2 Divide-and-Conquer

We use quicksort as an example for an algorithm that follows the divide-and-conquer paradigm. It has the reputation of being the fasted comparison-based sorting algorithm. Indeed it is very fast on the average but can be slow for some input, unless precautions are taken.

The algorithm. Quicksort follows the general paradigm of divide-and-conquer, which means it **divides** the unsorted array into two, it **recurses** on the two pieces, and it finally **combines** the two sorted pieces to obtain the sorted array. An interesting feature of quicksort is that the divide step separates small from large items. As a consequence, combining the sorted pieces happens automatically without doing anything extra.

```
void QUICKSORT(int ℓ, r)
  if ℓ < r then m = SPLIT(ℓ, r);
                QUICKSORT(ℓ, m − 1);
                QUICKSORT(m + 1, r)
  endif.
```

We assume the items are stored in $A[0..n-1]$. The array is sorted by calling QUICKSORT$(0, n-1)$.

Splitting. The performance of quicksort depends heavily on the performance of the split operation. The effect of splitting from ℓ to r is:

- $x = A[\ell]$ is moved to its correct location at $A[m]$;
- no item in $A[\ell..m-1]$ is larger than x;
- no item in $A[m+1..r]$ is smaller than x.

Figure 1 illustrates the process with an example. The nine items are split by moving a pointer i from left to right and another pointer j from right to left. The process stops when i and j cross. To get splitting right is a bit delicate, in particular in special cases. Make sure the algorithm is correct for (i) x is smallest item, (ii) x is largest item, (iii) all items are the same.

```
int SPLIT(int ℓ, r)
  x = A[ℓ]; i = ℓ; j = r + 1;
  repeat repeat i++ until x ≤ A[i];
         repeat j-- until x ≥ A[j];
         if i < j then SWAP(i, j) endif
  until i ≥ j;
  SWAP(ℓ, j); return j.
```

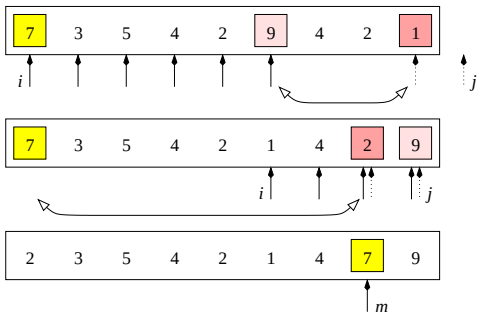

Figure 1: First, i and j stop at items 9 and 1, which are then swapped. Second, i and j cross and the pivot, 7, is swapped with item 2.

Special cases (i) and (iii) are ok but case (ii) requires a stopper at $A[r+1]$. This stopper must be an item at least as large as x. If $r < n-1$ this stopper is automatically given. For $r = n-1$, we create such a stopper by setting $A[n] = +\infty$.

Running time. The actions taken by quicksort can be expressed using a binary tree: each (internal) node represents a call and displays the length of the subarray; see Figure 2. The worst case occurs when A is already sorted.

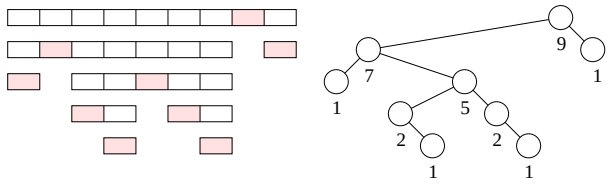

Figure 2: The total amount of time is proportional to the sum of lengths, which are the numbers of nodes in the corresponding subtrees. In the displayed case this sum is 29.

In this case the tree degenerates to a list without branching. The sum of lengths can be described by the following recurrence relation:

$$T(n) \;=\; n + T(n-1) \;=\; \sum_{i=1}^{n} i \;=\; \binom{n+1}{2}.$$

The running time in the worst case is therefore in $O(n^2)$.

In the best case the tree is completely balanced and the sum of lengths is described by the recurrence relation

$$T(n) \;=\; n + 2 \cdot T\left(\frac{n-1}{2}\right).$$

If we assume $n = 2^k - 1$ we can rewrite the relation as

$$\begin{aligned} U(k) &= (2^k - 1) + 2 \cdot U(k-1) \\ &= (2^k - 1) + 2(2^{k-1} - 1) + \ldots + 2^{k-1}(2-1) \\ &= k \cdot 2^k - \sum_{i=0}^{k-1} 2^i \\ &= 2^k \cdot k - (2^k - 1) \\ &= (n+1) \cdot \log_2(n+1) - n. \end{aligned}$$

The running time in the best case is therefore in $\mathrm{O}(n \log n)$.

Randomization. One of the drawbacks of quicksort, as described until now, is that it is slow on rather common almost sorted sequences. The reason are pivots that tend to create unbalanced splittings. Such pivots tend to occur in practice more often than one might expect. Human and often also machine generated data is frequently biased towards certain distributions (in this case, permutations), and it has been said that 80% of the time or more, sorting is done on either already sorted or almost sorted files. Such situations can often be helped by transferring the algorithm's dependence on the input data to internally made random choices. In this particular case, we use randomization to make the choice of the pivot independent of the input data. Assume $\textsc{Random}(\ell, r)$ returns an integer $p \in [\ell, r]$ with uniform probability:

$$\mathrm{Prob}[\textsc{Random}(\ell, r) = p] = \frac{1}{r - \ell + 1}$$

for each $\ell \le p \le r$. In other words, each $p \in [\ell, r]$ is equally likely. The following algorithm splits the array with a random pivot:

```
int rSplit(int ℓ, r)
  p = Random(ℓ, r); Swap(ℓ, p);
  return Split(ℓ, r).
```

We get a *randomized* implementation by substituting rSplit for Split. The behavior of this version of quicksort depends on p, which is produced by a random number generator.

Average analysis. We assume that the items in $A[0..n-1]$ are pairwise different. The pivot splits A into

$$A[0..m-1], \quad A[m], \quad A[m+1..n-1].$$

By assumption on function rSplit, the probability for each $m \in [0, n-1]$ is $\frac{1}{n}$. Therefore the average sum of array lengths split by QuickSort is

$$T(n) = n + \frac{1}{n} \cdot \sum_{m=0}^{n-1} \left(T(m) + T(n-m-1)\right).$$

To simplify, we multiply with n and obtain a second relation by substituting $n-1$ for n:

$$n \cdot T(n) = n^2 + 2 \cdot \sum_{i=0}^{n-1} T(i), \qquad (1)$$

$$(n-1) \cdot T(n-1) = (n-1)^2 + 2 \cdot \sum_{i=0}^{n-2} T(i). \qquad (2)$$

Next we subtract (2) from (1), we divide by $n(n+1)$, we use repeated substitution to express $T(n)$ as a sum, and finally split the sum in two:

$$\begin{aligned} \frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + \frac{2n-1}{n(n+1)} \\ &= \frac{T(n-2)}{n-1} + \frac{2n-3}{(n-1)n} + \frac{2n-1}{n(n+1)} \\ &= \sum_{i=1}^{n} \frac{2i-1}{i(i+1)} \\ &= 2 \cdot \sum_{i=1}^{n} \frac{1}{i+1} - \sum_{i=1}^{n} \frac{1}{i(i+1)}. \end{aligned}$$

Bounding the sums. The second sum is solved directly by transformation to a telescoping series:

$$\begin{aligned} \sum_{i=1}^{n} \frac{1}{i(i+1)} &= \sum_{i=1}^{n} \left(\frac{1}{i} - \frac{1}{i+1}\right) \\ &= 1 - \frac{1}{n+1}. \end{aligned}$$

The first sum is bounded from above by the integral of $\frac{1}{x}$ for x ranging from 1 to $n+1$; see Figure 3. The sum of $\frac{1}{i+1}$ is the sum of areas of the shaded rectangles, and because all rectangles lie below the graph of $\frac{1}{x}$ we get a bound for the total rectangle area:

$$\sum_{i=1}^{n} \frac{1}{i+1} < \int_1^{n+1} \frac{\mathrm{d}x}{x} = \ln(n+1).$$

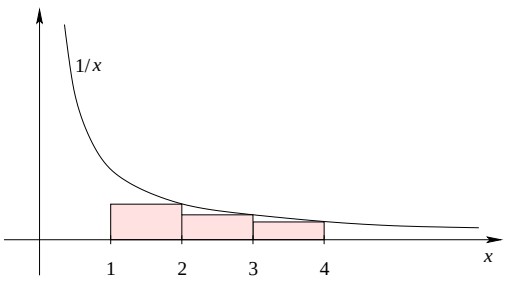

Figure 3: The areas of the rectangles are the terms in the sum, and the total rectangle area is bounded by the integral from 1 through $n+1$.

We plug this bound back into the expression for the average running time:

$$\begin{aligned} T(n) &< (n+1) \cdot \sum_{i=1}^{n} \frac{2}{i+1} \\ &< 2 \cdot (n+1) \cdot \ln(n+1) \\ &= \frac{2}{\log_2 e} \cdot (n+1) \cdot \log_2(n+1). \end{aligned}$$

In words, the running time of quicksort in the average case is only a factor of about $2/\log_2 e = 1.386\ldots$ slower than in the best case. This also implies that the worst case cannot happen very often, for else the average performance would be slower.

Stack size. Another drawback of quicksort is the recursion stack, which can reach a size of $\Omega(n)$ entries. This can be improved by always first sorting the smaller side and simultaneously removing the tail-recursion:

```
void QUICKSORT(int ℓ, r)
  i = ℓ; j = r;
  while i < j do
    m = RSPLIT(i, j);
    if m − i < j − m
      then QUICKSORT(i, m − 1); i = m + 1
      else QUICKSORT(m + 1, j); j = m − 1
    endif
  endwhile.
```

In each recursive call to QUICKSORT, the length of the array is at most half the length of the array in the preceding call. This implies that at any moment of time the stack contains no more than $1 + \log_2 n$ entries. Note that without removal of the tail-recursion, the stack can reach $\Omega(n)$ entries even if the smaller side is sorted first.

Summary. Quicksort incorporates two design techniques to efficiently sort n numbers: divide-and-conquer for reducing large to small problems and randomization for avoiding the sensitivity to worst-case inputs. The average running time of quicksort is in $\mathrm{O}(n \log n)$ and the extra amount of memory it requires is in $\mathrm{O}(\log n)$. For the deterministic version, the average is over all $n!$ permutations of the input items. For the randomized version the average is the expected running time for *every* input sequence.

3 Prune-and-Search

We use two algorithms for selection as examples for the prune-and-search paradigm. The problem is to find the i-smallest item in an unsorted collection of n items. We could first sort the list and then return the item in the i-th position, but just finding the i-th item can be done faster than sorting the entire list. As a warm-up exercise consider selecting the 1-st or smallest item in the unsorted array $A[1..n]$.

```
min = 1;
for j = 2 to n do
  if A[j] < A[min] then min = j endif
endfor.
```

The index of the smallest item is found in $n-1$ comparisons, which is optimal. Indeed, there is an adversary argument, that is, with fewer than $n-1$ comparisons we can change the minimum without changing the outcomes of the comparisons.

Randomized selection. We return to finding the i-smallest item for a fixed but arbitrary integer $1 \leq i \leq n$, which we call the *rank* of that item. We can use the splitting function of quicksort also for selection. As in quicksort, we choose a random pivot and split the array, but we recurse only for one of the two sides. We invoke the function with the range of indices of the current subarray and the rank of the desired item, i. Initially, the range consists of all indices between $\ell = 1$ and $r = n$, limits included.

```
int RSELECT(int ℓ, r, i)
  q = RSPLIT(ℓ, r); m = q − ℓ + 1;
  if i < m then return RSELECT(ℓ, q − 1, i)
    elseif i = m then return q
    else return RSELECT(q + 1, r, i − m)
  endif.
```

For small sets, the algorithm is relatively ineffective and its running time can be improved by switching over to sorting when the size drops below some constant threshold. On the other hand, each recursive step makes some progress so that termination is guaranteed even without special treatment of small sets.

