idea $\mathcal{I}$, and can be viewed as a generalization of 3-approximate distance oracle.

The $(2k-1)$-approximate distance oracle is obtained as follows.

1. Let $\mathcal{R}_k^1 \supset \mathcal{R}_k^2 \supset \cdots \mathcal{R}_k^k$ be a a hierarchy of subsets of vertices with $\mathcal{R}_k^1 = V$, and $\mathcal{R}_k^i, i > 1$ is formed by selecting each vertex of set $\mathcal{R}_k^{i-1}$ independently with probability $n^{-1/k}$.
2. For each vertex $u \in V$, store the distance from u to all the vertices of $Ball(u, \mathcal{R}_k^k, \emptyset)$ in a hash table.
3. For each $u \in V$ and each $i < k$, store the vertices of $Ball(u, R_k^i, R_k^{i+1})$ along with their distance from u in a hash table.
   For sake of conciseness and without causing any ambiguity in notations, henceforth we shall use $Ball^i(u)$ to denote $Ball(u, R_k^i, R_k^{i+1})$ or the corresponding hash-table storing $Ball(u, R_k^i, R_k^{i+1})$ for $i < k$.

The collection of the hash-tables $Ball^i(u) : u \in V, i \leq k$ constitute the data-structure that will facilitate answering of any approximate distance query in constant time. To provide a better insight into the data-structure, Figure 38.3 depicts the set of vertices constituting $\{Ball^i(u) | i \leq k\}$.

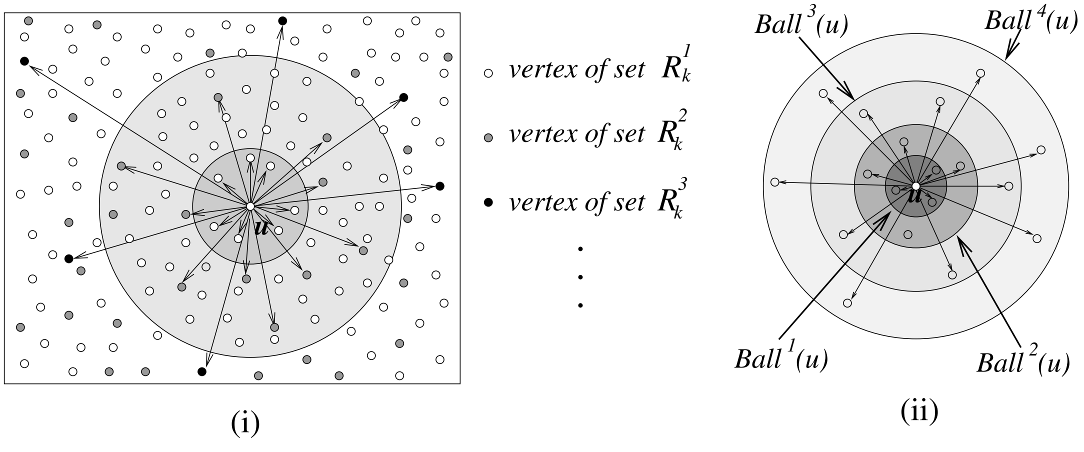


FIGURE 38.3: (i) Close description of $Ball^i(u), i < k$, (ii) hierarchy of balls around u.

Reporting distance with stretch at-most $(2k-1)$

Given any two vertices $u, v \in V$ whose intermediate distance has to be determined approximately. We shall now present the procedure to find approximate distance between the two vertices using the k-level data-structure described above.

Let $p^1(u) = u$ and let $p^i(u), i > 1$ be the vertex from the set $\mathcal{R}_k^i$ nearest to u. Since $p^i(u) \in Ball^i(u)$ for each $u \in V$, so distance from each u to $p^i(u)$ is known for each $i \leq k$.

The query answering process performs at-most k search steps. In the first step, we search $Ball^1(u)$ for the vertex $p^1(v)$. If $p^1(v)$ is not present in $Ball^1(u)$, we move to the next level and in the second step we search $Ball^2(v)$ for vertex $p^2(u)$. We proceed in this way querying balls of u and v alternatively : In ith step, we search $Ball^i(x)$ for $p^i(y)$, where $(x = u, y = v)$ if i is odd, and $(x = v, y = u)$ otherwise. The search ends at ith step if $p^i(y)$ belongs to $Ball^i(x)$, and then we report $\delta(x, p^i(y)) + \delta(y, p^i(y)$ as an approximate distance between u and v.

Distance_Report(u, v)

Algorithm for reporting $(2k-1)$*-approximate distance between* $u, v \in V$

```
$l \leftarrow 1$,
$x \leftarrow u$,$y \leftarrow v$
While $\left(p^l(y) \notin Ball^l(x)\right)$ do
        swap$(x, y)$,
        $l \leftarrow l + 1$
return $\delta(y, p^l(y)) + \delta(x, p^l(y))$
```

Note that $p^k(y) \in \mathcal{R}_k^k$, and we store the distance from x to all the vertices of set $\mathcal{R}_k^k$ in $Ball^k(x)$ (which is $Ball(x, \mathcal{R}_k^k, \emptyset)$. Therefore, the "while loop" of the distance reporting algorithm will execute at-most $k-1$ iterations, spending $O(1)$ time querying a hash table in each iteration.

In order to ensure that the above algorithm reports $(2k-1)$-approximate distance between $u, v \in V$, we first show that the following assertion holds :

$\mathcal{A}_i$: At the end of ith iteration of the *"while loop"*, $\delta(y, p^{i+1}(y)) \leq i\delta(u, v)$.

The assertion $\mathcal{A}_i$ can be proved using induction on i as follows. First note that the variables x and y take the values u and v alternatively during the *"while-loop"*. So $\delta(x, y) = \delta(u, v)$ always.

For the base case $(i = 0)$, $p^1(y)$ is same as y, and y is v. So $\delta(y, p^1(y)) = 0$. Hence $\mathcal{A}_0$ is true. For the rest of the inductive proof, it suffices to show that if $\mathcal{A}_j$ is true, then after $(j+1)$th iteration $\mathcal{A}_{j+1}$ is also true. The proof is as follows.

We consider the case of "even j", the arguments for the case of 'odd j' are similar. For even j, at the end of jth iteration, $\{x = u, y = v\}$, Thus $\mathcal{A}_j$ implies that at the end of jth iteration $\delta(v, p^{j+1}(v)) \leq j\delta(u, v)$. Consider the $(j+1)$th iteration. For the execution of $(j+1)$th iteration, the condition in the *'while-loop'* must have been true. Thus $p^{j+1}(v)$ does not belong to $Ball(u, \mathcal{R}_k^{j+1}, \mathcal{R}_k^{j+2})$. Hence by Definition 38.1, the vertex $p^{j+2}(u)$ must be lying closer to u than the vertex $p^{j+1}(v)$. So at the end of $(j+1)$th iteration, $\delta(y, p^{j+2}(y))$

can be bounded as follows

$$\begin{aligned}
\delta(y, p^{j+2}(y)) &= \delta(u, p^{j+2}(u)) \\
&\leq \delta(u, p^{j+1}(v)) \\
&\leq \delta(u, v) + \delta(v, p^{j+1}(v)) \quad \{\text{using triangle inequality}\} \\
&\leq \delta(u, v) + j\delta(u, v) \quad \{\text{using } \mathcal{A}_j \ \} \\
&= (j+1)\delta(u, v)
\end{aligned}$$

Thus the assertion $\mathcal{A}_{j+1}$ holds.

THEOREM 38.1 *The algorithm* Distance_Report(u, v) *reports* $(2k-1)$*-approximate distance between* u *and* v

Proof As an approximate distance between u and v, note that the algorithm *Distance-Report*(u, v) would output $\delta(y, p^l(y)) + \delta(x, p^l(y))$, which by triangle inequality is no more than $2\delta(y, p^l(y)) + \delta(x, y)$. Since $\delta(x, y) = \delta(u, v)$, and $\delta(y, p^l(y)) \leq (l-1)\delta(u, v)$ as follows from assertion $\mathcal{A}_l$. Therefore, the distance reported is no more than $(2l-1)\delta(u, v)$. Since the "while loop" will execute at-most $k-1$ iterations, so $l = k$, and therefore the distance reported by the oracle is at-most $(2k-1)\delta(u, v)$.

Size of the $(2k-1)$-approximate distance oracle

The expected size of the set R_k^k is $O(n^{1/k})$, and the expected size of each $Ball^i(u)$ is $n^{1/k}$ using Lemma 38.2. So the expected size of the $(2k-1)$-approximate distance oracle is $O(n^{1/k} \cdot n + (k-1) \cdot n \cdot n^{1/k}) = O(kn^{1+1/k})$.

38.2.4 Computing Approximate Distance Oracles

In this subsection, a sub-cubic running time algorithm is presented for computing $(2k-1)$-approximate distance oracles. It follows from the description of the data-structure associated with approximate distance oracle that after forming the sampled sets of vertices $\mathcal{R}_k^i$, that takes $O(m)$ time, all that is required is the computation of $Ball^i(u)$ along with the distance from u to the vertices belonging to these balls for each u and $i \leq k$.

Since $Ball^i(u)$ is the set of all the vertices of set R_k^i that lie closer to u than the vertex $p^{i+1}(u)$. So, in order to compute $Ball^i(u)$, first we compute $p^i(u)$ for all $u \in V, i \leq k$.

Computing $\mathbf{p}^i(u), \forall u \in V$

Recall from definition itself that $p^i(u)$ is the vertex of the set R_k^i that is nearest to u. Hence, computing $p^i(u)$ for each $u \in V$ requires solving the following problem with $X = R_k^i, Y = V \backslash X$.

Given $X, Y \subset V$ *in a graph* $G = (V, E)$*, with* $X \cap Y = \emptyset$*, compute the nearest vertex of set* X *for each vertex* $y \in Y$*.*

The above problem can be solved by running a single source shortest path algorithm (Dijkstra's algorithm) on a modified graph as follows. Modify the original graph G by adding a dummy vertex s to the set V, and joining it to each vertex of the set X by an edge of zero weight. Let G' be the modified graph. Running Dijkstra's algorithm from the vertex s as the source, it can be seen that the distance from s to a vertex $y \in Y$ is indeed the distance from y to the nearest vertex of set X. Moreover, if $e(s, x), x \in X$ is the edge leading to the shortest path from s to y, then x is the vertex from the set X that lies nearest to y. The running time of the Dijkstra's algorithm is $O(m \log n)$, we can thus state the following lemma.