Expected running time. For each $1 \leq m \leq n$, the probability that the array is split into subarrays of sizes $m-1$ and $n-m$ is $\frac{1}{n}$. For convenience we assume that n is even. The expected running time increases with increasing number of items, $T(k) \leq T(m)$ if $k \leq m$. Hence,

$$\begin{aligned} T(n) &\leq n + \frac{1}{n}\sum_{m=1}^{n} \max\{T(m-1), T(n-m)\} \\ &\leq n + \frac{2}{n}\sum_{m=\frac{n}{2}+1}^{n} T(m-1). \end{aligned}$$

Assume inductively that $T(m) \leq cm$ for $m < n$ and a sufficiently large positive constant c. Such a constant c can certainly be found for $m = 1$, since for that case the running time of the algorithm is only a constant. This establishes the basis of the induction. The case of n items reduces to cases of $m < n$ items for which we can use the induction hypothesis. We thus get

$$\begin{aligned} T(n) &\leq n + \frac{2c}{n}\sum_{m=\frac{n}{2}+1}^{n} m - 1 \\ &= n + c\cdot(n-1) - \frac{c}{2}\cdot\left(\frac{n}{2}+1\right) \\ &= n + \frac{3c}{4}\cdot n - \frac{3c}{2}. \end{aligned}$$

Assuming $c \geq 4$ we thus have $T(n) \leq cn$ as required. Note that we just proved that the expected running time of RSELECT is only a small constant times that of RSPLIT. More precisely, that constant factor is no larger than four.

Deterministic selection. The randomized selection algorithm takes time proportional to n^2 in the worst case, for example if each split is as unbalanced as possible. It is however possible to select in O(n) time even in the worst case. The *median* of the set plays a special role in this algorithm. It is defined as the i-smallest item where $i = \frac{n+1}{2}$ if n is odd and $i = \frac{n}{2}$ or $\frac{n+2}{2}$ if n is even. The deterministic algorithm takes five steps to select:

`Step 1.` Partition the n items into $\lceil \frac{n}{5} \rceil$ groups of size at most 5 each.

`Step 2.` Find the median in each group.

`Step 3.` Find the median of the medians recursively.

`Step 4.` Split the array using the median of the medians as the pivot.

`Step 5.` Recurse on one side of the pivot.

It is convenient to define $k = \lceil \frac{n}{5} \rceil$ and to partition such that each group consists of items that are multiples of k positions apart. This is what is shown in Figure 4 provided we arrange the items row by row in the array.

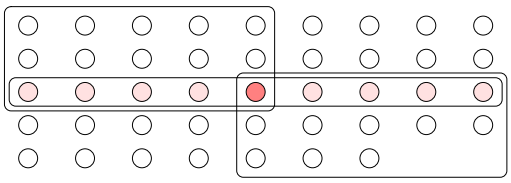

Figure 4: The 43 items are partitioned into seven groups of 5 and two groups of 4, all drawn vertically. The shaded items are the medians and the dark shaded item is the median of medians.

Implementation with insertion sort. We use insertion sort on each group to determine the medians. Specifically, we sort the items in positions $\ell, \ell+k, \ell+2k, \ell+3k, \ell+4k$ of array A, for each ℓ.

```
void ISort(int ℓ, k, n)
  j = ℓ + k;
  while j ≤ n do i = j;
    while i > ℓ and A[i] > A[i − k] do
      Swap(i, i − k); i = i − k
    endwhile;
    j = j + k
  endwhile.
```

Although insertion sort takes quadratic time in the worst case, it is very fast for small arrays, as in this application. We can now combine the various pieces and write the selection algorithm in pseudo-code. Starting with the code for the randomized algorithm, we first remove the randomization and second add code for Steps 1, 2, and 3. Recall that i is the rank of the desired item in $A[\ell..r]$. After sorting the groups, we have their medians arranged in the middle fifth of the array, $A[\ell+2k..\ell+3k-1]$, and we compute the median of the medians by recursive application of the function.

```
int Select(int ℓ, r, i)
  k = ⌈(r − ℓ + 1)/5⌉;
  for j = 0 to k − 1 do ISort(ℓ + j, k, r) endfor;
  m′ = Select(ℓ + 2k, ℓ + 3k − 1, ⌊(k + 1)/2⌋);
  Swap(ℓ, m′); q = Split(ℓ, r); m = q − ℓ + 1;
  if i < m then return Select(ℓ, q − 1, i)
    elseif i = m then return q
    else return Select(q + 1, r, i − m)
  endif.
```

Observe that the algorithm makes progress as long as there are at least three items in the set, but we need special treatment of the cases of one or of two items. The role of the median of medians is to prevent an unbalanced split of the array so we can safely use the deterministic version of splitting.

Worst-case running time. To simplify the analysis, we assume that n is a multiple of 5 and ignore ceiling and floor functions. We begin by arguing that the number of items less than or equal to the median of medians is at least $\frac{3n}{10}$. These are the first three items in the sets with medians less than or equal to the median of medians. In Figure 4, these items are highlighted by the box to the left and above but containing the median of medians. Symmetrically, the number of items greater than or equal to the median of medians is at least $\frac{3n}{10}$. The first recursion works on a set of $\frac{n}{5}$ medians, and the second recursion works on a set of at most $\frac{7n}{10}$ items. We have

$$T(n) \quad \leq \quad n + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right).$$

We prove $T(n) = \mathrm{O}(n)$ by induction assuming $T(m) \leq c \cdot m$ for $m < n$ and c a large enough constant.

$$\begin{aligned} T(n) &\leq n + \frac{c}{5} \cdot n + \frac{7c}{10} \cdot n \\ &= \left(1 + \frac{9c}{10}\right) \cdot n. \end{aligned}$$

Assuming $c \geq 10$ we have $T(n) \leq cn$, as required. Again the running time is at most some constant times that of splitting the array. The constant is about two and a half times the one for the randomized selection algorithm.

A somewhat subtle issue is the presence of equal items in the input collection. Such occurrences make the function Split unpredictable since they could occur on either side of the pivot. An easy way out of the dilemma is to make sure that the items that are equal to the pivot are treated as if they were smaller than the pivot if they occur in the first half of the array and they are treated as if they were larger than the pivot if they occur in the second half of the array.

Summary. The idea of prune-and-search is very similar to divide-and-conquer, which is perhaps the reason why some textbooks make no distinction between the two. The characteristic feature of prune-and-search is that the recursion covers only a constant fraction of the input set. As we have seen in the analysis, this difference implies a better running time.

It is interesting to compare the randomized with the deterministic version of selection:

- the use of randomization leads to a simpler algorithm but it requires a source of randomness;
- upon repeating the algorithm for the same data, the deterministic version goes through the exact same steps while the randomized version does not;
- we analyze the worst-case running time of the deterministic version and the expected running time (for the worst-case input) of the randomized version.

All three differences are fairly universal and apply to other algorithms for which we have the choice between a deterministic and a randomized implementation.

4 Dynamic Programming

Sometimes, divide-and-conquer leads to overlapping subproblems and thus to redundant computations. It is not uncommon that the redundancies accumulate and cause an exponential amount of wasted time. We can avoid the waste using a simple idea: **solve each subproblem only once**. To be able to do that, we have to add a certain amount of book-keeping to remember subproblems we have already solved. The technical name for this design paradigm is *dynamic programming*.

Edit distance. We illustrate dynamic programming using the edit distance problem, which is motivated by questions in genetics. We assume a finite set of *characters* or *letters*, Σ, which we refer to as the *alphabet*, and we consider *strings* or *words* formed by concatenating finitely many characters from the alphabet. The *edit distance* between two words is the minimum number of letter insertions, letter deletions, and letter substitutions required to transform one word to the other. For example, the edit distance between FOOD and MONEY is at most four:

$$\texttt{FOOD} \rightarrow \texttt{MOOD} \rightarrow \texttt{MOND} \rightarrow \texttt{MONED} \rightarrow \texttt{MONEY}$$

A better way to display the editing process is the *gap representation* that places the words one above the other, with a gap in the first word for every insertion and a gap in the second word for every deletion:

```
F O O   D
M O N E Y
```

Columns with two different characters correspond to substitutions. The number of editing steps is therefore the number of columns that do not contain the same character twice.

Prefix property. It is not difficult to see that you cannot get from FOOD to MONEY in less than four steps. However, for longer examples it seems considerably more difficult to find the minimum number of steps or to recognize an optimal edit sequence. Consider for example

```
A L G O R   I   T H M
A L   T R U I S T I C
```

Is this optimal or, equivalently, is the edit distance between ALGORITHM and ALTRUISTIC six? Instead of answering this specific question, we develop a dynamic programming algorithm that computes the edit distance between an m-character string $A[1..m]$ and an n-character string $B[1..n]$. Let $E(i, j)$ be the edit distance between the prefixes of length i and j, that is, between $A[1..i]$ and $B[1..j]$. The edit distance between the complete strings is therefore $E(m, n)$. A crucial step towards the development of this algorithm is the following observation about the gap representation of an optimal edit sequence.

PREFIX PROPERTY. If we remove the last column of an optimal edit sequence then the remaining columns represent an optimal edit sequence for the remaining substrings.

We can easily prove this claim by contradiction: if the substrings had a shorter edit sequence, we could just glue the last column back on and get a shorter edit sequence for the original strings.

Recursive formulation. We use the Prefix Property to develop a recurrence relation for E. The dynamic programming algorithm will be a straightforward implementation of that relation. There are a couple of obvious base cases:

- **Erasing:** we need i deletions to erase an i-character string, $E(i, 0) = i$.
- **Creating:** we need j insertions to create a j-character string, $E(0, j) = j$.

In general, there are four possibilities for the last column in an optimal edit sequence.

- **Insertion:** the last entry in the top row is empty, $E(i, j) = E(i, j - 1) + 1$.
- **Deletion:** the last entry in the bottom row is empty, $E(i, j) = E(i - 1, j) + 1$.
- **Substitution:** both rows have characters in the last column that are different, $E(i, j) = E(i - 1, j - 1) + 1$.
- **No action:** both rows end in the same character, $E(i, j) = E(i - 1, j - 1)$.

Let P be the logical proposition $A[i] \neq B[j]$ and denote by $|P|$ its indicator variable: $|P| = 1$ if P is true and $|P| = 0$ if P is false. We can now summarize and for $i, j > 0$ get the edit distance as the smallest of the possibilities:

$$E(i, j) \quad = \quad \min \left\{ \begin{array}{l} E(i, j - 1) + 1 \\ E(i - 1, j) + 1 \\ E(i - 1, j - 1) + |P| \end{array} \right\}.$$

The algorithm. If we turned this recurrence relation directly into a divide-and-conquer algorithm, we would have the following recurrence for the running time:

$$\begin{aligned} T(m,n) &= T(m,n-1)+T(m-1,n) \\ &\quad + T(m-1,n-1)+1. \end{aligned}$$

The solution to this recurrence is exponential in m and n, which is clearly not the way to go. Instead, let us build an $m+1$ times $n+1$ table of possible values of $E(i,j)$. We can start by filling in the base cases, the entries in the 0-th row and column. To fill in any other entry, we need to know the values directly to the left, directly above, and both to the left and above. If we fill the table from top to bottom and from left to right then whenever we reach an entry, the entries it depends on are already available.

```
int EDITDISTANCE(int m, n)
  for i = 0 to m do E[i,0] = i endfor;
  for j = 1 to n do E[0,j] = j endfor;
  for i = 1 to m do
    for j = 1 to n do
      E[i,j] = min{E[i,j-1]+1, E[i-1,j]+1,
                   E[i-1,j-1]+|A[i] ≠ B[j]|}
    endfor
  endfor;
  return E[m,n].
```

Since there are $(m+1)(n+1)$ entries in the table and each takes a constant time to compute, the total running time is in $O(mn)$.

An example. The table constructed for the conversion of ALGORITHM to ALTRUISTIC is shown in Figure 5. Boxed numbers indicate places where the two strings have equal characters. The arrows indicate the predecessors that define the entries. Each direction of arrow corresponds to a different edit operation: horizontal for insertion, vertical for deletion, and diagonal for substitution. Dotted diagonal arrows indicate free substitutions of a letter for itself.

Recovering the edit sequence. By construction, there is at least one path from the upper left to the lower right corner, but often there will be several. Each such path describes an optimal edit sequence. For the example at hand, we have three optimal edit sequences:

```
A L G O R I   T H M
A L T R U I S T I C
```

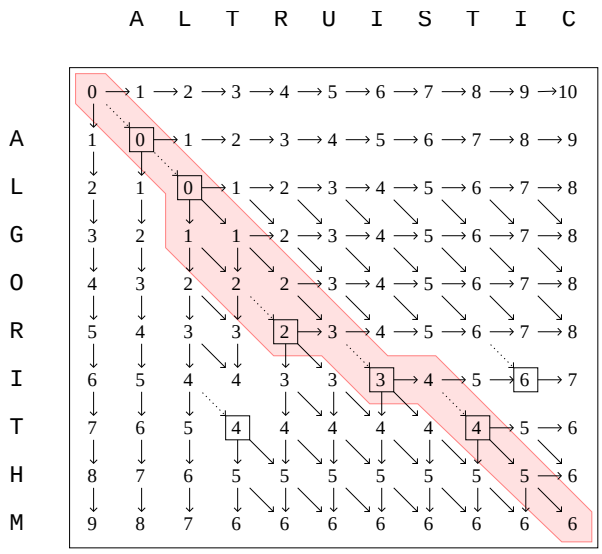

Figure 5: The table of edit distances between all prefixes of ALGORITHM and of ALTRUISTIC. The shaded area highlights the optimal edit sequences, which are paths from the upper left to the lower right corner.

```
A L G O R   I   T H M
A L T   R U I S T I C

A L G O R   I   T H M
A L   T R U I S T I C
```

They are easily recovered by tracing the paths backward, from the end to the beginning. The following algorithm recovers an optimal solution that also minimizes the number of insertions and deletions. We call it with the lengths of the strings as arguments, $\mathrm{R}(m,n)$.

```
void R(int i,j)
  if i > 0 or j > 0 then
    switch incoming arrow:
      case ↘: R(i-1,j-1); print(A[i], B[j])
      case ↓: R(i-1,j); print(A[i], _)
      case →: R(i,j-1); print(_, B[j]).
    endswitch
  endif.
```

Summary. The structure of dynamic programming is again similar to divide-and-conquer, except that the subproblems to be solved overlap. As a consequence, we get different recursive paths to the same subproblems. To develop a dynamic programming algorithm that avoids redundant solutions, we generally proceed in two steps:

1. We formulate the problem recursively. In other words, we write down the answer to the whole problem as a combination of the answers to smaller subproblems.
2. We build solutions from bottom up. Starting with the base cases, we work our way up to the final solution and (usually) store intermediate solutions in a table.

For dynamic programming to be effective, we need a structure that leads to at most some polynomial number of different subproblems. Most commonly, we deal with sequences, which have linearly many prefixes and suffixes and quadratically many contiguous substrings.

5 Greedy Algorithms

The philosophy of being greedy is shortsightedness. Always go for the seemingly best next thing, always optimize the presence, without any regard for the future, and never change your mind about the past. The greedy paradigm is typically applied to optimization problems. In this section, we first consider a scheduling problem and second the construction of optimal codes.

A scheduling problem. Consider a set of activities $1, 2, \ldots, n$. Activity i starts at time s_i and finishes at time $f_i > s_i$. Two activities i and j *overlap* if $[s_i, f_i] \cap [s_j, f_j] \neq \emptyset$. The objective is to select a maximum number of pairwise non-overlapping activities. An example is shown in Figure 6. The largest number of activities can be scheduled by choosing activities with early finish times first. We first sort and reindex such that $i < j$ implies $f_i \leq f_j$.

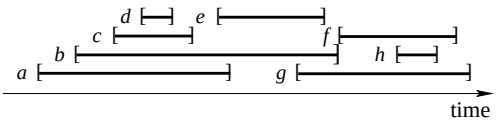

Figure 6: A best schedule is c, e, f, but there are also others of the same size.

$S = \{1\}$; $last = 1$;
`for` $i = 2$ `to` n `do`
  `if` $f_{last} < s_i$ `then`
    $S = S \cup \{i\}$; $last = i$
  `endif`
`endfor`.

The running time is O($n \log n$) for sorting plus O(n) for the greedy collection of activities.

It is often difficult to determine how close to the optimum the solutions found by a greedy algorithm really are. However, for the above scheduling problem the greedy algorithm always finds an optimum. For the proof let $1 = i_1 < i_2 < \ldots < i_k$ be the greedy schedule constructed by the algorithm. Let $j_1 < j_2 < \ldots < j_\ell$ be any other feasible schedule. Since $i_1 = 1$ has the earliest finish time of any activity, we have $f_{i_1} \leq f_{j_1}$. We can therefore add i_1 to the feasible schedule and remove at most one activity, namely j_1. Among the activities that do not overlap i_1, i_2 has the earliest finish time, hence $f_{i_2} \leq f_{j_2}$. We can again add i_2 to the feasible schedule and remove at most one activity, namely j_2 (or possibly j_1 if it was not removed before). Eventually, we replace the entire feasible schedule by the greedy schedule without decreasing the number of activities. Since we could have started with a maximum feasible schedule, we conclude that the greedy schedule is also maximum.

Binary codes. Next we consider the problem of encoding a text using a string of 0s and 1s. A *binary code* maps each letter in the alphabet of the text to a unique string of 0s and 1s. Suppose for example that the letter 't' is encoded as '001', 'h' is encoded as '101', and 'e' is encoded as '01'. Then the word 'the' would be encoded as the concatenation of codewords: '00110101'. This particular encoding is unambiguous because the code is *prefix-free*: no codeword is prefix of another codeword. There is a one-to-one correspondence between prefix-free binary codes and binary trees where each leaf is a letter and the corresponding codeword is the path from the root to that leaf. Figure 7 illustrates the correspondence for the above 3-letter code. Being prefix-free corresponds to leaves not having children. The tree in Figure 7 is not full because three of its internal nodes have only one child. This is an indication of waste. The code can be improved by replacing each node with one child by its child. This changes the above code to '00' for 't', '1' for 'h', and '01' for 'e'.

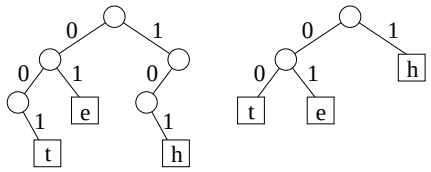

Figure 7: Letters correspond to leaves and codewords correspond to maximal paths. A left edge is read as '0' and a right edge as '1'. The tree to the right is full and improves the code.

Huffman trees. Let w_i be the frequency of the letter c_i in the given text. It will be convenient to refer to w_i as the *weight* of c_i or of its external node. To get an efficient code, we choose short codewords for common letters. Suppose δ_i is the length of the codeword for c_i. Then the number of bits for encoding the entire text is

$$P = \sum_i w_i \cdot \delta_i.$$

Since δ_i is the depth of the leaf c_i, P is also known as the *weighted external path length* of the corresponding tree.

The *Huffman tree* for the c_i minimizes the weighted external path length. To construct this tree, we start with n nodes, one for each letter. At each stage of the algorithm, we greedily pick the two nodes with smallest weights and make them the children of a new node with weight equal to the sum of two weights. We repeat until only one node remains. The resulting tree for a collection of nine letters with displayed weights is shown in Figure 8. Ties that

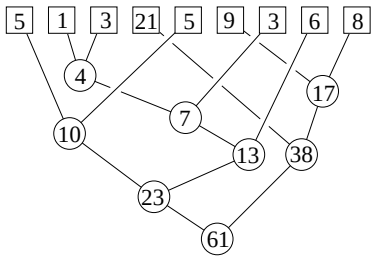

Figure 8: The numbers in the external nodes (squares) are the weights of the corresponding letters, and the ones in the internal nodes (circles) are the weights of these nodes. The Huffman tree is full by construction.

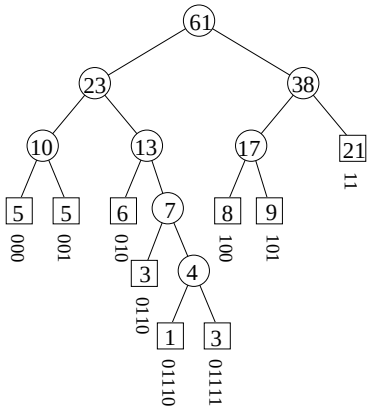

Figure 9: The weighted external path length is $15 + 15 + 18 + 12 + 5 + 15 + 24 + 27 + 42 = 173$.

arise during the algorithm are broken arbitrarily. We redraw the tree and order the children of a node as left and right child arbitrarily, as shown in Figure 9.

The algorithm works with a collection N of nodes which are the roots of the trees constructed so far. Initially, each leaf is a tree by itself. We denote the weight of a node by $w(\mu)$ and use a function EXTRACTMIN that returns the node with the smallest weight and, at the same time, removes this node from the collection.

```
Tree HUFFMAN
  loop μ = EXTRACTMIN(N);
       if N = ∅ then return μ endif;
       ν = EXTRACTMIN(N);
       create node κ with children μ and ν
          and weight w(κ) = w(μ) + w(ν);
       add κ to N
  forever.
```

Straightforward implementations use an array or a linked list and take time O(n) for each operation involving N. There are fewer than $2n$ extractions of the minimum and fewer than n additions, which implies that the total running time is O(n^2). We will see later that there are better ways to implement N leading to running time O($n \log n$).

An inequality. We prepare the proof that the Huffman tree indeed minimizes the weighted external path length. Let T be a full binary tree with weighted external path length $P(T)$. Let $\Lambda(T)$ be the set of leaves and let μ and ν be any two leaves with smallest weights. Then we can construct a new tree T' with

(1) set of leaves $\Lambda(T') = (\Lambda(T) - \{\mu, \nu\}) \dot\cup \{\kappa\}$,

(2) $w(\kappa) = w(\mu) + w(\nu)$,

(3) $P(T') \leq P(T) - w(\mu) - w(\nu)$, with equality if μ and ν are siblings.

We now argue that T' really exists. If μ and ν are siblings then we construct T' from T by removing μ and ν and declaring their parent, κ, as the new leaf. Then

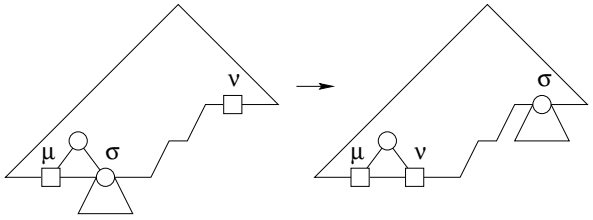

Figure 10: The increase in the depth of ν is compensated by the decrease in depth of the leaves in the subtree of σ.

$$\begin{aligned} P(T') &= P(T) - w(\mu)\delta - w(\nu)\delta + w(\kappa)(\delta - 1) \\ &= P(T) - w(\mu) - w(\nu), \end{aligned}$$

where $\delta = \delta(\mu) = \delta(\nu) = \delta(\kappa) + 1$ is the common depth of μ and ν. Otherwise, assume $\delta(\mu) \geq \delta(\nu)$ and let σ be

the sibling of μ, which may or may not be a leaf. Exchange ν and σ. Since the length of the path from the root to σ is at least as long as the path to μ, the weighted external path length can only decrease; see Figure 10. Then do the same as in the other case.

Proof of optimality. The optimality of the Huffman tree can now be proved by induction.

HUFFMAN TREE THEOREM. Let T be the Huffman tree and X another tree with the same set of leaves and weights. Then $P(T) \leq P(X)$.

PROOF. If there are only two leaves then the claim is obvious. Otherwise, let μ and ν be the two leaves selected by the algorithm. Construct trees T' and X' with

$$\begin{aligned} P(T') &= P(T) - w(\mu) - w(\nu), \\ P(X') &\leq P(X) - w(\mu) - w(\nu). \end{aligned}$$

T' is the Huffman tree for $n-1$ leaves so we can use the inductive assumption and get $P(T') \leq P(X')$. It follows that

$$\begin{aligned} P(T) &= P(T') + w(\mu) + w(\nu) \\ &\leq P(X') + w(\mu) + w(\nu) \\ &\leq P(X). \end{aligned}$$

□

Huffman codes are binary codes that correspond to Huffman trees as described. They are commonly used to compress text and other information. Although Huffman codes are optimal in the sense defined above, there are other codes that are also sensitive to the frequency of sequences of letters and this way outperform Huffman codes for general text.

Summary. The greedy algorithm for constructing Huffman trees works bottom-up by stepwise merging, rather than top-down by stepwise partitioning. If we run the greedy algorithm backwards, it becomes very similar to dynamic programming, except that it pursues only one of many possible partitions. Often this implies that it leads to suboptimal solutions. Nevertheless, there are problems that exhibit enough structure that the greedy algorithm succeeds in finding an optimum, and the scheduling and coding problems described above are two such examples.

First Homework Assignment

Write the solution to each problem on a single page. The deadline for handing in solutions is September 18.