LEMMA 38.3 Given $X, Y \subset V$ in a graph $G = (V, E)$, with $X \cap Y = \emptyset$, it takes $O(m \log n)$ to compute the nearest vertex of set X for each vertex $y \in Y$.

COROLLARY 38.1 Given a weighted undirected graph $G = (V, E)$, and a hierarchy of subsets $\{\mathcal{R}_k^i | i \leq k\}$, we can compute $p^i(u)$ for all $i \leq k, u \in V$ in $O(km \log n)$ time

Computing $\mathbf{Ball^i(u)}$ efficiently

In order to compute $Ball^i(u)$ for each vertex $u \in V$ efficiently, we first compute clusters $\{C(v, \mathcal{R}_k^{i+1}) | v \in \mathcal{R}_k^i\}$ which are defined as follows :

DEFINITION 38.2 For a graph $G = (V, E)$, and a set $X \subset V$, the cluster $C(v, X)$ consists of each vertex $w \in V$ for whom v lies closer than any vertex of set X. That is, $\delta(w, v) < \delta(w, x)$ for each $x \in X$.

It follows from the definition given above that $u \in C(v, \mathcal{R}_k^{i+1})$ if and only if $v \in Ball^i(u)$. So, given clusters $\{C(v, \mathcal{R}_k^{i+1}) | v \in \mathcal{R}_k^i\}$, we can compute $\{Ball^i(u) : u \in V\}$ as follows.

```
For each $v \in \mathcal{R}_k^i$ do
        For each $u \in C(v, \mathcal{R}_k^{i+1})$ do
                $Ball^i(u) \longleftarrow Ball^i(u) \cup \{v\}$
```

Hence we can state the following Lemma.

LEMMA 38.4 Given the family of clusters $\{C(v, \mathcal{R}_k^{i+1}) | v \in \mathcal{R}_k^i\}$, the time required to compute $\{Ball^i(u)\}$ is bounded by $O(\sum_{u \in V} |Ball^i(u)|)$.

The following property of the cluster $C(v, \mathcal{R}_k^{i+1})$ will be used in its efficient computation.

LEMMA 38.5 If $u \in C(v, \mathcal{R}_k^{i+1})$, then all the vertices on the shortest path from v to u also belong to the set $C(v, \mathcal{R}_k^{i+1})$.

Proof We give a proof by contradiction. Given that $u \in C(v, \mathcal{R}_k^{i+1})$, let w be any vertex on the shortest path from v to u. If $w \notin C(v, \mathcal{R}_k^{i+1})$, the vertex v doesn't lie closer to w than the vertex $p^{i+1}(w)$. See Figure 38.4. In other words $\delta(w, v) \geq \delta(w, p^{i+1}(w))$. Hence

$$\delta(u, v) = \delta(u, w) + \delta(w, v) \geq \delta(u, w) + \delta(w, p^{i+1}(w)) \geq \delta(u, p^{i+1}(w))$$

Thus v does not lie closer to u than $p^{i+1}(w)$ which is a vertex of set $\mathcal{R}_k^{i+1}$. Hence by definition, $u \notin C(v, \mathcal{R}_k^{i+1})$, thus a contradiction.

From Lemma 38.5, it follows that the graph induced by the vertices of the cluster $C(v, \mathcal{R}_k^{i+1})$ is connected (hence the name cluster). Moreover, the entire cluster $C(v, \mathcal{R}_k^{i+1})$

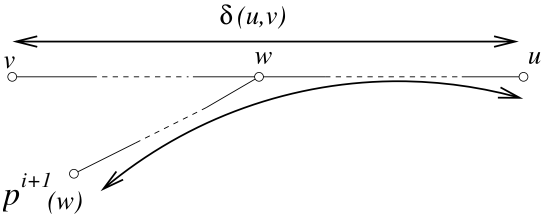


FIGURE 38.4: if w does not lie in $C(v, \mathcal{R}_k^{i+1})$, then $p^{i+1}(w)$ would lie closer to u than v.

appears as a sub-tree of the shortest path tree rooted at v in the graph. As follows from the definition, for each vertex $x \in C(v, \mathcal{R}_k^{i+1})$, $\delta(v,x) < \delta(x, p^{i+1}(x))$. Based on these two observations, here follows an efficient algorithm that computes the set $C(v, \mathcal{R}_k^{i+1})$. The algorithm performs a *restricted* Dijkstra's algorithm from the vertex v, wherein we don't proceed along any vertex that does not belong to the set $C(v, \mathcal{R}_k^{i+1})$.

A restricted Dijkstra's algorithm : Note that the Dijkstra's algorithm starts with singleton tree $\{v\}$ and performs $n-1$ steps to grow the complete shortest path tree. Each vertex $x \in V \backslash \{v\}$ is assigned a label $L(x)$, which is infinity in the beginning, but eventually becomes the distance from v to x. Let V_i denotes the set of i nearest vertices from v. The algorithm maintains the following invariant at the end of lth step :

$\mathcal{I}(l)$: For all the vertices of the set V_l, the label $L(x) = \delta(v,x)$, and for every other vertex $y \in V \backslash V_l$, the label $L(y)$ is equal to the length of the shortest path from v to y that passes through vertices of V_l only.

During the $(j+1)$th step, we select the vertex, say w from set $V - V_j$ with least value of $L(\cdot)$. Since all the edge weights are positive, it follows from the invariant $\mathcal{I}(j)$ that $L(w) = \delta(w,v)$. Thus we add w to set V_j to get the set V_{j+1}. Now in order to satisfy the invariant $\mathcal{I}(j+1)$, we relax each edge $e(w,y)$ incident from w to a vertex $y \in V - V_{j+1}$ as follows : $L(y) \leftarrow \min\{L(y), L(w) + weight(w,y)\}$. It is easy to observe that this ensures the validity of the invariant $\mathcal{I}(j+1)$.

In the restricted Dijkstra's algorithm, we will put the following restriction on relaxation of an edge $e(w,y)$: we relax the edge $e(w,y)$ only if $L(w) + weight(w,y)$ is less than $\delta(y, p^i(y))$. This will ensure that a vertex $y \notin C(v, \mathcal{R}_k^{i+1})$ will never be visited during the algorithm. The fact that the vertices of the cluster $C(v, \mathcal{R}_k^{i+1})$ form a sub-tree of the shortest path tree rooted at v, ensures that the above restricted Dijkstra's algorithm indeed finds all the vertices (along with their distance from v) that form the cluster $C(v, \mathcal{R}_k^{i+1})$. Since the running time of Dijkstra's algorithm is dominated by the number of edges relaxed, and each edge relaxation takes $\log(n)$ time only, therefore, the restricted Dijkstra's algorithm will run in time of the order of $\sum_{x \in C(v, \mathcal{R}_k^{i+1})} degree(x) \log n$. Thus the total time for computing all the clusters $\{C(v, \mathcal{R}_k^{i+1}) | v \in \mathcal{R}_k^i\}$ is given by :

$$\begin{aligned} \sum_{v \in \mathcal{R}_k^i, x \in C(v, \mathcal{R}_k^{i+1})} degree(x) \log n &= \left(\sum_{x \in V, v \in Ball^i(x)} degree(x) \right) \log n \\ &= \left(\sum_{x \in V} |Ball^i(x)| \cdot degree(x) \right) \log n \end{aligned}$$

By Lemma 38.2, the expected size of $Ball^i(x)$ is bounded by $n^{1/k}$, hence using linearity of expectation, the total expected cost of computing $\{C(v, \mathcal{R}_k^{i+1}) | v \in \mathcal{R}_k^i\}$ is asymptotically bounded by

$$\sum_{x \in V} n^{1/k} \cdot degree(x) \log n = 2mn^{1/k} \log n$$

Using the above result and Lemma 38.4, we can thus conclude that for a given weighted graph $G = (V, E)$ and an integer k, it takes a total of $\tilde{O}(kmn^{1/k} \log n)$ time for computing $\{Ball^i(u) | i < k, u \in V\}$. If we use Fibonacci heaps instead of binary heaps in implementation of the restricted Dijkstra's algorithm, we can get rid of the logarithmic factor in the running time. Hence the total expected running time for building the data-structure is $O(kmn^{1/k})$. As mentioned before, the expected size of the data-structure will be $O(kn^{1+1/k})$. To get $O(kn^{1+1/k})$ bound on the worst case size of the data-structure, we repeat the preprocessing algorithm. The expected number of iterations will be just a constant. Hence, we can state the following theorem.

THEOREM 38.2 *Given a weighted undirected graph $G = (V, E)$ and an integer k, a data-structure of size $O(kn^{1+1/k})$ can be built in $O(kmn^{1/k})$ expected time so that given any pair of vertices, $(2k - 1)$-approximate distance between them can be reported in $O(k)$ time.*

38.3 A Randomized Data-Structure for Decremental APASP

There are a number of applications that require efficient solutions of the APASP problem for a dynamic graph. In these applications, an initial graph is given, followed by an on-line sequence of queries interspersed with updates that can be insertion or deletion of edges. We have to carry out the updates and answer the queries on-line in an efficient manner. The goal of a dynamic graph algorithm is to update the solution efficiently after the dynamic changes, rather than having to re-compute it from scratch each time.

The approximate distance oracles described in the previous section can be used for answering approximate distance query in a static graph. However, there does not seem to be any efficient way to dynamize these oracles in order to answer distance queries in a graph under deletion of edges. In this section we shall describe a hierarchical data structure for efficiently maintaining APASP in an undirected unweighted graph under deletion of edges. In addition to maintaining approximate shortest paths for all-pairs of vertices, this scheme has been used for efficiently maintaining approximate shortest paths for pair of vertices separated by distance in an interval $[a, b]$ for any $1 \leq a < b \leq n$. However, to avoid giving too much detail in this chapter, we would outline an efficient algorithm for the following problem only.

APASP-d *: Given an undirected unweighted graph $G = (V, E)$ that is undergoing deletion of edges, and a distance parameter $d \leq n$, maintain approximate shortest paths for all-pairs of vertices separated by distance at-most d.*

38.3.1 Main Idea

For an undirected unweighted graph $G = (V, E)$, a breadth-first-search (BFS) tree rooted at a vertex $u \in V$ stores distance information with respect to the vertex u. So in order to maintain shortest paths for all-pairs of vertices separated by distance $\leq d$, it suffices to maintain a BFS tree of depth d rooted at each vertex under deletion of edges. This is the approach taken by the previously existing algorithms.

The main idea underlying the hierarchical data-structure that would provide efficient update time for maintaining APASP can be summarized as follows : Instead of maintaining exact distance information separately from each vertex, keep *small* BFS trees around each vertex for maintaining distance information within locality of each vertex, and some what *larger* BFS trees around *fewer* vertices for maintaining global distance information.

We now provide the underlying intuition of the above idea and a brief outline of the new techniques used.

Let B_u^d denote the BFS tree of depth d rooted at vertex $u \in V$. There exists a simple algorithm for maintaining a BFS tree B_u^d under deletion of edges that takes a total of $\mu(B_u^d) \cdot d$ time, where $\mu(t)$ is the number of edges in the graph induced by tree t. Thus the total update time for maintaining shortest path for all-pairs separated by distance at-most d is of the order of $\sum_{u \in V} \mu(B_u^d) \cdot d$. Potentially $\mu(B_u^d)$ can be as large as $\theta(m)$, and so the total update time over any sequence of edge deletions will be $O(mnd)$. Dividing this total update cost uniformly over the entire sequence of edge deletions, we can see that it takes $O(nd)$ amortized update time per edge deletion, and $O(1)$ time for reporting exact distance between any pair of vertices separated by distance at-most d.

In order to achieve $o(nd)$ bound on the update time for the problem APASP-d, we closely look at the expression of total update time $\sum_{u \in V} \mu(B_u^d) \cdot d$. There are n terms in this expression each of potential size $\theta(m)$. A decrease in either the total number of terms or the size of each term would give an improvement in the total update time. Thus the following simple ideas come to mind.

- Is it possible to solve the problem APASP-d by keeping very *few* depth-d BFS trees ?
- Is there some other alternative t for depth bounded BFS tree B_u^d that has $o(m)$ bound on $\mu(t)$?

While it appears difficult for any of the above ideas to succeed individually, they can be combined in the following way: *Build and maintain BFS trees of depth* $2d$ *on vertices of a set* $S \subset V$ *of size* $o(n)$*, called* the set of special vertices*, and for each remaining vertex* $u \in V \backslash S$*, maintain a BFS tree (denoted by* B_u^S*) rooted at* u *and containing all the vertices that lie closer to* u *than the nearest special vertex, say* $\mathcal{N}(u, S)$.

Along the above lines, we present a 2-level data-structure (and its generalization to k-levels) for the problem APASP-d.

It can be seen that unlike the tree B_u^d, the new BFS tree B_u^S might not contain all the vertices lying within distance d from u. In order to ensure that our scheme leads to a solution of problem APASP-d, we use the following observation similar to that of 3-approximate distance oracle in the previous section. If v is a vertex lying within distance d from u but not present in B_u^S, an *approximate* distance from u to v can be extracted from the tree rooted at the nearest special vertex $\mathcal{N}(u, S)$. This is because (by triangle inequality) the distance from $\mathcal{N}(u, S)$ to v is at most twice the distance from u to v.

For our hierarchical scheme to lead to improved update time, it is crucial that we establish sub-linear upper bounds on $\mu(B_u^S)$. We show that if the set S is formed by picking each vertex independently with *suitable* probability, then $\mu(B_u^S) = \tilde{O}(m/|S|)$ with probability arbitrarily close to 1.

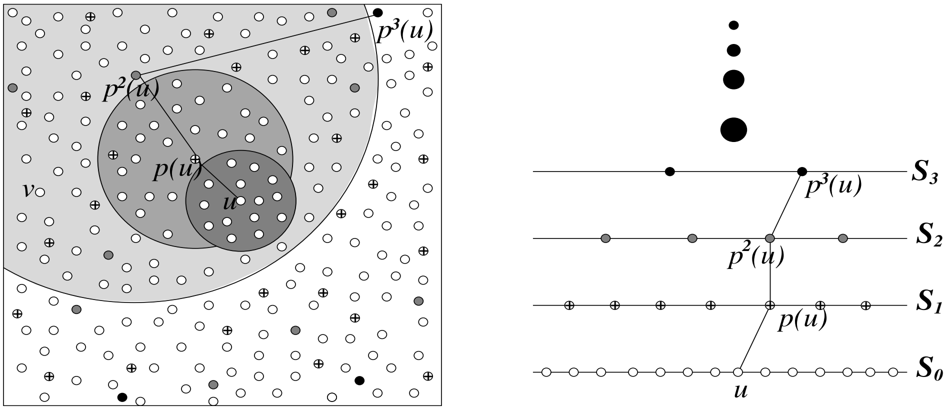


FIGURE 38.5: Hierarchical scheme for maintaining approximate distance.

38.3.2 Notations

For an undirected unweighted graph $G = (V, E)$, $S \subset V$, and a distance parameter $d \le n$,

- $\delta(u, v)$: distance between u and v.
- $\mathcal{N}(v, S)$: the vertex of the set $S \subset V$ nearest to v.
- B_v^d : The BFS tree of depth d rooted at $v \in V$.
- B_v^S : The BFS tree of depth $(\delta(u, \mathcal{N}(u, S)) - 1)$ rooted at v.
- $B_v^{d,S}$: The BFS tree of depth $\min\{d, \delta(v, \mathcal{N}(v, S)) - 1\}$ rooted at v.
- $\mu(t)$: the number of edges in the sub-graph (of G) induced by the tree t.
- $\nu(t)$: the number of vertices in tree t.
- For a sequence $\{S_0, S_1, \cdots S_{k-1}\}, S_i \subset V$, and a vertex $u \in S_0$, we define
  $p^0(u) = u$.
  $p^{i+1}(u) =$ the vertex from set S_{i+1} nearest to $p^i(u)$.
- $\overline{\alpha}$: the smallest integer of the form 2^i which is greater than α.

38.3.3 Hierarchical Distance Maintaining Data-Structure

Based on the idea of "keeping *many small trees, and a few large trees*", we define a k-level hierarchical data-structure for efficiently maintaining approximate distance information as follows. (See Figure 38.5)

Let $\mathcal{S} = \{S_0, S_1, \cdots, S_{k-1} : S_i \subset V, |S_{i+1}| < |S_i|\}$ be a sequence. For a given distance parameter $d \le n$ and $i < k - 1$, let $\mathcal{F}_i$ be the collection $\{B_u^{2^i d, S_{i+1}} : u \in S_i\}$ of BFS trees, and $\mathcal{F}_{k-1}$ be the collection of BFS trees of depth $2^{k-1}d$ rooted at each $u \in S_{k-1}$. We shall denote the set $\{(S_0, \mathcal{F}_0), (S_1, \mathcal{F}_1), \cdots, (S_{k-1}, \mathcal{F}_{k-1})\}$ as the k-level hierarchy $\mathcal{H}_d^k$ induced by the sequence $\mathcal{S}$.

Let v be a vertex within distance d from u. If v is present in B_u^{d,S_1}, we can report exact distance between them. Otherwise, (as will soon become clear) we can extract the approximate distance between u and v from the collection of the BFS trees rooted at the vertices $u, p(u), \cdots, p^{k-1}(u)$ (see Figure 38.5). The following Lemma is the basis for estimating the distance between two vertices using the hierarchy $\mathcal{H}_d^k$.

LEMMA 38.6 Given a hierarchy $\mathcal{H}_d^k$, if $j < k - 1$ is such that v is not present in any of

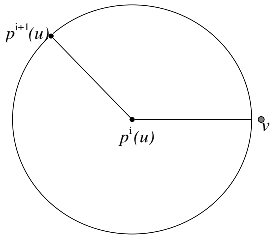


FIGURE 38.6: Bounding the approximate distance between $p^{i+1}(u)$ and v.

the BFS trees $\{B^{2^i d, S_{i+1}}_{p^i(u)} | 0 \le i \le j\}$, then for all $i \le j$

$$\delta(p^{i+1}(u), p^i(u)) \le 2^i \delta(u,v) \quad \text{and} \quad \delta(p^{i+1}(u), v) \le 2^{i+1}\delta(u,v).$$

Proof We give a proof by induction on j.
Base Case $(j = 0)$ **:** Since v is not present in $B^{S_1}_u$, so the vertex $p(u)$ must be lying equidistant or closer to u than v. Hence $\delta(p(u), u) \le \delta(u, v)$. Using triangle inequality, it follows that $\delta(p(u), v) \le \delta(p(u), u) + \delta(u, v) = 2\delta(u, v)$.
Induction Hypothesis :
$\delta(p^{i+1}(u), p^i(u)) \le 2^i\delta(u, v)$, and
$\delta(p^{i+1}(u), v) \le 2^{i+1}\delta(u, v)$, for all $i < l$.
Induction Step $(j = l)$ **:** if $v \notin B^{S_{l+1}}_{p^l(u)}$, then the distance between p^{l+1} and $p^l(u)$ must not be longer than $\delta(p^l(u), v)$, which is less than $2^l\delta(u, v)$ (using induction hypothesis).

Now using triangle inequality (see the Figure 38.6) we can bound $\delta(p^{l+1}(u), v)$ as follows.

$$\begin{aligned}\delta(p^{l+1}(u), v) &\le \delta(p^{l+1}(u), p^l(u)) + \delta(p^l(u), v)\\ &\le 2^l\delta(u, v) + \delta(p^l(u), v)\\ &\le 2^l\delta(u, v) + 2^l\delta(u, v) \quad \{\text{ using I.H.}\}\\ &= 2^{l+1}\delta(u, v)\end{aligned}$$

Since the depth of a BFS tree at $(k - 1)$th level of hierarchy $\mathcal{H}^k_d$ is $2^{k-1}d$, therefore the following corollary holds true.

COROLLARY 38.2 If $\delta(u, v) \le d$, then there is some $p^i(u), i < k$ such that v is present in the BFS tree rooted at $p^i(u)$ in the hierarchy $\mathcal{H}^k_d$.

LEMMA 38.7 Given a hierarchy $\mathcal{H}^k_d$, if $j < k - 1$ is such that v is not present in any of the BFS trees $\{B^{2^i d, S_i}_{p^i(u)} | 0 \le i \le j\}$, then $\delta(p^{i+1}(u), u) \le (2^{i+1} - 1)\delta(u, v)$, for all $i \le j$.