Problem 1. (20 points). Consider two sums, $X = x_1 + x_2 + \ldots + x_n$ and $Y = y_1 + y_2 + \ldots + y_m$. Give an algorithm that finds indices i and j such that swapping x_i with y_j makes the two sums equal, that is, $X - x_i + y_j = Y - y_j + x_i$, if they exist. Analyze your algorithm. (You can use sorting as a subroutine. The amount of credit depends on the correctness of the analysis and the running time of your algorithm.)

Problem 2. ($20 = 10 + 10$ points). Consider distinct items $x_1, x_2, \ldots, x_n$ with positive weights $w_1, w_2, \ldots, w_n$ such that $\sum_{i=1}^{n} w_i = 1.0$. The *weighted median* is the item x_k that satisfies

$$\sum_{x_i < x_k} w_i < 0.5 \quad \text{and} \quad \sum_{x_j > x_k} w_j \leq 0.5.$$

(a) Show how to compute the weighted median of n items in worst-case time O($n \log n$) using sorting.

(b) Show how to compute the weighted median in worst-case time O(n) using a linear-time median algorithm.

Problem 3. ($20 = 6 + 14$ points). A game-board has n columns, each consisting of a top number, the cost of visiting the column, and a bottom number, the maximum number of columns you are allowed to jump to the right. The top number can be any positive integer, while the bottom number is either 1, 2, or 3. The objective is to travel from the first column off the board, to the right of the nth column. The cost of a game is the sum of the costs of the visited columns.

Assuming the board is represented in a two-dimensional array, $B[2, n]$, the following recursive procedure computes the cost of the cheapest game:

```
int CHEAPEST(int i)
  if i > n then return 0 endif;
  x = B[1,i] + CHEAPEST(i+1);
  y = B[1,i] + CHEAPEST(i+2);
  z = B[1,i] + CHEAPEST(i+3);
  case B[2,i] = 1: return x;
       B[2,i] = 2: return min{x,y};
       B[2,i] = 3: return min{x,y,z}
  endcase.
```

(a) Analyze the asymptotic running time of the procedure.

(b) Describe and analyze a more efficient algorithm for finding the cheapest game.

Problem 4. ($20 = 10 + 10$ points). Consider a set of n intervals $[a_i, b_i]$ that cover the unit interval, that is, $[0, 1]$ is contained in the union of the intervals.

(a) Describe an algorithm that computes a minimum subset of the intervals that also covers $[0, 1]$.

(b) Analyze the running time of your algorithm.

(For question (b) you get credit for the correctness of your analysis but also for the running time of your algorithm. In other words, a fast algorithm earns you more points than a slow algorithm.)

Problem 5. ($20 = 7 + 7 + 6$ points). Let $A[1..m]$ and $B[1..n]$ be two strings.

(a) Modify the dynamic programming algorithm for computing the edit distance between A and B for the case in which there are only two allowed operations, insertions and deletions of individual letters.

(b) A (not necessarily contiguous) *subsequence* of A is defined by the increasing sequence of its indices, $1 \leq i_1 < i_2 < \ldots < i_k \leq m$. Use dynamic programming to find the longest common subsequence of A and B and analyze its running time.

(c) What is the relationship between the edit distance defined in (a) and the longest common subsequence computed in (b)?

II SEARCHING

6 Binary Search Trees

One of the purposes of sorting is to facilitate fast searching. However, while a sorted sequence stored in a linear array is good for searching, it is expensive to add and delete items. Binary search trees give you the best of both worlds: fast search and fast update.

Definitions and terminology. We begin with a recursive definition of the most common type of tree used in algorithms. A *(rooted) binary tree* is either empty or a node (the *root*) with a binary tree as left subtree and binary tree as right subtree. We store items in the nodes of the tree. It is often convenient to say the items *are* the nodes. A binary tree is sorted if each item is between the smaller or equal items in the left subtree and the larger or equal items in the right subtree. For example, the tree illustrated in Figure 11 is sorted assuming the usual ordering of English characters. Terms for relations between family members such as *child*, *parent*, *sibling* are also used for nodes in a tree. Every node has one parent, except the root which has no parent. A *leaf* or *external node* is one without children; all other nodes are *internal*. A node ν is a *descendent* of μ if $\nu = \mu$ or ν is a descendent of a child of μ. Symmetrically, μ is an *ancestor* of ν if ν is a descendent of μ. The *subtree* of μ consists of all descendents of μ. An *edge* is a parent-child pair.

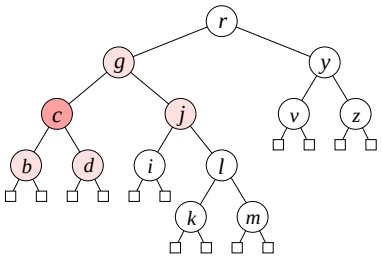

Figure 11: The parent, sibling and two children of the dark node are shaded. The internal nodes are drawn as circles while the leaves are drawn as squares.

The *size* of the tree is the number of nodes. A binary tree is *full* if every internal node has two children. Every full binary tree has one more leaf than internal node. To count its edges, we can either count 2 for each internal node or 1 for every node other than the root. Either way, the total number of edges is one less than the size of the tree. A *path* is a sequence of contiguous edges without repetitions. Usually we only consider paths that descend or paths that ascend. The *length* of a path is the number of edges. For every node μ, there is a unique path from the root to μ. The length of that path is the *depth* of μ. The *height* of the tree is the maximum depth of any node. The *path length* is the sum of depths over all nodes, and the *external path length* is the same sum restricted to the leaves in the tree.

Searching. A *binary search tree* is a sorted binary tree. We assume each node is a record storing an item and pointers to two children:

```
struct Node{item info; Node *ℓ, *r};
typedef Node *Tree.
```

Sometimes it is convenient to also store a pointer to the parent, but for now we will do without. We can search in a binary search tree by tracing a path starting at the root.

```
Node *SEARCH(Tree ϱ, item x)
  case ϱ = NULL: return NULL;
       x < ϱ → info: return SEARCH(ϱ → ℓ, x);
       x = ϱ → info: return ϱ;
       x > ϱ → info: return SEARCH(ϱ → r, x)
  endcase.
```

The running time depends on the length of the path, which is at most the height of the tree. Let n be the size. In the worst case the tree is a linked list and searching takes time $O(n)$. In the best case the tree is perfectly balanced and searching takes only time $O(\log n)$.

Insert. To add a new item is similarly straightforward: follow a path from the root to a leaf and replace that leaf by a new node storing the item. Figure 12 shows the tree obtained after adding w to the tree in Figure 11. The running time depends again on the length of the path. If the insertions come in a random order then the tree is usually

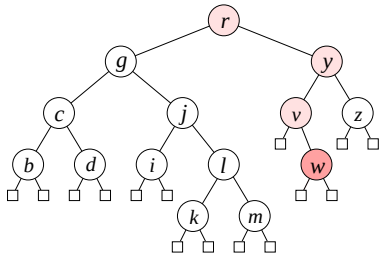

Figure 12: The shaded nodes indicate the path from the root we traverse when we insert w into the sorted tree.

close to being perfectly balanced. Indeed, the tree is the same as the one that arises in the analysis of quicksort. The expected number of comparisons for a (successful) search is one n-th of the expected running time of quicksort, which is roughly $2 \ln n$.

Delete. The main idea for deleting an item is the same as for inserting: follow the path from the root to the node ν that stores the item.

`Case 1.` ν has no internal node as a child. Remove ν.

`Case 2.` ν has one internal child. Make that child the child of the parent of ν.

`Case 3.` ν has two internal children. Find the rightmost internal node in the left subtree, remove it, and substitute it for ν, as shown in Figure 13.

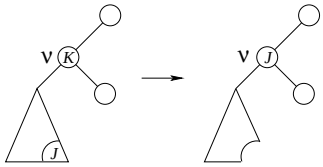

Figure 13: Store J in ν and delete the node that used to store J.

The analysis of the expected search time in a binary search tree constructed by a random sequence of insertions and deletions is considerably more challenging than if no deletions are present. Even the definition of a random sequence is ambiguous in this case.

Optimal binary search trees. Instead of hoping the incremental construction yields a shallow tree, we can construct the tree that minimizes the search time. We consider the common problem in which items have different probabilities to be the target of a search. For example, some words in the English dictionary are more commonly searched than others and are therefore assigned a higher probability. Let $a_1 < a_2 < \ldots < a_n$ be the items and p_i the corresponding probabilities. To simplify the discussion, we only consider successful searches and thus assume $\sum_{i=1}^{n} p_i = 1$. The expected number of comparisons for a successful search in a binary search tree T storing the n items is

$$\begin{aligned} 1 + C(T) &= \sum_{i=1}^{n} p_i \cdot (\delta_i + 1) \\ &= 1 + \sum_{i=1}^{n} p_i \cdot \delta_i, \end{aligned}$$

where δ_i is the depth of the node that stores a_i. $C(T)$ is the *weighted path length* or the *cost* of T. We study the problem of constructing a tree that minimizes the cost. To develop an example, let $n = 3$ and $p_1 = \frac{1}{2}$, $p_2 = \frac{1}{3}$, $p_3 = \frac{1}{6}$. Figure 14 shows the five binary trees with three nodes and states their costs. It can be shown that the

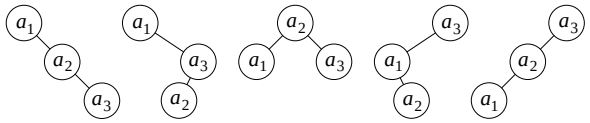

Figure 14: There are five different binary trees of three nodes. From left to right their costs are $\frac{2}{3}, \frac{5}{6}, \frac{2}{3}, \frac{7}{6}, \frac{4}{3}$. The first tree and the third tree are both optimal.

number of different binary trees with n nodes is $\frac{1}{n+1}\binom{2n}{n}$, which is exponential in n. This is far too large to try all possibilities, so we need to look for a more efficient way to construct an optimum tree.

Dynamic programming. We write T_i^j for the optimum weighted binary search tree of $a_i, a_{i+1}, \ldots, a_j$, C_i^j for its cost, and $p_i^j = \sum_{k=i}^{j} p_k$ for the total probability of the items in T_i^j. Suppose we know that the optimum tree stores item a_k in its root. Then the left subtree is T_i^{k-1} and the right subtree is T_{k+1}^j. The cost of the optimum tree is therefore $C_i^j = C_i^{k-1} + C_{k+1}^j + p_i^j - p_k$. Since we do not know which item is in the root, we try all possibilities and find the minimum:

$$C_i^j = \min_{i \le k \le j} \{C_i^{k-1} + C_{k+1}^j + p_i^j - p_k\}.$$

This formula can be translated directly into a dynamic programming algorithm. We use three two-dimensional arrays, one for the sums of probabilities, p_i^j, one for the costs of optimum trees, C_i^j, and one for the indices of the items stored in their roots, R_i^j. We assume that the first array has already been computed. We initialize the other two arrays along the main diagonal and add one dummy diagonal for the cost.

```
for k = 1 to n do
  C[k, k−1] = C[k, k] = 0;  R[k, k] = k
endfor;
C[n+1, n] = 0.
```

We fill the rest of the two arrays one diagonal at a time.

```
for ℓ = 2 to n do
 for i = 1 to n − ℓ + 1 do
   j = i + ℓ − 1;  C[i, j] = ∞;
   for k = i to j do
     cost = C[i, k−1] + C[k+1, j]
          + p[i, j] − p[k, k];
     if cost < C[i, j] then
       C[i, j] = cost;  R[i, j] = k
     endif
   endfor
 endfor
endfor.
```

The main part of the algorithm consists of three nested loops each iterating through at most n values. The running time is therefore in $O(n^3)$.

Example. Table 1 shows the partial sums of probabilities for the data in the earlier example. Table 2 shows

$6p$	1	2	3
1	3	5	6
2		2	3
3			1

Table 1: Six times the partial sums of probabilities used by the dynamic programming algorithm.

the costs and the indices of the roots of the optimum trees computed for all contiguous subsequences. The optimum

$6C$	1	2	3
1	0	2	4
2		0	1
3			0

R	1	2	3
1	1	1	1
2		2	2
3			3

Table 2: Six times the costs and the roots of the optimum trees.

tree can be constructed from R as follows. The root stores the item with index $R[1, 3] = 1$. The left subtree is therefore empty and the right subtree stores a_2, a_3. The root of the optimum right subtree stores the item with index $R[2, 3] = 2$. Again the left subtree is empty and the right subtree consists of a single node storing a_3.

Improved running time. Notice that the array R in Table 2 is monotonic, both along rows and along columns. Indeed it is possible to prove $R_i^{j-1} \leq R_i^j$ in every row and $R_i^j \leq R_{i+1}^j$ in every column. We omit the proof and show how the two inequalities can be used to improve the dynamic programming algorithm. Instead of trying all roots from i through j we restrict the innermost `for`-loop to

$$\texttt{for } k = R[i, j-1] \texttt{ to } R[i+1, j] \texttt{ do}$$

The monotonicity property implies that this change does not alter the result of the algorithm. The running time of a single iteration of the outer `for`-loop is now

$$U_\ell(n) \quad = \quad \sum_{i=1}^{n-\ell+1} (R_{i+1}^j - R_i^{j-1} + 1).$$

Recall that $j = i + \ell - 1$ and note that most terms cancel, giving

$$\begin{aligned} U_\ell(n) &= R_{n-\ell+2}^n - R_1^{\ell-1} + (n - \ell + 1) \\ &\leq 2n. \end{aligned}$$

In words, each iteration of the outer `for`-loop takes only time $O(n)$, which implies that the entire algorithm takes only time $O(n^2)$.

7 Red-Black Trees

Binary search trees are an elegant implementation of the *dictionary* data type, which requires support for

```
item SEARCH (item),
void INSERT (item),
void DELETE (item),
```

and possible additional operations. Their main disadvantage is the worst case time $\Omega(n)$ for a single operation. The reasons are insertions and deletions that tend to get the tree unbalanced. It is possible to counteract this tendency with occasional local restructuring operations and to guarantee logarithmic time per operation.

2-3-4 trees. A special type of balanced tree is the 2-3-4 *tree*. Each internal node stores one, two, or three items and has two, three, or four children. Each leaf has the same depth. As shown in Figure 15, the items in the internal nodes separate the items stored in the subtrees and thus facilitate fast searching. In the smallest 2-3-4 tree of

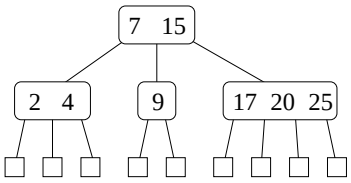

Figure 15: A 2-3-4 tree of height two. All items are stored in internal nodes.

height h, every internal node has exactly two children, so we have 2^h leaves and $2^h - 1$ internal nodes. In the largest 2-3-4 tree of height h, every internal node has four children, so we have 4^h leaves and $(4^h - 1)/3$ internal nodes. We can store a 2-3-4 tree in a binary tree by expanding a node with $i > 1$ items and $i + 1$ children into i nodes each with one item, as shown in Figure 16.

Red-black trees. Suppose we color each edge of a binary search tree either red or black. The color is conveniently stored in the lower node of the edge. Such a edge-colored tree is a *red-black tree* if

(1) there are no two consecutive red edges on any descending path and every maximal such path ends with a black edge;

(2) all maximal descending paths have the same number of black edges.

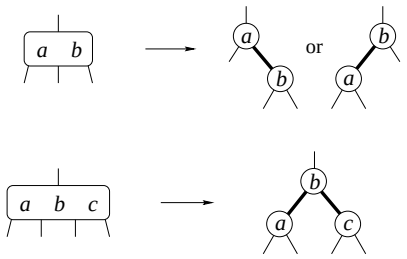

Figure 16: Transforming a 2-3-4 tree into a binary tree. Bold edges are called red and the others are called black.

The number of black edges on a maximal descending path is the *black height*, denoted as $bh(\varrho)$. When we transform a 2-3-4 tree into a binary tree as in Figure 16, we get a red-black tree. The result of transforming the tree in Figure 15

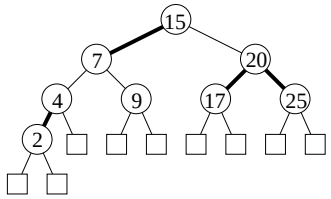

Figure 17: A red-black tree obtained from the 2-3-4 tree in Figure 15.

is shown in Figure 17.

HEIGHT LEMMA. A red-black tree with n internal nodes has height at most $2\log_2(n+1)$.

PROOF. The number of leaves is $n + 1$. Contract each red edge to get a 2-3-4 tree with $n + 1$ leaves. Its height is $h \leq \log_2(n+1)$. We have $bh(\varrho) = h$, and by Rule (1) the height of the red-black tree is at most $2bh(\varrho) \leq 2\log_2(n+1)$. □

Rotations. Restructuring a red-black tree can be done with only one operation (and its symmetric version): a *rotation* that moves a subtree from one side to another, as shown in Figure 18. The ordered sequence of nodes in the left tree of Figure 18 is

$$\ldots, \text{order}(A), \nu, \text{order}(B), \mu, \text{order}(C), \ldots,$$

and this is also the ordered sequence of nodes in the right tree. In other words, a rotation maintains the ordering. Function ZIG below implements the right rotation:

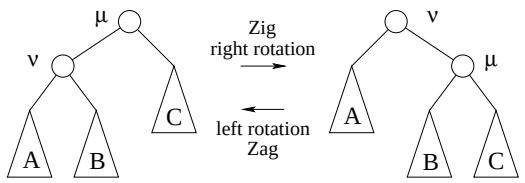

Figure 18: From left to right a right rotation and from right to left a left rotation.

```
Node * ZIG(Node * μ)
  assert μ ≠ NULL and ν = μ → ℓ ≠ NULL;
  μ → ℓ = ν → r; ν → r = μ; return ν.
```

Function ZAG is symmetric and performs a left rotation. Occasionally, it is necessary to perform two rotations in sequence, and it is convenient to combine them into a single operation referred to as a *double rotation*, as shown in Figure 19. We use a function ZIGZAG to implement a

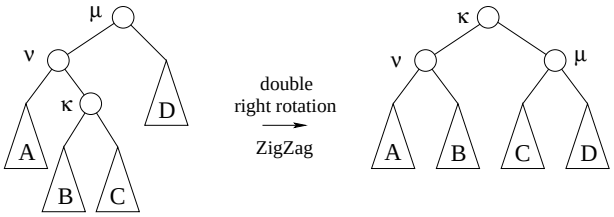

Figure 19: The double right rotation at μ is the concatenation of a single left rotation at ν and a single right rotation at μ.

double right rotation and the symmetric function ZAGZIG to implement a double left rotation.

```
Node * ZIGZAG(Node * μ)
  μ → ℓ = ZAG(μ → ℓ); return ZIG(μ).
```

The double right rotation is the composition of two single rotations: $\text{ZIGZAG}(\mu) = \text{ZIG}(\mu) \circ \text{ZAG}(\nu)$. Remember that the composition of functions is written from right to left, so the single left rotation of ν precedes the single right rotation of μ. Single rotations preserve the ordering of nodes and so do double rotations.

Insertion. Before studying the details of the restructuring algorithms for red-black trees, we look at the trees that arise in a short insertion sequence, as shown in Figure 20. After adding 10, 7, 13, 4, we have two red edges in sequence and repair this by promoting 10 (A). After adding 2, we repair the two red edges in sequence by a single rotation of 7 (B). After adding 5, we promote 4 (C), and after adding 6, we do a double rotation of 7 (D).

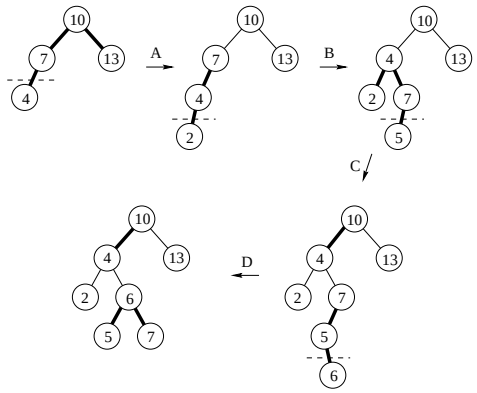

Figure 20: Sequence of red-black trees generated by inserting the items 10, 7, 13, 4, 2, 5, 6 in this sequence.

An item x is added by substituting a new internal node for a leaf at the appropriate position. To satisfy Rule (2) of the red-black tree definition, color the incoming edge of the new node red, as shown in Figure 21. Start the

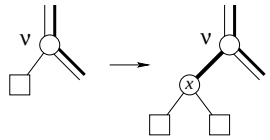

Figure 21: The incoming edge of a newly added node is always red.

adjustment of color and structure at the parent ν of the new node. We state the properties maintained by the insertion algorithm as invariants that apply to a node ν traced by the algorithm.

INVARIANT I. The only possible violation of the red-black tree properties is that of Rule (1) at the node ν, and if ν has a red incoming edge then it has exactly one red outgoing edge.

Observe that Invariant I holds right after adding x. We continue with the analysis of all the cases that may arise. The local adjustment operations depend on the neighborhood of ν.

`Case 1.` The incoming edge of ν is black. Done.

`Case 2.` The incoming edge of ν is red. Let μ be the parent of ν and assume ν is left child of μ.

`Case 2.1.` Both outgoing edges of μ are red, as in Figure 22. Promote μ. Let ν be the parent of μ and recurse.

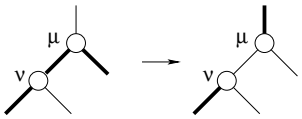

Figure 22: Promotion of μ. (The colors of the outgoing edges of ν may be the other way round).

`Case 2.2.` Only one outgoing edge of μ is red, namely the one from μ to ν.

`Case 2.2.1.` The left outgoing edge of ν is red, as in Figure 23 to the left. Right rotate μ. Done.

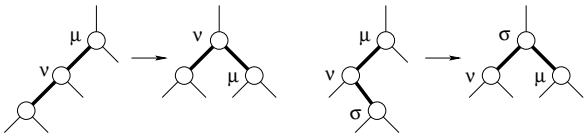

Figure 23: Right rotation of μ to the left and double right rotation of μ to the right.

`Case 2.2.2.` The right outgoing edge of ν is red, as in Figure 23 to the right. Double right rotate μ. Done.

Case 2 has a symmetric case where left and right are interchanged. An insertion may cause logarithmically many promotions but at most two rotations.

Deletion. First find the node π that is to be removed. If necessary, we substitute the inorder successor for π so we can assume that both children of π are leaves. If π is last in inorder we substitute symmetrically. Replace π by a leaf ν, as shown in Figure 24. If the incoming edge of π is red then change it to black. Otherwise, remember the incoming edge of ν as 'double-black', which counts as two black edges. Similar to insertions, it helps to understand the deletion algorithm in terms of a property it maintains.

INVARIANT D. The only possible violation of the red-black tree properties is a double-black incoming edge of ν.

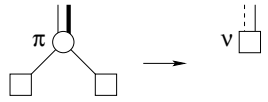

Figure 24: Deletion of node π. The dashed edge counts as two black edges when we compute the black depth.

Note that Invariant D holds right after we remove π. We now present the analysis of all the possible cases. The adjustment operation is chosen depending on the local neighborhood of ν.

`Case 1.` The incoming edge of ν is black. Done.

`Case 2.` The incoming edge of ν is double-black. Let μ be the parent and κ the sibling of ν. Assume ν is left child of μ and note that κ is internal.

`Case 2.1.` The edge from μ to κ is black.

`Case 2.1.1.` Both outgoing edges of κ are black, as in Figure 25. Demote μ. Recurse for $\nu = \mu$.

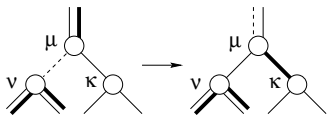

Figure 25: Demotion of μ.

`Case 2.1.2.` The right outgoing edge of κ is red, as in Figure 26 to the left. Change the color of that edge to black and left rotate μ. Done.

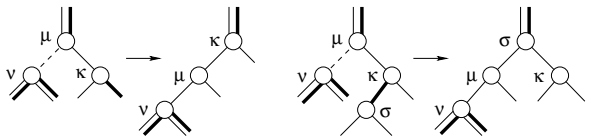

Figure 26: Left rotation of μ to the left and double left rotation of μ to the right.

`Case 2.1.3.` The right outgoing edge of κ is black, as in Figure 26 to the right. Change the color of the left outgoing edge to black and double left rotate μ. Done.

`Case 2.2.` The edge from μ to κ is red, as in Figure 27. Left rotate μ. Recurse for ν.