Proof Using simple triangle inequality, it follows that

$$\begin{aligned}\delta(p^{i+1}(u), u) &\le \sum_{l \le i} \delta(p^{l+1}(u), p^l(u))\\ &\le \sum_{l \le i} 2^l\delta(u, v) = (2^{i+1} - 1)\delta(u, v)\end{aligned}$$

It follows from Lemma 38.6 and Lemma 38.7 that if l is the smallest integer such that v is present in the BFS tree rooted at $p^l(u)$ in the hierarchy $\mathcal{H}_d^k$, then we can report $\delta(p^l(u), u) + \delta(p^l(u), v)$ as an approximate distance between u and v. Along these lines, we shall present an improved decremental algorithms for APASP-d.

38.3.4 Bounding the Size of $\mathbf{B_u^{d,S}}$ under Edge-Deletions

We shall now present a scheme based on random sampling to find a set $S \subset V$ of vertices that will establish a sub-linear bound on the number of vertices ($\nu(B_u^S)$) as well as the number of edges ($\mu(B_u^S)$) induced by B_u^S under deletion of edges. Since $B_u^{d,S} \subset B_u^S$, so these upper bounds also hold for $B_u^{d,S}$.

Build the set S of vertices by picking each vertex from V independently with probability $\frac{n^c}{n}$. The expected size of S is $O(n^c)$. Consider an ordering of vertices V according to their levels in the BFS tree B_u^S (see Figure 38.7). The set of vertices lying at higher levels than the nearest sampled vertex in this ordering is what constitutes the BFS tree B_u^S. Along similar lines as that of Lemma 38.1, it follows that the expected size of this set (and hence $\nu(B_u^S)$) is $\frac{n}{n^c}$. Moreover, it can be shown that $\nu(B_u^S)$ is no more than $\frac{4n \ln n}{n^c}$ with probability $> 1 - \frac{1}{n^4}$. Now as the edges are being deleted, the levels of the vertices in the tree B_u^S may fall, and so the ordering of the vertices may change. There will be a total of m such orderings during the entire course of edge deletions. Since the vertices are picked randomly and independently, therefore, the upper bound of $\frac{4n \ln n}{n^c}$ holds for $\nu(B_u^S)$ with probability $(1 - \frac{1}{n^4})$ for any of these orderings. So we can conclude that $\nu(B_u^S)$, the number of vertices of tree B_u^S never exceeds $(\frac{4n \ln n}{n^c})$ during the entire course of edge deletions with probability $> 1 - \frac{1}{n^2}$.

To bound the number of edges induced by B_u^S, consider the following scheme. Pick every edge independently with probability $\frac{n^c}{m}$. The set S consists of the end points of the sampled edges. The expected size of S is $O(n^c)$. Consider an ordering of the edges according to their level in B_u^S (level of an edge is defined as the minimum of the levels of its end points). Along the lines of arguments given above (for bounding the the number of vertices of B_u^S), it can be shown that $\mu(B_u^S)$, the number of edges induced by B_u^S remains $\leq \frac{4m \ln n}{n^c}$ with probability $> 1 - \frac{1}{n^2}$ during the entire course of edge deletions.

Note that in the sampling scheme to bound the number of vertices of tree B_u^S, a vertex v is picked with probability $\frac{n^c}{n}$. Whereas in the sampling scheme for bounding the number of edges in the sub-graph induced by B_u^S, a vertex v is picked with probability $\frac{degree(v) \cdot n^c}{m}$. It can thus be seen that both the bounds can be achieved simultaneously by the following random sampling scheme :

$\mathcal{R}(c)$: Pick each vertex $v \in V$ independently with probability $\frac{n^c}{n} + \frac{degree(v) \cdot n^c}{m}$.

It is easy to see that the expected size of the set formed by the sampling scheme $\mathcal{R}(c)$ will be $O(n^c)$.

THEOREM 38.3 *Given an undirected unweighted graph $G = (V, E)$, a constant $c < 1$, and a distance parameter d; a set S of size $O(n^c)$ vertices can be found that will ensure the following bound on the number of vertices and number of edges in the sub-graph of G induced by $B_u^{d,S}$.*

$$\nu(B_u^{d,S}) = O\left(\frac{n \ln n}{n^c}\right), \qquad \mu(B_u^{d,S}) = O\left(\frac{m \ln n}{n^c}\right)$$

with probability $\Omega(1 - \frac{1}{n^2})$ during the entire sequence of edge deletions.

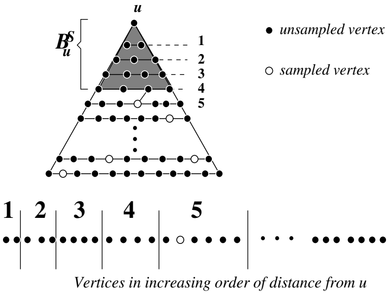


FIGURE 38.7: Bounding the size of BFS tree B_u^S.

Maintaining the BFS tree $\mathbf{B_u^{d,S}}$ under edge deletions

Even and Shiloach [7] design an algorithm for maintaining a depth-d BFS tree in an undirected unweighted graph.

LEMMA 38.8 [Even, Shiloach [7]] Given a graph under deletion of edges, a BFS tree $B_u^d, u \in V$ can be maintained in $O(d)$ amortized time per edge deletion.

For maintaining a $B_u^{d,S}$ tree under edge deletions, we shall use the same algorithm of [7] with the modification that whenever the depth of $B_u^{d,S}$ has to be increased (due to recent edge deletion), we grow the tree to its new level $\min\{d, \delta(u, \mathcal{N}(u,S)) - 1\}$. We analyze the total update time required for maintaining $B_u^{d,S}$ as follows.

There are two computational tasks : one extending the level of the tree, and another that of maintaining the levels of the vertices in the tree $B_u^{d,S}$ under edge deletions. For the first task, the time required is bounded by the edges of the new level introduced which is $O(\mu(B_u^{d,S}))$. For the second task, we give a variant of the proof of Even and Shiloach [7] (for details, please refer [7]). The running time is dominated by the processing of the edges in this process. In-between two consecutive processing of an edge, level of one of the end-points of the edge falls down by at least one unit. The processing cost of an edge can thus be charged to the level from which it has fallen. Clearly the maximum number of edges passing a level i is bounded by $\mu(B_u^{d,S})$. The number of levels in the tree $B_u^{d,S}$ is $\min\{d, \nu(B_u^{d,S})\}$. Thus the total cost for maintaining the BFS tree $B_u^{d,S}$ over the entire sequence of edge deletions is $O(\mu(B_u^{d,S}) \cdot \min\{d, \nu(B_u^{d,S})\})$.

LEMMA 38.9 Given an undirected unweighted graph $G = (V, E)$ under edge deletions, a distance parameter d, and a set $S \subset V$; a BFS tree $B_u^{d,S}$ can be maintained in

$$O\left(\frac{\mu(B_u^{d,S})}{m} \cdot \min\{d, \nu(B_u^{d,S})\}\right)$$

amortized update time per edge deletion.

Some technical details

As the edges are being deleted, we need an efficient mechanism to detect any increase in the depth of tree $B_u^{d,S}$. We outline one such mechanism as follows.

For every vertex $v \notin S$, we keep a count $C[v]$ of the vertices of the S that are neighbors of v. It is easy to maintain $C[u], \forall u \in V$ under edge-deletions. We use the count $C[v]$ in order to detect any increase in the depth of a tree $B_u^{d,S}$ as follows. Note that when depth of a tree $B_u^{d,S}$ is less than d, there has to be at-least one vertex w at leaf-level in $B_u^{d,S}$ with $C[w] \geq 1$ (as an indicator that the vertex $p(u)$ is at next level). Therefore, after an edge deletion if there is no vertex w at leaf level with $C[w] \geq 1$, we grow the BFS tree $B_u^{d,S}$ beyond its previous level until either depth becomes d or we reach some vertex w' with $C[w'] \geq 1$.

Another technical issue is that when an edge $e(x, y)$ is deleted, we must update only those trees which contain x and y. For this purpose, we maintain for each vertex, a set of roots of all the BFS trees containing it. We maintain this set using any dynamic search tree.

38.3.5 Improved Decremental Algorithm for APASP up to Distance d

Let $\{(S_0, \mathcal{F}_0), (S_1, \mathcal{F}_1), \cdots, (S_{k-1}, \mathcal{F}_{k-1})\}$ be a k-level hierarchy $\mathcal{H}_d^k$ with $S_0 = V$ and $n^{c_i} = |S_i|$, where each $c_i, i < k$ is a fraction to be specified soon. Each set $S_i, i > 0$ is formed by picking the vertices from set V using the random sampling scheme $\mathcal{R}$ mentioned in the previous subsection.

To report distance from u to v, we start form the level 0. We first inquire if v lies in B_u^{d,S_1}. If v does not lie in the tree, we move to the first level and inquire if v lies in $B_{p(u)}^{2d,S_2}$. It follows from the Corollary 38.2 that if $\delta(u, v) \leq d$, then proceeding in this way, we eventually find a vertex $p^l(u), l \leq k-1$ in the hierarchy $\mathcal{H}_d^k$ such that v is present in the BFS tree rooted at $p^l(u)$. (See Figure 38.5). We then report the sum of distances from $p^l(u)$ to both u and v.

Algorithm for reporting approximate distance using $\mathcal{H}_d^k$

```
Distance($u, v$)
{  $D \longleftarrow 0; l \longleftarrow 0$
  While ($v \notin B_{p^l(u)}^{2^l d, S_{l+1}} \wedge l < k-1$) do
  {
   If $u \in B_{p^l(u)}^{2^l d, S_{l+1}}$, then $D \leftarrow \delta(p^l(u), u)$,
   $D \leftarrow D + \delta(p^l(u), p^{l+1}(u))$
   $l \leftarrow l + 1$;
  }
  If $v \notin B_{p^l(u)}^{2^l d, S_{l+1}}$, then "$\delta(u, v)$ is greater than $d$",
                    else return $\delta(p^l(u), v) + D$
}
```

The approximation factor ensured by the above algorithm can be bounded as follows.

It follows from the Lemma 38.7 that the final value of D in the algorithm given above is bounded by $(2^l - 1)\delta(u, v)$, and it follows from Lemma 38.6 that $\delta(p^l(u), v)$ is bounded by $2^l \delta(u, v)$. Since $l \leq k-1$, therefore the distance reported by the algorithm is bounded by $(2^k - 1)\delta(u, v)$ if v is at distance $\leq d$.

LEMMA 38.10 Given an undirected unweighted graph $G = (V, E)$, and a distance parameter d. If α is the desired approximation factor, then there exists a hierarchical scheme $\mathcal{H}_d^k$ with $k = \log_2 \overline{\alpha}$, that can report α-approximate shortest distance between any two vertices separated by distance $\leq d$, in time $O(k)$.