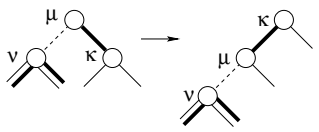

Figure 27: Left rotation of μ.

Case 2 has a symmetric case in which ν is the right child of μ. Case 2.2 seems problematic because it recurses without moving ν any closer to the root. However, the configuration excludes the possibility of Case 2.2 occurring again. If we enter Cases 2.1.2 or 2.1.3 then the termination is immediate. If we enter Case 2.1.1 then the termination follows because the incoming edge of μ is red. The deletion may cause logarithmically many demotions but at most three rotations.

Summary. The red-black tree is an implementation of the dictionary data type and supports the operations search, insert, delete in logarithmic time each. An insertion or deletion requires the equivalent of at most three single rotations. The red-black tree also supports finding the minimum, maximum and the inorder successor, predecessor of a given node in logarithmic time each.

8 Amortized Analysis

Amortization is an analysis technique that can influence the design of algorithms in a profound way. Later in this course, we will encounter data structures that owe their very existence to the insight gained in performance due to amortized analysis.

Binary counting. We illustrate the idea of amortization by analyzing the cost of counting in binary. Think of an integer as a linear array of bits, $n = \sum_{i \geq 0} A[i] \cdot 2^i$. The following loop keeps incrementing the integer stored in A.

```
loop i = 0;
  while A[i] = 1 do A[i] = 0; i++ endwhile;
  A[i] = 1.
forever.
```

We define the *cost* of counting as the total number of bit changes that are needed to increment the number one by one. What is the cost to count from 0 to n? Figure 28 shows that counting from 0 to 15 requires 26 bit changes. Since n takes only $1 + \lfloor \log_2 n \rfloor$ bits or positions in A,

Figure 28: The numbers are written vertically from top to bottom. The boxed bits change when the number is incremented.

a single increment does at most $2 + \log_2 n$ steps. This implies that the cost of counting from 0 to n is at most $n \log_2 n + 2n$. Even though the upper bound of $2 + \log_2 n$ is almost tight for the worst single step, we can show that the total cost is much less than n times that. We do this with two slightly different amortization methods referred to as aggregation and accounting.

Aggregation. The aggregation method takes a global view of the problem. The pattern in Figure 28 suggests we define b_i equal to the number of 1s and t_i equal to the number of trailing 1s in the binary notation of i. Every other number has no trailing 1, every other number of the remaining ones has one trailing 1, etc. Assuming $n = 2^k - 1$, we therefore have exactly $j - 1$ trailing 1s for $2^{k-j} = (n+1)/2^j$ integers between 0 and $n - 1$. The total number of bit changes is therefore

$$T(n) \;=\; \sum_{i=0}^{n-1} (t_i + 1) \;=\; (n+1) \cdot \sum_{j=1}^{k} \frac{j}{2^j}.$$

We use index transformation to show that the sum on the right is less than 2:

$$\begin{aligned} \sum_{j \geq 1} \frac{j}{2^j} &= \sum_{j \geq 1} \frac{j-1}{2^{j-1}} \\ &= 2 \cdot \sum_{j \geq 1} \frac{j}{2^j} - \sum_{j \geq 1} \frac{1}{2^{j-1}} \\ &= 2. \end{aligned}$$

Hence the cost is $T(n) < 2(n+1)$. The *amortized cost* per operation is $\frac{T(n)}{n}$, which is about 2.

Accounting. The idea of the accounting method is to charge each operation what we think its amortized cost is. If the amortized cost exceeds the actual cost, then the surplus remains as a credit associated with the data structure. If the amortized cost is less than the actual cost, the accumulated credit is used to pay for the cost overflow. Define the amortized cost of a bit change $0 \to 1$ as \$2 and that of $1 \to 0$ as \$0. When we change 0 to 1 we pay \$1 for the actual expense and \$1 stays with the bit, which is now 1. This \$1 pays for the (later) cost of changing the 1 to 0. Each increment has amortized cost \$2, and together with the money in the system, this is enough to pay for all the bit changes. The cost is therefore at most $2n$.

We see how a little trick, like making the $0 \to 1$ changes pay for the $1 \to 0$ changes, leads to a very simple analysis that is even more accurate than the one obtained by aggregation.

Potential functions. We can further formalize the amortized analysis by using a potential function. The idea is similar to accounting, except there is no explicit credit saved anywhere. The accumulated credit is an expression of the well-being or potential of the data structure. Let c_i be the actual cost of the i-th operation and D_i the data structure after the i-th operation. Let $\Phi_i = \Phi(D_i)$ be the potential of D_i, which is some numerical value depending on the concrete application. Then we define $a_i = c_i + \Phi_i - \Phi_{i-1}$ as the *amortized cost* of the i-th

operation. The sum of amortized costs of n operations is

$$\begin{aligned}\sum_{i=1}^{n} a_i &= \sum_{i=1}^{n}(c_i + \Phi_i - \Phi_{i-1}) \\ &= \sum_{i=1}^{n} c_i + \Phi_n - \Phi_0.\end{aligned}$$

We aim at choosing the potential such that $\Phi_0 = 0$ and $\Phi_n \geq 0$ because then we get $\sum a_i \geq \sum c_i$. In words, the sum of amortized costs covers the sum of actual costs. To apply the method to binary counting we define the potential equal to the number of 1s in the binary notation, $\Phi_i = b_i$. It follows that

$$\begin{aligned}\Phi_i - \Phi_{i-1} &= b_i - b_{i-1} \\ &= (b_{i-1} - t_{i-1} + 1) - b_{i-1} \\ &= 1 - t_{i-1}.\end{aligned}$$

The actual cost of the i-th operation is $c_i = 1 + t_{i-1}$, and the amortized cost is $a_i = c_i + \Phi_i - \Phi_{i-1} = 2$. We have $\Phi_0 = 0$ and $\Phi_n \geq 0$ as desired, and therefore $\sum c_i \leq \sum a_i = 2n$, which is consistent with the analysis of binary counting with the aggregation and the accounting methods.

2-3-4 trees. As a more complicated application of amortization we consider 2-3-4 trees and the cost of restructuring them under insertions and deletions. We have seen 2-3-4 trees earlier when we talked about red-black trees. A set of keys is stored in sorted order in the internal nodes of a 2-3-4 tree, which is characterized by the following rules:

(1) each internal node has $2 \leq d \leq 4$ children and stores $d - 1$ keys;

(2) all leaves have the same depth.

As for binary trees, being sorted means that the left-to-right order of the keys is sorted. The only meaningful definition of this ordering is the ordered sequence of the first subtree followed by the first key stored in the root followed by the ordered sequence of the second subtree followed by the second key, etc.

To insert a new key, we attach a new leaf and add the key to the parent ν of that leaf. All is fine unless ν overflows because it now has five children. If it does, we repair the violation of Rule (1) by climbing the tree one node at a time. We call an internal node *non-saturated* if it has fewer than four children.

`Case 1.` ν has five children and a non-saturated sibling to its left or right. Move one child from ν to that sibling, as in Figure 29.

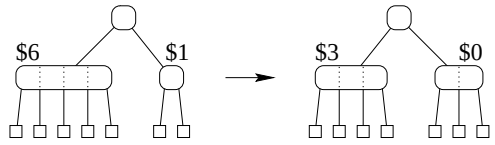

Figure 29: The overflowing node gives one child to a non-saturated sibling.

`Case 2.` ν has five children and no non-saturated sibling. Split ν into two nodes and recurse for the parent of ν, as in Figure 30. If ν has no parent then create a new root whose only children are the two nodes obtained from ν.

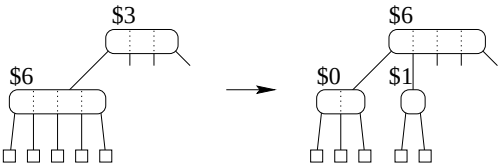

Figure 30: The overflowing node is split into two and the parent is treated recursively.

Deleting a key is done is a similar fashion, although there we have to battle with nodes ν that have too few children rather than too many. Let ν have only one child. We repair Rule (1) by adopting a child from a sibling or by merging ν with a sibling. In the latter case the parent of ν looses a child and needs to be visited recursively. The two operations are illustrated in Figures 31 and 32.

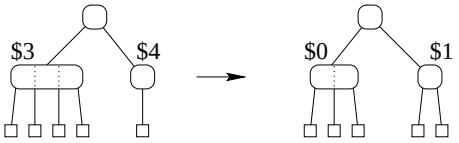

Figure 31: The underflowing node receives one child from a sibling.

Amortized analysis. The worst case for inserting a new key occurs when all internal nodes are saturated. The insertion then triggers logarithmically many splits. Symmetrically, the worst case for a deletion occurs when all

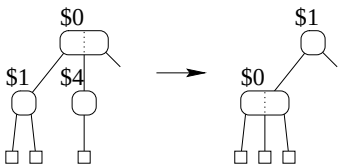

Figure 32: The underflowing node is merged with a sibling and the parent is treated recursively.

internal nodes have only two children. The deletion then triggers logarithmically many mergers. Nevertheless, we can show that in the amortized sense there are at most a constant number of split and merge operations per insertion and deletion.

We use the accounting method and store money in the internal nodes. The best internal nodes have three children because then they are flexible in both directions. They require no money, but all other nodes are given a positive amount to pay for future expenses caused by split and merge operations. Specifically, we store \$4, \$1, \$0, \$3, \$6 in each internal node with 1, 2, 3, 4, 5 children. As illustrated in Figures 29 and 31, an adoption moves money only from ν to its sibling. The operation keeps the total amount the same or decreases it, which is even better. As shown in Figure 30, a split frees up \$5 from ν and spends at most \$3 on the parent. The extra \$2 pay for the split operation. Similarly, a merger frees \$5 from the two affected nodes and spends at most \$3 on the parent. This is illustrated in Figure 32. An insertion makes an initial investment of at most \$3 to pay for creating a new leaf. Similarly, a deletion makes an initial investment of at most \$3 for destroying a leaf. If we charge $\$2$ for each split and each merge operation, the money in the system suffices to cover the expenses. This implies that for n insertions and deletions we get a total of at most $\frac{3n}{2}$ split and merge operations. In other words, the amortized number of split and merge operations is at most $\frac{3}{2}$.

Recall that there is a one-to-one correspondence between 2-3-4 tree and red-black trees. We can thus translate the above update procedure and get an algorithm for red-black trees with an amortized constant restructuring cost per insertion and deletion. We already proved that for red-black trees the number of rotations per insertion and deletion is at most a constant. The above argument implies that also the number of promotions and demotions is at most a constant, although in the amortized and not in the worst-case sense as for the rotations.

9 Splay Trees

Splay trees are similar to red-black trees except that they guarantee good shape (small height) only on the average. They are simpler to code than red-black trees and have the additional advantage of giving faster access to items that are more frequently searched. The reason for both is that splay trees are self-adjusting.

Self-adjusting binary search trees. Instead of explicitly maintaining the balance using additional information (such as the color of edges in the red-black tree), splay trees maintain balance implicitly through a self-adjusting mechanism. Good shape is a side-effect of the operations that are applied. These operations are applied while *splaying* a node, which means moving it up to the root of the tree, as illustrated in Figure 33. A detailed analysis will

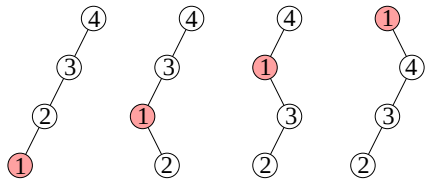

Figure 33: The node storing 1 is splayed using three single rotations.

reveal that single rotations do not imply good amortized performance but combinations of single rotations in pairs do. Aside from double rotations, we use *roller-coaster rotations* that compose two single left or two single right rotations, as shown in Figure 35. The sequence of the two single rotations is important, namely first the higher then the lower node. Recall that $\textsc{Zig}(\kappa)$ performs a single right rotation and returns the new root of the rotated subtree. The roller-coaster rotation to the right is then

```
Node * ZigZig(Node * κ)
  return Zig(Zig(κ)).
```

Function ZAGZAG is symmetric, exchanging left and right, and functions ZIGZAG and ZAGZIG are the two double rotations already used for red-black trees.