Update time for maintaining the hierarchy $\mathcal{H}_{\mathbf{k}}^{\mathbf{d}}$: The update time per edge deletion for maintaining the hierarchy $\mathcal{H}_k^d$ is the sum total of the update time for maintaining the set of BFS trees $\mathcal{F}_i, i \leq k-1$.

Each BFS tree from the set $\mathcal{F}_{k-1}$ has depth $2^{k-1}d$, and edges $O(m)$. Therefore, using Lemma 38.8, each tree from set $\mathcal{F}_{k-1}$ requires $O(2^{k-1}d)$ amortized update time per edge deletion. So, the amortized update time T_{k-1} per edge deletion for maintaining the set $\mathcal{F}_{k-1}$ is

$$T_{k-1} = O(n^{c_{k-1}} 2^{k-1} d)$$

It follows from the Theorem 38.3 that a tree t from a set $\mathcal{F}_i, i < (k-1)$, has $\mu(t) = m \ln n / n^{c_{i+1}}$, and depth $= \min\{2^i d, n \ln n / n^{c_{i+1}}\}$. Therefore, using the Lemma 38.9, each tree $t \in \mathcal{F}_i, i < k-1$ requires $O(\min\{2^i d / n^{c_{i+1}}, n \ln n / n^{2c_{i+1}}\})$ amortized update time per edge deletion. So the amortized update time T_i per edge deletion for maintaining the set $\mathcal{F}_i$ is

$$T_i = O\left(\min\left\{2^i d \frac{n^{c_i}}{n^{c_{i+1}}} \ln n, \frac{n^{1+c_i}}{n^{2c_{i+1}}} \ln^2 n\right\}\right), \quad i < k-1$$

Hence, the amortized update time T per edge deletion for maintaining the hierarchy $\mathcal{H}_k^d$ is

$$\begin{aligned} T &= T_{k-1} + \sum_{i<k-1} T_i \\ &= O(n^{c_{k-1}} 2^{k-1} d) + \sum_{i=0}^{i=k-2} O\left(\min\left\{2^i d \frac{n^{c_i}}{n^{c_{i+1}}} \ln n, \frac{n^{1+c_i}}{n^{2c_{i+1}}} \ln^2 n\right\}\right) \end{aligned}$$

To minimize the sum on right hand side in the above equation, we balance all the terms constituting the sum, and get

$$T = \tilde{O}\left(2^{k-1} \cdot \min\left\{\sqrt[k]{n}d, (nd)^{\frac{2^{(k-1)}}{2^k-1}}\right\}\right)$$

If α is the desired approximation factor, then it follows from Lemma 38.10 that the number of levels k, in the hierarchy are $\log_2 \overline{\alpha}$. So the amortized update time required is $\tilde{O}(\alpha \cdot \min\{\sqrt[\log_2 \overline{\alpha}]{n}d, (nd)^{\frac{\overline{\alpha}}{2(\overline{\alpha}-1)}}\})$.

THEOREM 38.4 *Let $G = (V, E)$ be an undirected unweighted graph undergoing edge deletions, d be a distance parameter, and $\alpha > 2$ be the desired approximation factor. There exists a data-structure $\dot{\mathcal{D}}_\alpha(1, d)$ for maintaining α-approximate distances for all-pairs separated by distance $\leq d$ in $\tilde{O}(\alpha \cdot \min\{\sqrt[\log_2 \overline{\alpha}]{n}d, (nd)^{\frac{\overline{\alpha}}{2(\overline{\alpha}-1)}}\})$ amortized update time per edge deletion, and $O(\log \overline{\alpha})$ query time.*

Based on the data-structure of [7], the previous best algorithm for maintaining all-pairs exact shortest paths of length $\leq d$ requires $O(nd)$ amortized update time. We have been able to achieve $o(nd)$ update time at the expense of introducing approximation as shown in Table 38.1 on the following page.

38.4 Further Reading and Bibliography

Zwick [10] presents a very recent and comprehensive survey on the existing algorithms for all-pairs approximate/exact shortest paths. Based on the fastest known matrix multiplication

Data-structure	α (the approximation factor)	Amortized update time per edge deletion
$\dot{\mathcal{D}}_3(1,d)$	3	$\tilde{O}(\min(\sqrt{n}d, (nd)^{2/3}))$
$\dot{\mathcal{D}}_7(1,d)$	7	$\tilde{O}(\min(\sqrt[3]{n}d, (nd)^{4/7}))$
$\dot{\mathcal{D}}_{15}(1,d)$	15	$\tilde{O}(\min(\sqrt[4]{n}d, (nd)^{8/15}))$

TABLE 38.1 Maintaining α-approximate distances for all-pairs of vertices separated by distance $\leq d$.

algorithms given by Coppersmith and Winograd [3], the best bound for computing all-pairs shortest paths is $O(n^{2.575})$ [11].

Approximate distance oracles are designed by Thorup and Zwick [9]. Based on a 1963 girth conjecture of Erdős [6], they also show that $\Omega(n^{1+1/k})$ space is needed in the worst case for any oracle that achieves stretch strictly smaller than $(2k+1)$. The space requirement of their approximate distance oracle is, therefore, essentially optimal. Also note that the preprocessing time of $(2k-1)$-approximate distance oracle is $O(mn^{1/k})$, which is sub-cubic. However, for further improvement in the computation time for approximate distance oracles, Thorup and Zwick pose the following question : *Can* $(2k-1)$*-approximate distance oracle be computed in* $\tilde{O}(n^2)$ *time?* Recently Baswana and Sen [2] answer their question in affirmative for unweighted graphs. However, the question for weighted graphs is still open.

For maintaining fully dynamic all-pairs shortest paths in graphs, the best known algorithm is due to Demetrescu and Italiano [5]. They show that it takes $O(n^2)$ amortized time to maintain all-pairs exact shortest paths after each update in the graph. Baswana et al. [1] present a hierarchical data-structure based on random sampling that provides efficient decremental algorithm for maintaining APASP in undirected unweighted graphs. In addition to achieving $o(nd)$ update time for the problem APASP-d (as described in this chapter), they also employ the same hierarchical scheme for designing efficient data-structures for maintaining approximate distance information for all-pairs of vertices separated by distance in an interval $[a,b], 1 \leq a < b \leq n$.

Acknowledgment

The work of the first author is supported, in part, by a fellowship from Infosys Technologies Limited, Bangalore.

References

[1] Baswana, S., Hariharan R., and Sen, S., Maintaining all-pairs approximate shortest paths under deletion of edges, in *Proc. of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2003, 394.

[2] Baswana, S. and Sen S., Approximate distance oracle for unweighted graphs in $\tilde{O}(n^2)$ time, to appear in *Proc. of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2004.

[3] Coppersmith, D. and Winograd, S., Matrix multiplication via arithmetic progressions, *J. symbolic computation*, 9, 251, 1990.

[4] Cormen, T.H., Leiserson, C.E., and Rivest, R.L., *Introduction to Algorithms*, the MIT Press, 1990, chapter 12.

[5] Demetrescu, C., and Italiano, G.F., A new approach to dynamic all pairs shortest

paths, in *Proc. of 35th ACM Symposium on Theory of Computing (STOC)*, 2003, 159.

[6] Erdős, P., Extremal problems in graph theory, *Theory of Graphs and its Applications* (Proc. Sympos. Smolenice,1963), Publ. House Czechoslovak Acad. Sci., Prague. 1964, 29.

[7] Even, S. and Shiloach, Y., An on-line edge-deletion problem, *J. ACM*, 28, 1, 1981.

[8] Fredman, M.L., Komlós, J., and Szemerédi, E., Storing a sparse table with $O(1)$ worst case time, *J. ACM*, 31, 538, 1984.

[9] Thorup, M. and Zwick, U., Approximate distance oracles, in *Proc. of 33rd ACM symposium on theory of computing (STOC)*, 2001, 183.

[10] Zwick, U., Exact and approximate distances in graphs - a survey, in *Proc. of the 9th European Symposium on Algorithms (ESA)*, 2001, 33.

[11] Zwick, U., All-pairs shortest paths in weighted directed graphs - exact and almost exact algorithms, in *Proc. of the 39th IEEE Symposium on Foundations of Computer Science (FOCS)*, 1998, 310.

39

Searching and Priority Queues in o(log n) Time

Arne Andersson
Uppsala University

39.1 Introduction

In many cases of algorithm design, the comparison-based model of computation is not the obvious choice. In this chapter, we show how to design data structures with very good complexity on a realistic model of computation where keys are regarded as binary strings, each one contained in one or more machine words (registers). This model is sometimes referred to as the *RAM model* [*], and it may be argued that it reflects real computers more accurately than the comparison-based model.

In the RAM-model the *word length* becomes a natural part of the model. A comparison does not necessarily take constant time, on the other hand we may use a larger variety of operations on data. This model allows for comparison-based algorithms to be used but also for algorithms like tries, bucket sort, radix sort etc, which are known to be efficient in practice.

39.2 Model of Computation

We use a unit-cost RAM with word size w. In the standard case we assume that the n keys are w-bit keys that can be treated as binary strings or integers, but we may also consider

[*] The term RAM is used for many models. There are also RAM models with infinite word length.

key that occupy multiple words. It should be noted that the assumption that keys can be treated as binary strings or integers also holds for floating-point numbers (*cf.* IEEE 754 floating-point standard [18, p. 228]).

In the RAM-model, we can use other operations than comparisons, for instance indirect addressing, shifting, bitwise logical operations, and multiplication. Without loss of generality, we assume that $w = \Omega(\log n)$, since otherwise we could not even fit the number n, or a pointer, into a machine word. (If we can not fit the number n into a constant number of words, the traditional analysis for comparison-based algorithms would also fail.)

Our complexity analysis has *two* parameters, the number of keys n and the word length w. In cases where the complexity is expressed only in terms of n, it is supposed to hold for any possible value of w, and vice versa.

For the searching problem, we assume that an ordered set is maintained and that operations like range queries and neighbour queries are supported. We say that we study ordered dictionaries, as defined below.

DEFINITION 39.1 A dictionary is ordered if neighbour queries and range queries are supported at the same cost as member queries (plus the reporting cost), and if the keys can be reported in sorted order in linear time.