Splay. A splay operation finds an item and uses rotations to move the corresponding node up to the root position. Whenever possible, a double rotation or a roller-coaster rotation is used. We dispense with special cases and show

Function SPLAY for the case the search item x is less than the item in the root.

```
if x < ϱ → info then μ = ϱ → ℓ;
  if x < μ → info then
    μ → ℓ = Splay(μ → ℓ, x);
    return ZigZig(ϱ)
  elseif x > μ → info then
    μ → r = Splay(μ → r, x);
    return ZigZag(ϱ)
  else
    return Zig(ϱ)
endif.
```

If x is stored in one of the children of ϱ then it is moved to the root by a single rotation. Otherwise, it is splayed recursively to the third level and moved to the root either by a double or a roller-coaster rotation. The number of rotation depends on the length of the path from ϱ to x. Specifically, if the path is i edges long then x is splayed in $\lfloor i/2 \rfloor$ double and roller-coaster rotations and zero or one single rotation. In the worst case, a single splay operation takes almost as many rotations as there are nodes in the tree. We will see shortly that the amortized number of rotations is at most logarithmic in the number of nodes.

Amortized cost. Recall that the amortized cost of an operation is the actual cost minus the cost for work put into improving the data structure. To analyze the cost, we use a potential function that measures the well-being of the data structure. We need definitions:

the *size* $s(\nu)$ is the number of descendents of node ν, including ν,

the *balance* $\beta(\nu)$ is twice the floor of the binary logarithm of the size, $\beta(\nu) = 2\lfloor \log_2 s(\nu) \rfloor$,

the *potential* Φ of a tree or a collection of trees is the sum of balances over all nodes, $\Phi = \sum \beta(\nu)$,

the *actual cost* c_i of the i-th splay operation is 1 plus the number of single rotations (counting a double or roller-coaster rotation as two single rotations).

the *amortized cost* a_i of the i-th splay operation is $a_i = c_i + \Phi_i - \Phi_{i-1}$.

We have $\Phi_0 = 0$ for the empty tree and $\Phi_i \geq 0$ in general. This implies that the total actual cost does not exceed the total amortized cost, $\sum c_i = \sum a_i - \Phi_n + \Phi_0 \leq \sum a_i$.

To get a feeling for the potential, we compute Φ for the two extreme cases. Note first that the integral of the

natural logarithm is $\int \ln x = x \ln x - x$ and therefore $\int \log_2 x = x \log_2 x - x/\ln 2$. In the extreme unbalanced case, the balance of the i-th node from the bottom is $2\lfloor \log_2 i \rfloor$ and the potential is

$$\Phi = 2\sum_{i=1}^{n} \lfloor \log_2 i \rfloor = 2n \log_2 n - \mathrm{O}(n).$$

In the balanced case, we bound Φ from above by $2U(n)$, where $U(n) = 2U(\frac{n}{2}) + \log_2 n$. We prove that $U(n) < 2n$ for the case when $n = 2^k$. Consider the perfectly balanced tree with n leaves. The height of the tree is $k = \log_2 n$. We encode the term $\log_2 n$ of the recurrence relation by drawing the hook-like path from the root to the right child and then following left edges until we reach the leaf level. Each internal node encodes one of the recursively surfacing $\log$-terms by a hook-like path starting at that node. The paths are pairwise edge-disjoint, which implies that their total length is at most the number of edges in the tree, which is $2n - 2$.

Investment. The main part of the amortized time analysis is a detailed study of the three types of rotations: single, roller-coaster, and double. We write $\beta(\nu)$ for the balance of a node ν before the rotation and $\beta'(\nu)$ for the balance after the rotation. Let ν be the lowest node involved in the rotation. The goal is to prove that the amortized cost of a roller-coaster and a double rotation is at most $3[\beta'(\nu) - \beta(\nu)]$ each, and that of a single rotation is at most $1 + 3[\beta'(\nu) - \beta(\nu)]$. Summing these terms over the rotations of a splay operation gives a telescoping series in which all terms cancel except the first and the last. To this we add 1 for the at most one single rotation and another 1 for the constant cost in definition of actual cost.

INVESTMENT LEMMA. The amortized cost of splaying a node ν in a tree ϱ is at most $2 + 3[\beta(\varrho) - \beta(\nu)]$.

Before looking at the details of the three types of rotations, we prove that if two siblings have the same balance then their common parent has a larger balance. Because balances are even integers this means that the balance of the parent exceeds the balance of its children by at least 2.

BALANCE LEMMA. If μ has children ν, κ and $\beta(\nu) = \beta(\kappa) = \beta$ then $\beta(\mu) \geq \beta + 2$.

PROOF. By definition $\beta(\nu) = 2\lfloor \log_2 s(\nu) \rfloor$ and therefore $s(\nu) \geq 2^{\beta/2}$. We have $s(\mu) = 1 + s(\nu) + s(\kappa) \geq 2^{1+\beta/2}$, and therefore $\beta(\mu) \geq \beta + 2$. □

Single rotation. The amortized cost of a single rotation shown in Figure 34 is 1 for performing the rotation plus the change in the potential:

$$\begin{aligned} a &= 1 + \beta'(\nu) + \beta'(\mu) - \beta(\nu) - \beta(\mu) \\ &\leq 1 + 3[\beta'(\nu) - \beta(\nu)] \end{aligned}$$

because $\beta'(\mu) \leq \beta(\mu)$ and $\beta(\nu) \leq \beta'(\nu)$.

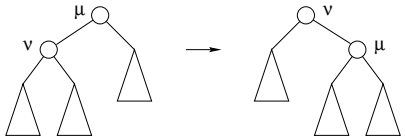

Figure 34: The size of μ decreases and that of ν increases from before to after the rotation.

Roller-coaster rotation. The amortized cost of a roller-coaster rotation shown in Figure 35 is

$$\begin{aligned} a &= 2 + \beta'(\nu) + \beta'(\mu) + \beta'(\kappa) \\ &\quad - \beta(\nu) - \beta(\mu) - \beta(\kappa) \\ &\leq 2 + 2[\beta'(\nu) - \beta(\nu)] \end{aligned}$$

because $\beta'(\kappa) \leq \beta(\kappa)$, $\beta'(\mu) \leq \beta'(\nu)$, and $\beta(\nu) \leq \beta(\mu)$. We distinguish two cases to prove that a is bounded from above by $3[\beta'(\nu) - \beta(\nu)]$. In both cases, the drop in the

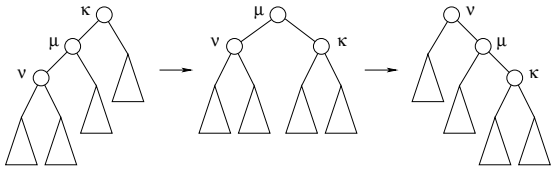

Figure 35: If in the middle tree the balance of ν is the same as the balance of μ then by the Balance Lemma the balance of κ is less than that common balance.

potential pays for the two single rotations.

`Case` $\beta'(\nu) > \beta(\nu)$. The difference between the balance of ν before and after the roller-coaster rotation is at least 2. Hence $a \leq 3[\beta'(\nu) - \beta(\nu)]$.

`Case` $\beta'(\nu) = \beta(\nu) = \beta$. Then the balances of nodes ν and μ in the middle tree in Figure 35 are also equal to β. The Balance Lemma thus implies that the balance of κ in that middle tree is at most $\beta - 2$. But since the balance of κ after the roller-coaster rotation is the same as in the middle tree, we have $\beta'(\kappa) < \beta$. Hence $a \leq 0 = 3[\beta'(\nu) - \beta(\nu)]$.

Double rotation. The amortized cost of a double rotation shown in Figure 36 is

$$\begin{aligned} a &= 2 + \beta'(\nu) + \beta'(\mu) + \beta'(\kappa) \\ &\quad - \beta(\nu) - \beta(\mu) - \beta(\kappa) \\ &\leq 2 + [\beta'(\nu) - \beta(\nu)] \end{aligned}$$

because $\beta'(\kappa) \leq \beta(\kappa)$ and $\beta'(\mu) \leq \beta(\mu)$. We again distinguish two cases to prove that a is bounded from above by $3[\beta'(\nu) - \beta(\nu)]$. In both cases, the drop in the potential pays for the two single rotations.

`Case` $\beta'(\nu) > \beta(\nu)$. The difference is at least 2, which implies $a \leq 3[\beta'(\nu) - \beta(\nu)]$, as before.

`Case` $\beta'(\nu) = \beta(\nu) = \beta$. Then $\beta(\mu) = \beta(\kappa) = \beta$. We have $\beta'(\mu) < \beta'(\nu)$ or $\beta'(\kappa) < \beta'(\nu)$ by the Balance Lemma. Hence $a \leq 0 = 3[\beta'(\nu) - \beta(\nu)]$.

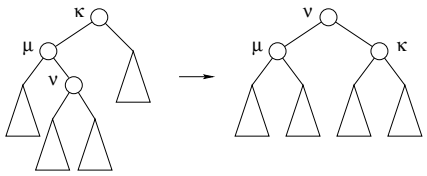

Figure 36: In a double rotation, the sizes of μ and κ decrease from before to after the operation.

Dictionary operations. In summary, we showed that the amortized cost of splaying a node ν in a binary search tree with root ϱ is at most $1 + 3[\beta(\varrho) - \beta(\nu)]$. We now use this result to show that splay trees have good amortized performance for all standard dictionary operations and more.

To **access** an item we first splay it to the root and return the root even if it does not contain x. The amortized cost is $O(\beta(\varrho))$.

Given an item x, we can **split** a splay tree into two, one containing all items smaller than or equal to x and the other all items larger than x, as illustrated in Figure 37. The amortized cost is the amortized cost for splaying plus

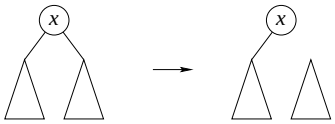

Figure 37: After splaying x to the root, we split the tree by unlinking the right subtree.

the increase in the potential, which we denote as $\Phi' - \Phi$. Recall that the potential of a collection of trees is the sum of the balances of all nodes. Splitting the tree decreases the number of descendents and therefore the balance of the root, which implies that $\Phi' - \Phi < 0$. It follows that the amortized cost of a split operation is less than that of a splay operation and therefore in $O(\beta(\varrho))$.

Two splay trees can be **joined** into one if all items in one tree are smaller than all items in the other tree, as illustrated in Figure 38. The cost for splaying the maximum

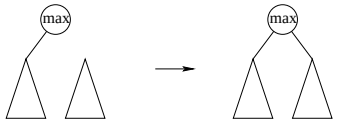

Figure 38: We first splay the maximum in the tree with the smaller items and then link the two trees.

in the first tree is $O(\beta(\varrho_1))$. The potential increase caused by linking the two trees is

$$\begin{aligned} \Phi' - \Phi &\leq 2\lfloor \log_2(s(\varrho_1) + s(\varrho_2)) \rfloor \\ &\leq 2\log_2 s(\varrho_1) + 2\log_2 s(\varrho_2). \end{aligned}$$

The amortized cost of joining is thus $O(\beta(\varrho_1) + \beta(\varrho_2))$.

To **insert** a new item, x, we split the tree. If x is already in the tree, we undo the split operation by linking the two trees. Otherwise, we make the two trees the left and right subtrees of a new node storing x. The amortized cost for splaying is $O(\beta(\varrho))$. The potential increase caused by linking is

$$\begin{aligned} \Phi' - \Phi &\leq 2\lfloor \log_2(s(\varrho_1) + s(\varrho_2) + 1) \rfloor \\ &= \beta(\varrho). \end{aligned}$$

The amortized cost of an insertion is thus $O(\beta(\varrho))$.

To **delete** an item, we splay it to the root, remove the root, and join the two subtrees. Removing x decreases the potential, and the amortized cost of joining the two subtrees is at most $O(\beta(\varrho))$. This implies that the amortized cost of a deletion is at most $O(\beta(\varrho))$.

Weighted search. A nice property of splay trees not shared by most other balanced trees is that they automatically adapt to biased search probabilities. It is plausible that this would be the case because items that are often accessed tend to live at or near the root of the tree. The analysis is somewhat involved and we only state the result. Each item or node has a positive weight, $w(\nu) > 0$,

and we define $W = \sum_\nu w(\nu)$. We have the following generalization of the Investment Lemma, which we state without proof.

WEIGHTED INVESTMENT LEMMA. The amortized cost of splaying a node ν in a tree with total weight W is at most $2 + 3\log_2(W/w(\nu))$.

It can be shown that this result is asymptotically best possible. In other words, the amortized search time in a splay tree is at most a constant times the optimum, which is what we achieve with an optimum weighted binary search tree. In contrast to splay trees, optimum trees are expensive to construct and they require explicit knowledge of the weights.