39.3 Overview

The basic purpose of this chapter is to introduce some of the basic techniques and give references to recent development:

- We start by presenting some simple data structures, which allow us to explain how the "information-theoretic $O(\log n)$ barrier" bay be surpassed. These data structures use a two-step approach: First, *range reduction* is used to decrease key length, such that we only need to consider keys that are much shorter than w. Secondly, we treat these short keys efficiently by *packed computing* where many keys are packed together in words.
- Next, we discuss some more elaborate data structures. In particular, we show how to achieve low worst-case complexity in linear space.
    - The fusion tree, the first data structure presented that achieved sublogarithmic complexity.
    - The *exponential search tree*, which achieves tight worst-case bound on dynamic ordered dictionaries.
- We also give references to recent results on efficient priority queue implementations and sorting.

39.4 Achieving Sub-Logarithmic Time per Element by Simple Means

In this section, we show that it is surprisingly simple to achieve a sublogarithmic complexity in n independent of w, which implies sorting and searching asymptotically faster than comparison-based algorithms.

We use indirect addressing and large arrays. As a consequence, the data structures will need much space. However, all algorithms presented here can be fit into linear space with randomization (i.e. with universal hashing [11]).

In some cases, we will consider keys that are shorter than w, we will then use b or k to denote key length.

In this section, we will use $F(n, b)$ to express the complexity of searching, as specified below.

DEFINITION 39.2 Let $F(n, b)$ be the worst-case cost of performing one search or update in an ordered dictionary storing n keys of length b.

Unless we use hashing to obtain linear space, the methods discussed in this section can all be implemented with a simple instruction set. All necessary instructions are standard, they are even in AC^0. (An instruction is in AC^0 if it is implementable by a constant depth, unbounded fan-in (AND,OR,NOT)-circuit of size $w^{O(1)}$. An example of a non-AC^0 instruction is multiplication [9].)

39.4.1 Range Reduction

One way to simplify a computational problem is by *range reduction*. In this case, we reduce the problem of dealing with w-bit keys to that of dealing with k-bits keys, $k < w$.

Assume that we view our w-bit keys as consisting of two $w/2$-bit characters and store these in a trie of height 2. Each internal node in the trie contains

- a reference to the min-element below the node; the min-element is not stored in any subtrie;
- a table of subtries, where each existing subtrie is represented by a $w/2$-bit key;
- a data structure for efficient neighbour search among the $w/2$-bit keys representing the subtries.

Since each node except the root has one incoming edge and each node contains exactly one element (the min-element), the trie has exactly n nodes and $n - 1$ edges.

We make neighbour searches in the following way: Traverse down the trie. If we find a leaf, the search ends, otherwise we end up at an empty entry in the subtrie table of some node. By making a neighbour search in that node, we are done. The cost for traversing the trie is $O(1)$ and the cost for a local neighbour search is $O(F(n, b/2))$ by definition.

The space requirements depend on how the table of subtrie pointers is implemented. If the table is implemented as an array of length $2^{b/2}$, each node in the trie requires $\Theta(2^{b/2})$ space. If we instead represent each table as a hash table, the total space of all hash tables is proportional to the total number of edges in the trie, which is $n - 1$.

We summarize this in the following equation.

$$F(n, w) = O(1) + F(n, w/2). \tag{39.1}$$

We can use the same construction recursively. That is, the local data structure for neighbour search among $w/2$-bit keys can be a trie of height 2 where $w/4$ bits are used for branching, etc.

In order to apply recursion properly, we have to be a bit careful with the space consumption. First, note that if the number of edges in a trie with n elements was larger than n, for instance $2n$, the total space (number of edges) would grow exponentially with the number of recursive levels. Therefore, we need to ensure that the number of edges in a trie is not just $O(n)$ but actually at most n. This is the reason why each node contains a min-element; in this way we guarantee that the number of edges is $n - 1$.

Secondly, even when we can guarantee that the space per recursive level does not increase, we are still faced with $\Theta(n)$ space (with hashing) per level. If we use more than a constant number of levels, this will require superlinear space. This is handled in the following way: When we apply the recursive construction r times, we only keep a small part of the elements in the recursive structure. Instead, the elements are kept in sorted lists of size $\Theta(r)$, and we keep only the smallest element from each list in our recursive trie. When searching for a key, we first search for its list in the recursive trie structure, we then scan the list. Insertions and deletions are made in the lists, and the sizes of the lists are maintained by merging and splitting lists. Now, the total space taken by each level of the recursive trie construction is $\Theta(n/r)$ and the total space for r recursive levels is $\Theta(n)$. Searching, splitting and merging within the lists only takes $O(r)$ time. In summary, setting $r = \log(w/k)$ we get the following lemma.

LEMMA 39.1 $F(n, w) = O(\log(w/k)) + F(n, k)$

This recursive reduction was first used in van Emde Boas trees [20, 25–27].

39.4.2 Packing Keys

If the word length is small enough—as in today's computers—the range reduction technique discussed above will decrease the key length to a constant at a low cost. However, in order to make a really convincing comparison between comparison-based algorithms and algorithms based on indirect addressing, we must make the complexity independent of the word size. This can be done by combining range reduction with *packed computation.* The basic idea behind packed computation is to exploit the bit-parallelism in a computer; many short keys can be packed in a word and treated simultaneously.

The central observation is due to Paul and Simon [21]; they observed that one subtraction can be used to perform comparisons in parallel. Assume that the keys are of length k. We may then pack $\Theta(w/k)$ keys in a word in the following way: Each key is represented by a $(k + 1)$-bit field. The first (leftmost) bit is a *test bit* and the following bits contain the key, cf. Figure 39.1. Let X and Y be two words containing the same number of packed keys, all test bits in X are 0 and all test bits in Y are 1. Let M be a fixed mask in which all test bits are 1 and all other bits are 0. Let

$$R \leftarrow (Y - X) \text{ AND } M. \tag{39.2}$$

Then, the ith test bit in R will be 1 if and only if $y_i > x_i$. All other test bits, as well as all other bits, in R will be 0.

We use packed comparisons to achieve the following result.

Y	1	00010	1	00111	1	01001	1	01110	1	10101	1	11000	1	11011	1	11110
X	0	01011	0	01011	0	01011	0	01011	0	01011	0	01011	0	01011	0	01011
$Y - X$	0	10111	0	11100	0	11110	1	00011	1	01010	1	01101	1	10000	0	10011
M	1	00000	1	00000	1	00000	1	00000	1	00000	1	00000	1	00000	1	00000
$(Y - X)$ AND M	0	00000	0	00000	0	00000	1	00000	1	00000	1	00000	1	00000	1	00000

FIGURE 39.1: A multiple comparison in a packed B-tree.

LEMMA 39.2 $F(n,k) = O\left(\log(w/k) + \frac{\log n}{\log(w/k)}\right)$.

Proof (Sketch) We use a packed B-tree [2].

Th packed B-tree has nodes of degree $\Theta(w/k)$. In each node, the search keys are packed together in a single word, in sorted order from left to right. When searching for a k-bit key x in a packed B-tree, we take the following two steps:

1. We construct a word X containing multiple copies of the query key x. X is created by a simple doubling technique: Starting with a word containing x in the rightmost part, we copy the word, shift the copy $k+1$ steps and unite the words with a bitwise OR. The resulting word is copied, shifted $2k+2$ steps and united, etc. Altogether X is generated in $O(\log(w/k))$ time.
2. After the word X has been constructed, we traverse the tree. At each node, we compute the rank of x in constant time with a packed comparison. The cost of the traversal is proportional to the height of the tree, which is $O(\log n / \log(w/k))$.

A packed comparison at a node is done as in Expression 39.2. The keys in the B-tree node are stored in Y and X contains multiple copies of the query key. After subtraction and masking, the rightmost p test bits in R will be 1 if and only if there are p keys in Y which are greater than x. This is illustrated in Figure 39.1. Hence, by finding the position of the leftmost 1-bit in R we can compute the rank of x among the keys in Y. In order to find the leftmost key, we can simply store all possible values of R in a lookup table. Since the number of possible values equals the number of keys in a B-tree node plus one, a hash table implementation of this lookup table would require only $\Theta(w/k)$ space.

Above, we omitted a lot of details, such as how to perform updates and how pointers within a packed B-tree are represented. Details can be found in [2].

39.4.3 Combining

We can now derive our first bounds for searching. First, we state bounds in terms of w. The following bound holds for searching [25–27]:

THEOREM 39.1 $F(n,w) = O(\log w)$.

Proof (Sketch) Apply Lemma 39.1 with $k = 1$.

Next, we show how to remove the dependency of word length [2]:

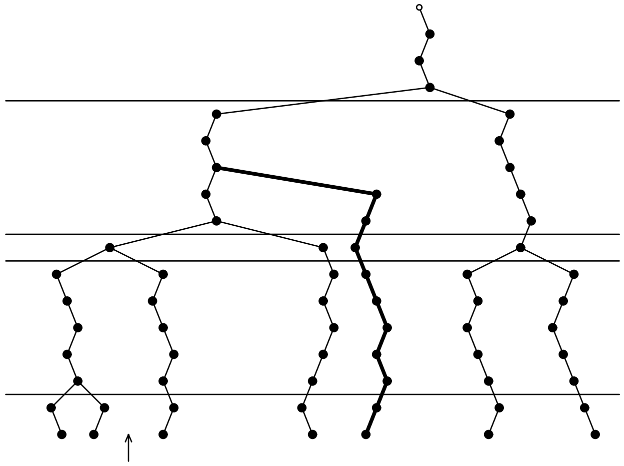


FIGURE 39.2: Searching in the internal trie in a fusion tree node. Horizontal lines represent significant bit positions. The thin paths represent the 6 keys in the trie, while the fat path represents the search key x (which is not present in the trie). The arrow marks the position of the compressed key x' among the keys in the trie.

THEOREM 39.2 $F(n, w) = O(\sqrt{\log n})$.

Proof (Sketch) If $\log w = O(\sqrt{\log n})$, Theorem 39.1 is sufficient. Otherwise, Lemma 39.1 with $k = w/2^{\sqrt{\log n}}$ gives $F(n, w) = O(\sqrt{\log n}) + F(n, w/2^{\sqrt{\log n}})$. Lemma 39.2 gives that $F(n, w/2^{\sqrt{\log n}}) = O(\sqrt{\log n})$.

39.5 Deterministic Algorithms and Linear Space

The data structures in this section are more complicated than the previous ones. They also need more powerful—but standard—instructions, like multiplication. On the other hand, these structures achieves linear space without randomization (i.e. without hashing).

DEFINITION 39.3 Let $D(n)$ be the worst-case search cost and the amortized update cost in an ordered dictionary storing n keys in $O(n)$ space.

39.5.1 Fusion Trees

The fusion tree was the first data structure to surpass the logarithmic barrier for searching. The central part of the fusion tree [13] is a static data structure with the following properties:

LEMMA 39.3 For any d, $d = O\left(w^{1/6}\right)$, a static data structure containing d keys can be constructed in $O\left(d^4\right)$ time and space, such that it supports neighbour queries in $O(1)$ worst-case time.

Proof (Sketch) The main idea behind the fusion tree is to view the keys as stored in an implicit binary trie and concentrate at the branching levels in this trie. We say that branching occurs at significant bit positions. We illustrate this view with an example, shown in Figure 39.2.

In the example, $w = 16$ and $d = 6$. We store a set Y of keys $y_1, \ldots, y_d$. Each key in Y is represented as a path in a binary trie. In the figure, a left edge denotes a 0 and a right edge denotes a 1. For example, y_3 is $\boxed{1010010101011010}$. The significant bit positions correspond to the branching levels in the trie. In this example the levels are 4, 9, 10, and 15, marked by horizontal lines. By extracting the significant bit positions from each key, we create a set Y' of compressed keys $y'_1, \ldots, y'_d$. In our example the compressed keys are $\boxed{0000}$, $\boxed{0001}$, $\boxed{0011}$, $\boxed{0110}$, $\boxed{1001}$, and $\boxed{1011}$. Since the trie has exactly d leaves, it contains exactly $d-1$ binary nodes. Therefore, the number of significant bit positions, and the length of a compressed key, is at most $d-1$. This implies that we can pack the d keys, including test bits, in d^2 bits. Since $d = O\left(w^{1/6}\right)$, the packed keys fit in a constant number of words.

This extraction of bits is nontrivial; it can be done with multiplication and masking. However, the extraction is not as perfect as described here; in order to avoid problems with carry bits etc, we need to extract some more bits than just the significant ones. Here, we ignore these problems and assume that we can extract the desired bits properly. For details we refer to Fredman and Willard [13].

The d compressed keys may be used to determine the rank of a query key among the original d keys with packed computation. Assume that we search for $x = \boxed{1010011001110100}$, represented as the fat path in Figure 39.2. First, we extract the proper bits to form a compressed key $x' = \boxed{0010}$. Then, we use packed searching to determine the rank of x' among $y'_1, \ldots, y'_d$.. In this case, the packed searching will place x' between y'_2 and y'_3. as indicated by the arrow in Figure 39.2. This is not the proper rank of the original key x, but nevertheless it is useful. The important information is obtained by finding the position of the first differing bit of x and one of the keys y_2 and y_3. In this example, the 7th bit is the first differing bit. and, since x has a 1 at this bit position, we can conclude that it is greater than all keys in Y with the same 6-bit prefix. Furthermore, the remaining bits in x are insignificant. Therefore, we can replace x by the key $\boxed{1010011111111111}$, where all the last bits are 1s. When compressed, this new key becomes $\boxed{0111}$. Making a second packed searching with this key instead, the proper rank will be found.

Hence, in constant time we can determine the rank of a query key among our d keys.

The original method by Fredman and Willard is slightly different. Instead of filling the query keys with 1s (or 0s) and making a second packed searching, they use a large lookup table in each node. Fusion trees can be implemented without multiplication, using only AC^0 instructions, provided that some simple non-standard instructions are allowed [5].

THEOREM 39.3 $D(n) = O(\log n / \log\log n)$.

Proof (Sketch) Based on Lemma 39.3, we use a B-tree where only the upper levels in the tree contain B-tree nodes, all having the same degree (within a constant factor). At the lower levels, traditional (i.e. comparison-based) weight-balanced trees are used. The reason for using weight-balanced trees is that the B-tree nodes are costly to reconstruct; the trees at the bottom ensure that few updates propagate to the upper levels. In this way, the amortized cost of updating a B-tree node is small.

The amortized cost of searches and updates is $O(\log n/\log d+\log d)$ for any $d = O\left(w^{1/6}\right)$. The first term corresponds to the number of B-tree levels and the second term corresponds to the height of the weight-balanced trees. Since $w \geq \log n$ (otherwise a pointer would not fit in a word), the cost becomes at most $O(\log n/\log\log n)$.

39.5.2 Exponential Search Trees

The *exponential search tree* [3, 7] allows efficient dynamization of static dictionary structures. The key feature is:

Any static data structure for searching that can be constructed in polynomial time and space can be efficiently used in a dynamic data structure.

The basic data structure is a multiway tree where the degrees of the nodes decrease doubly-exponentially down the tree. In each node, we use a static data structure for navigation. The way the tree is maintained, we can guarantee that, before an update occurs at a certain node, a polynomial number of updates will be made below it. Hence, even if an update requires a costly reconstruction of a static data structure, this will occur with large enough intervals.

LEMMA 39.4 Suppose a static data structure containing d keys can be constructed in $O\left(d^4\right)$ time and space, such that it supports neighbour queries in $O(S(d))$ worst-case time. Then,

$$D(n) = O\left(S\left(n^{1/5}\right)\right) + D\left(n^{4/5}\right);$$

Proof (Sketch) We use an exponential search tree. It has the following properties:

- Its root has degree $\Theta(n^{1/5})$.
- The keys of the root are stored in a local (static) data structure, with the properties stated above. During a search, the local data structure is used to determine in which subtree the search is to be continued.
- The subtrees are exponential search trees of size $\Theta(n^{4/5})$.

First, we show that, given n sorted keys, an exponential search tree can be constructed in linear time and space. The cost of constructing a node of degree d is $O\left(d^4\right)$, and the total construction cost $C(n)$ is (essentially) given by

$$C(n) = O\left(\left(n^{1/5}\right)^4\right) + n^{1/5} \cdot C\left(n^{4/5}\right) \qquad \Rightarrow \qquad C(n) = O(n). \tag{39.3}$$

Furthermore, with a similar equation, the space required by the data structure can be shown to be $O(n)$.

Balance is maintained by joining and splitting subtrees. The basic idea is the following: A join or split occurs when the size of a subtree has changed significantly, i.e. after $\Omega(n^{4/5})$ updates. Then, a constant number of subtrees will be reconstructed; according to Equation 39.3, the cost of this is linear in the size of the subtrees $= O(n^{4/5})$. Also, some keys will be inserted or deleted from the root, causing a reconstruction of the root; the cost of this is by definition $O(n^{4/5})$. Amortizing these two costs over the $\Omega(n^{4/5})$ updates, we get $O(1)$ amortized cost for reconstructing the root. Hence, the restructuring cost is dominated by the search cost.

Finally, the search cost follows immediately from the description of the exponential search tree.

Exponential search trees may be combined with various other data structures, as illustrated by the following two lemmas:

LEMMA 39.5 A static data structure containing d keys can be constructed in $O\left(d^4\right)$ time and space, such that it supports neighbour queries in $O\left(\frac{\log d}{\log w} + 1\right)$ worst-case time.

Proof (Sketch) We just construct a static B-tree where each node has the largest possible degree according to Lemma 39.3. That is, it has a degree of $\min\left(d, w^{1/6}\right)$. This tree satisfies the conditions of the lemma.

LEMMA 39.6 A static data structure containing d keys and supporting neighbour queries in $O(\log w)$ worst-case time can be constructed in $O\left(d^4\right)$ time and space.

Proof (Sketch) We study two cases.

Case 1: $w > d^{1/3}$. Lemma 39.5 gives constant query cost.

Case 2: $w \leq d^{1/3}$. The basic idea is to combine a van Emde Boas tree (Theorem 39.1) with perfect hashing. The data structure of Theorem 39.1 uses much space, which can be reduced to $O(d)$ by hash coding. Since we can afford a rather slow construction, we can use the deterministic algorithm by Fredman, Komlós, and Szemerédi [12]. With this algorithm, we can construct a perfectly hashed van Emde Boas tree in $O(d^3 w) = o(d^4)$ time.

Combining these two lemmas, we get a significantly improved upper bound on deterministic sorting and searching in linear space:

THEOREM 39.4 $D(n) = O(\sqrt{\log n})$.

Proof (Sketch) If we combine Lemmas 39.4, 39.5, and 39.6, we obtain the following equation

$$D(n) = O\left(\min\left(1 + \frac{\log n}{\log w}, \qquad \log w\right)\right) + D\left(n^{4/5}\right) \tag{39.4}$$

which, when solved, gives the theorem.

Taking both n and w as parameters, $D(n)$ is $o(\sqrt{\log n})$ in many cases [3]. For example, it can be shown that $D(n) = O(\log w \log\log n)$.

The strongest possible bound is achieved by using the following result by Beame and Fich [9]

LEMMA 39.7 [Beame and Fich [9]] In polynomial time and space, we can construct a deterministic data structure over d keys supporting searches in $O(\min\{\sqrt{\log d/\log\log d}, \frac{\log w}{\log\log w}\})$ time.

Combining this with the exponential search tree we get, among others, the following theorem.

THEOREM 39.5 $D(n) = O(\sqrt{\log n / \log\log n})$.

Since a matching lower bound was also given by Beame and Fich, this bound is optimal.

39.6 From Amortized Update Cost to Worst-Case

In fact, there are worst-case efficient versions of the data structures above. Willard [28] gives a short sketch on how to make fusion trees worst-case efficient, and as shown by Andersson and Thorup [7], the exponential search tree can be modified into a worst-case data structure. Here, we give a brief description of how exponential search trees are modified.

In the above definition of exponential search trees, the criteria for when a subtree is too large or too small depend on the degree of its parent. Therefore, when a node is joined or split, the requirements on its children will change. Above, we handled that by simply rebuilding the entire subtree at each join or split, but in a worst-case setting, we need to let the children of a node remain unchanged at a join or split. In order to do this, we need to switch from a top-down definition to a bottom-up definition.

DEFINITION 39.4 (Worst-case efficient exponential search trees) In an exponential search tree all leaves are on the same depth. Let the height of a node to be the distance from the node to the leaves descending from it. For a non-root node v at height $i > 0$, the weight (number of descending leaves) is $|v| = \Theta(n_i)$ where $n_i = \alpha^{(1+1/(k-1))^i}$ and $\alpha = \Theta(1)$. If the root has height h, its weight is $O(n_h)$.

With the exception of the root, Definition 39.4 follows our previous definition of exponential search trees (when $k = 5$), that is, if v is a non-root node, it has $\Theta(|v|^{1/k})$ children, each of weight $\Theta(|v|^{1-1/k})$.

The worst-case efficiency is mainly based on careful scheduling: the static search structures in the nodes are rebuilt in the background so that they remain sufficiently updated as nodes get joined and split. This scheduling is developed in terms of a general theorem about rebuilding, which has some interesting properties as a tool for other de-amortization applications [7].

39.7 Sorting and Priority Queues

In the comparison-based model of computation, the cost per element is the same ($O(\log n)$) for searching and sorting. However, in the RAM model of computation, the sorting problem can be solved faster than searching. The simple intuitive explanation of this is that the bit-parallelism in packed computation can be utilized more efficiently when a number of keys are treated simultaneously, as in sorting, than when they are treated one-by-one as in searching.

Even more, it turns out that priority queues can be as implemented as efficiently as sorting. (The intuitive reason for this is that a priority queue can use sorting as a subroutine, and only keep a small part of the queue perfectly sorted.) Thorup [24] has shown the following reduction:

THEOREM 39.6 *If we can sort n keys in time $S(n)$ per key, then we can implement a priority queue supporting find-min in constant time and updates (insert and delete) in $S(n)$ time.*

In the following, we will use $T(n, b)$ to denote the cost of sorting n b-bit keys.

39.7.1 Range Reduction

In sorting algorithms, range reduction is an often used technique. For example, we may view traditional radix sort, where we sort long strings by dividing them into shorter parts, as range reduction.

For our purposes, we will use a range reduction technique by Kirkpatrick and Reisch [19], which is similar to the van Emde Boas tree, cf. Section 39.4.1. The only difference from Section 39.4.1 is that instead of letting each trie node contain a data structure for efficient neighbour search among the outgoing edges, we just keep an unsorted list of all outgoing edges (plus the array for constant-time indexing of edges). Then, after all elements have been inserted into the trie, we create sorted lists of edges at all nodes by the following method:

1. Mark each edge with its parent node.
2. Concatenate all edge lists and sort the entire list.
3. Scan the sorted list and put each edge back in the proper node.
4. All edges lists are now sorted. By a recursive traversal of the trie we can report all leafs in sorted order.

Other details, such as the need to store one key per node to avoid space blow up, are handled in the same way as in Section 39.4.1. Altogether, we get the reduction

$$T(n, w) = O(n) + T(n, w/2). \tag{39.5}$$

Applied recursively, this gives

$$T(n, w) = O(n \log(w/k)) + T(n, k). \tag{39.6}$$

39.7.2 Packed Sorting

For the sorting problem, multiple comparisons can be utilized more efficiently than in a packed B-tree. In the packed B-tree, we used the bit-parallelism to compare one key to many keys, in this way we implemented a parallel version of a linear search, which is not the most efficient search method.

For sorting, however, we can utilize the packed computation more efficiently. It turns out that algorithms for *sorting networks* are well suited for implementation by packed computation. A sorting network is a “static” algorithm; it contains a number of compare-and-swap operations, these are the same regardless of the outcome of the comparisons. The merging technique by Batcher [8], originally used to design *odd-even merge sort*, can be efficiently implemented with packed computation. As a sorting network, Batcher's merging algorithm has depth $\Theta(\log n)$ where each level has $O(n)$ compare-and-swap units. Based on the merging, we can sort in $\Theta(\log^2 n)$ time where the total work is $\Theta(n \log^2 n)$

Batcher's merging technique is well suited for combination with the Paul-Simon technique, as shown by Albers and Hagerup [1].

LEMMA 39.8 $T(n, w/\log n) \leq O(n \log\log n)$.

Proof (Sketch) The key idea is that by packing $\Theta(\log n)$ keys in a machine word, we can combine Batcher's algorithm with packed computation to merge two sorted sequences in $O(\log\log n)$ time. And, if we can merge two sequences of length $\Theta(\log n)$ in $O(\log\log n)$ time (instead of $O(\log n)$ by a comparison-based algorithm), we can use this as a subroutine tom implement a variant of merge sort that sorts n keys in $O(n \log\log n)$ time (instead of $O(n \log n)$).

39.7.3 Combining the Techniques

First, the bound on searching from Theorem 39.1 has a corresponding theorem for sorting [19]:

THEOREM 39.7 $T(n, w) = O(n \log(w/\log n))$.

Proof (Sketch) Apply Eq. 39.6 with $k = \log n$. Keys of length $\log n$ can be sorted in linear time with bucket sort.

Secondly, we combine range reduction with packed computation. We get the following bound [4]:

THEOREM 39.8 $T(n, w) = O(n \log\log n)$.

Proof (Sketch) If $\log w = O(\log\log n)$, Theorem 39.7 is sufficient. Otherwise, Eq. 39.6 with $k = w/\log n$ gives $T(n, w) = O(n \log\log n) + T(n, w/\log n)$. Lemma 39.8 gives the final bound.

39.7.4 Further Techniques and Faster Randomized Algorithms

Apart from these rather simple techniques, there are a number of more elaborate techniques that allows the complexity to be improved further. Examples of such techniques are *signature sort* [4] and *packed bucketing* [17]. Here, we give a short sketch of signature sorting.

Consider a situation where the word length w is very large, and we wish to reduce the problem of sorting w-bit keys to that of sorting k-bit keys, $k \gg \log n$. Instead of treating these k-bit keys directly, we represent each such key by a b-bit *signature*, where the b bits are a hash function of the k bits. In fact, for one w-bit key, we can in constant time replace it by a shorter key, consisting of q b-bit signatures (for details, we refer to the literature [4, 17]).

1. Replace each w-bit key by a qb-bit key of concatenated signatures.
2. Sort the qb-bit keys.
3. Compute, for each qb-bit key, its first distinguishing signature. This can be done by constructing a signature-based trie of all keys.
4. If we know the first distinguishing signature in a qb-bit key, we know the first distinguishing k-bit field in the corresponding w-bit key. Finding these k-bit

fields, the range reduction is completed and we can continue by sorting these shorter keys.

It should be noted that the sorted set of qb-bit keys does not correspond to a sorted set of w-bit keys. However, the ordering we get is enough to find the proper distinguishing fields. Furthermore, since we use hash coding, we might get collisions, in which case the method will not work. By choosing b large enough, the risk of failure is small enough that we can afford to redo the entire sorting in case of failure: still the expected time for the range reduction step will be linear.

As an important recent result, Han and Thorup presents a linear algorithm for splitting n integers into subsets, where each subset is of size $O(\log n)$. Combining this with techniques like signature sorting, they manage to improve the randomized complexity of sorting to $O(n\sqrt{\log\log n})$. This, in turn, implies that a priority queue can be implemented at $O(\sqrt{\log\log n})$ time per update and find-min in constant time.

Other relevant reading can be found in the cited articles, or in [6, 10, 14–16, 22, 23]

References

[1] S. Albers and T. Hagerup. Improved parallel integer sorting without concurrent writing. *Inf. Contr.*, 136:25–51, 1997. Announced at SODA '92.

[2] A. Andersson. Sublogarithmic searching without multiplications. In *Proc. 36th FOCS*, pages 655–663, 1995.

[3] A. Andersson. Faster deterministic sorting and searching in linear space. In *Proc. 37th FOCS*, pages 135–141, 1996.

[4] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? *J. Comp. Syst. Sc.*, 57:74–93, 1998. Announced at STOC'95.

[5] A. Andersson, P.B. Miltersen, and M. Thorup. Fusion trees can be implemented with AC^0 instructions only. *Theoretical Computer Science*, 215(1-2):337–344, 1999.

[6] A. Andersson and S. Nilsson. A new efficient radix sort. In *Proc. 35th Annual IEEE Symposium on Foundations of Computer Science*, pages 714–721. IEEE Computer Society Press, 1994.

[7] A. Andersson and M. Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proc. 32th STOC*, 2000.

[8] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 307–314, 1968. Volume 32.

[9] P. Beame and F. Fich. Optimal bounds for the predecessor problem. In *Proc. 31st STOC*, pages 295–304, 1999.

[10] A. Brodnik, P. B. Miltersen, and I. Munro. Trans-dichotomous algorithms without multiplication - some upper and lower bounds. In *Proc. 5th WADS, LNCS 1272*, pages 426–439, 1997.

[11] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[12] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[13] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47:424–436, 1993. Announced at STOC'90.

[14] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.*, 48:533–551, 1994.

[15] Y. Han. Improved fast integer sorting in linear space. *Inform. Comput.*, 170(8):81–94, 2001. Announced at STACS'00 and SODA'01.

[16] Y. Han. Fast integer sorting in linear space. In *Proc.* 34^{th} *STOC*, pages 602–608, 2002.

[17] Y. Han and M. Thorup. Integer sorting in $o(n\sqrt{\log\log n})$ expected time and linear space. in Proc. FOCS '02, 2002.

[18] J. L. Hennessy and D. A. Patterson. *Computer Organization and Design: The Hardware/Software Interface.* Morgan Kaufmann Publ., San Mateo, CA, 1994.

[19] D. Kirkpatrick and S. Reisch. Upper bounds for sorting integers on random access machines. *Theor. Comp. Sc.*, 28:263–276, 1984.

[20] K. Mehlhorn and S. Nähler. Bounded ordered dictionaries in $O(\log\log n)$ time and $O(n)$ space. *Inf. Proc. Lett.*, 35(4):183–189, 1990.

[21] W. J. Paul and J. Simon. Decision trees and random access machines. In *Logic and Algorithmic: An International Symposium Held in Honour of Ernst Specker*, pages 331–340. L'Enseignement Mathématique, Université de Genevè, 1982.

[22] R. Raman. Priority queues: small, monotone and trans-dichotomous. In *Proc.* 4^{th} *ESA, LNCS 1136*, pages 121–137, 1996.

[23] M. Thorup. On RAM priority queues. *SIAM J. Comp.*, 30(1):86–109, 2000.

[24] M. Thorup. Equivalence between priority queues and sorting, 2002. in Proc. FOCS'02.

[25] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual IEEE Symposium on Foundations of Computer Science*, pages 75–84, 1975.

[26] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Proc. Lett.*, 6(3):80–82, 1977.

[27] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Syst. Theory*, 10:99–127, 1977.

[28] D. E. Willard. Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM Journal on Computing*, 29(3):1030–1049, 2000. Announced at SODA'92.