Second Homework Assignment

Write the solution to each problem on a single page. The deadline for handing in solutions is October 02.

Problem 1. ($20 = 12 + 8$ points). Consider an array $A[1..n]$ for which we know that $A[1] \geq A[2]$ and $A[n-1] \leq A[n]$. We say that i is a *local minimum* if $A[i-1] \geq A[i] \leq A[i+1]$. Note that A has at least one local minimum.

(a) We can obviously find a local minimum in time O(n). Describe a more efficient algorithm that does the same.

(b) Analyze your algorithm.

Problem 2. (20 points). A *vertex cover* for a tree is a subset V of its vertices such that each edge has at least one endpoint in V. It is *minimum* if there is no other vertex cover with a smaller number of vertices. Given a tree with n vertices, describe an O(n)-time algorithm for finding a minimum vertex cover. (Hint: use dynamic programming or the greedy method.)

Problem 3. (20 points). Consider a red-black tree formed by the sequential insertion of $n > 1$ items. Argue that the resulting tree has at least one red edge.

[Notice that we are talking about a red-black tree formed by insertions. Without this assumption, the tree could of course consist of black edges only.]

Problem 4. (20 points). Prove that $2n$ rotations suffice to transform any binary search tree into any other binary search tree storing the same n items.

Problem 5. ($20 = 5 + 5 + 5 + 5$ points). Consider a collection of items, each consisting of a key and a cost. The keys come from a totally ordered universe and the costs are real numbers. Show how to maintain a collection of items under the following operations:

(a) ADD(k, c): assuming no item in the collection has key k yet, add an item with key k and cost c to the collection;

(b) REMOVE(k): remove the item with key k from the collection;

(c) MAX(k_1, k_2): assuming $k_1 \leq k_2$, report the maximum cost among all items with keys $k \in [k_1, k_2]$.

(d) COUNT(c_1, c_2): assuming $c_1 \leq c_2$, report the number of items with cost $c \in [c_1, c_2]$;

Each operation should take at most O($\log n$) time in the worst case, where n is the number of items in the collection when the operation is performed.

III PRIORITIZING

10 Heaps and Heapsort

A heap is a data structure that stores a set and allows fast access to the item with highest priority. It is the basis of a fast implementation of selection sort. On the average, this algorithm is a little slower than quicksort but it is not sensitive to the input ordering or to random bits and runs about as fast in the worst case as on the average.

Priority queues. A data structure implements the *priority queue* abstract data type if it supports at least the following operations:

`void` INSERT (`item`),
`item` FINDMIN (`void`),
`void` DELETEMIN (`void`).

The operations are applied to a set of items with priorities. The priorities are totally ordered so any two can be compared. To avoid any confusion, we will usually refer to the priorities as ranks. We will always use integers as priorities and follow the convention that smaller ranks represent higher priorities. In many applications, FINDMIN and DELETEMIN are combined:

```
void EXTRACTMIN(void)
  r = FINDMIN; DELETEMIN; return r.
```

Function EXTRACTMIN removes and returns the item with smallest rank.

Heap. A heap is a particularly compact priority queue. We can think of it as a binary tree with items stored in the internal nodes, as in Figure 39. Each level is full, except

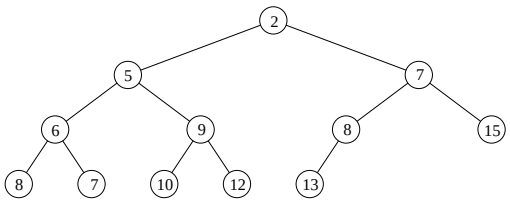

Figure 39: Ranks increase or, more precisely, do not decrease from top to bottom.

possibly the last, which is filled from left to right until we run out of items. The items are stored in *heap-order*: every node μ has a rank larger than or equal to the rank of its parent. Symmetrically, μ has a rank less than or equal to the ranks of both its children. As a consequence, the root contains the item with smallest rank.

We store the nodes of the tree in a linear array, level by level from top to bottom and each level from left to right, as shown in Figure 40. The embedding saves ex-

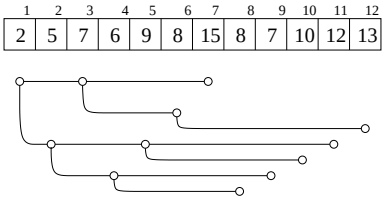

Figure 40: The binary tree is layed out in a linear array. The root is placed in $A[1]$, its children follow in $A[2]$ and $A[3]$, etc.

plicit pointers otherwise needed to establish parent-child relations. Specifically, we can find the children and parent of a node by index computation: the left child of $A[i]$ is $A[2i]$, the right child is $A[2i+1]$, and the parent is $A[\lfloor i/2 \rfloor]$. The item with minimum rank is stored in the first element:

```
item FINDMIN(int n)
  assert n ≥ 1; return A[1].
```

Since the index along a path at least doubles each step, paths can have length at most $\log_2 n$.

Deleting the minimum. We first study the problem of repairing the heap-order if it is violated at the root, as shown in Figure 41. Let n be the length of the array. We

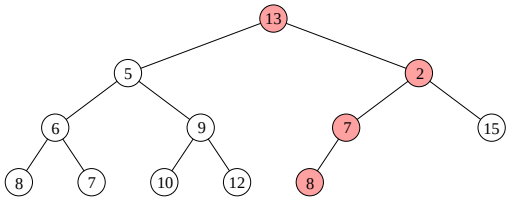

Figure 41: The root is exchanged with the smaller of its two children. The operation is repeated along a single path until the heap-order is repaired.

repair the heap-order by a sequence of swaps along a single path. Each swap is between an item and the smaller of its children:

```
void SIFT-DN(int i, n)
  if 2i ≤ n then
    k = arg min{A[2i], A[2i+1]}
    if A[k] < A[i] then SWAP(i, k);
                        SIFT-DN(k, n)
    endif
  endif.
```

Here we assume that $A[n+1]$ is defined and larger than $A[n]$. Since a path has at most $\log_2 n$ edges, the time to repair the heap-order takes time at most $O(\log n)$. To delete the minimum we overwrite the root with the last element, shorten the heap, and repair the heap-order:

```
void DELETEMIN(int *n)
  A[1] = A[*n]; *n--; SIFT-DN(1, *n).
```

Instead of the variable that stores n, we pass a pointer to that variable, $*n$, in order to use it as input and output parameter.

Inserting. Consider repairing the heap-order if it is violated at the last position of the heap. In this case, the item moves up the heap until it reaches a position where its rank is at least as large as that of its parent.

```
void SIFT-UP(int i)
  if i ≥ 2 then k = ⌊i/2⌋;
    if A[i] < A[k] then SWAP(i, k);
                        SIFT-UP(k)
    endif
  endif.
```

An item is added by first expanding the heap by one element, placing the new item in the position that just opened up, and repairing the heap-order.

```
void INSERT(int *n, item x)
  *n++; A[*n] = x; SIFT-UP(*n).
```

A heap supports FINDMIN in constant time and INSERT and DELETEMIN in time $O(\log n)$ each.

Sorting. Priority queues can be used for sorting. The first step throws all items into the priority queue, and the second step takes them out in order. Assuming the items are already stored in the array, the first step can be done by repeated heap repair:

```
for i = 1 to n do SIFT-UP(i) endfor.
```

In the worst case, the i-th item moves up all the way to the root. The number of exchanges is therefore at most $\sum_{i=1}^{n} \log_2 i \le n \log_2 n$. The upper bound is asymptotically tight because half the terms in the sum are at least $\log_2 \frac{n}{2} = \log_2 n - 1$. It is also possible to construct the initial heap in time $O(n)$ by building it from bottom to top. We modify the first step accordingly, and we implement the second step to rearrange the items in sorted order:

```
void HEAPSORT(int n)
  for i = n downto 1 do SIFT-DN(i, n) endfor;
  for i = n downto 1 do
    SWAP(i, 1); SIFT-DN(1, i − 1)
  endfor.
```

At each step of the first `for`-loop, we consider the subtree with root $A[i]$. At this moment, the items in the left and right subtrees rooted at $A[2i]$ and $A[2i+1]$ are already heaps. We can therefore use one call to function SIFT-DN to make the subtree with root $A[i]$ a heap. We will prove shortly that this bottom-up construction of the heap takes time only $O(n)$. Figure 42 shows the array after each iteration of the second `for`-loop. Note how the heap gets smaller by one element each step. A sin-

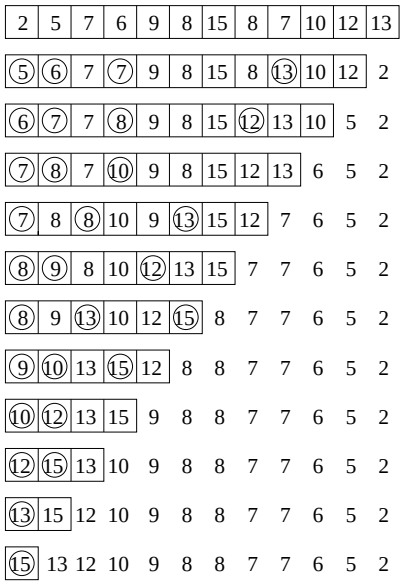

Figure 42: Each step moves the last heap element to the root and thus shrinks the heap. The circles mark the items involved in the sift-down operation.

gle sift-down operation takes time $O(\log n)$, and in total HEAPSORT takes time $O(n \log n)$. In addition to the input array, HEAPSORT uses a constant number of variables

and memory for the recursion stack used by SIFT-DN. We can save the memory for the stack by writing function SIFT-DN as an iteration. The sort can be changed to non-decreasing order by reversing the order of items in the heap.

Analysis of heap construction. We return to proving that the bottom-up approach to constructing a heap takes only $O(n)$ time. Assuming the worst case, in which every node sifts down all the way to the last level, we draw the swaps as edges in a tree; see Figure 43. To avoid

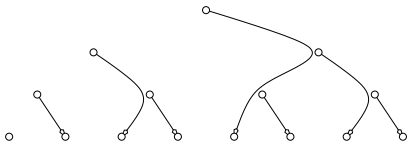

Figure 43: Each node generates a path that shares no edges with the paths of the other nodes.

drawing any edge twice, we always first swap to the right and then continue swapping to the left until we arrive at the last level. This introduces only a small inaccuracy in our estimate. The paths cover each edge once, except for the edges on the leftmost path, which are not covered at all. The number of edges in the tree is $n - 1$, which implies that the total number of swaps is less than n. Equivalently, the amortized number of swaps per item is less than 1. There is a striking difference in time-complexity to sorting, which takes an amortized number of about $\log_2 n$ comparisons per item. The difference between 1 and $\log_2 n$ may be interpreted as a measure of how far from sorted a heap-ordered array still is.

11 Fibonacci Heaps

The Fibonacci heap is a data structure implementing the priority queue abstract data type, just like the ordinary heap but more complicated and asymptotically faster for some operations. We first introduce binomial trees, which are special heap-ordered trees, and then explain Fibonacci heaps as collections of heap-ordered trees.

Binomial trees. The *binomial tree* of height h is a tree obtained from two binomial trees of height $h-1$, by linking the root of one to the other. The binomial tree of height 0 consists of a single node. Binomial trees of heights up to 4 are shown in Figure 44. Each step in the construction increases the height by one, increases the *degree* (the number of children) of the root by one, and doubles the size of the tree. It follows that a binomial tree of height h has root degree h and size 2^h. The root has the largest degree of any node in the binomial tree, which implies that every node in a binomial tree with n nodes has degree at most $\log_2 n$.

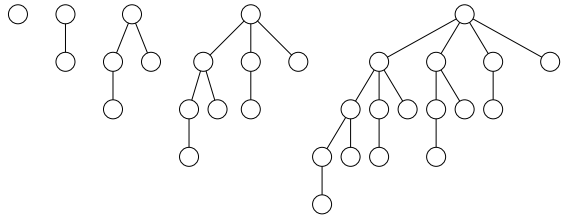

Figure 44: Binomial trees of heights 0, 1, 2, 3, 4. Each tree is obtained by linking two copies of the previous tree.

To store any set of items with priorities, we use a small collection of binomial trees. For an integer n, let n_i be the i-th bit in the binary notation, so we can write $n = \sum_{i\geq 0} n_i 2^i$. To store n items, we use a binomial tree of size 2^i for each $n_i = 1$. The total number of binomial trees is thus the number of 1's in the binary notation of n, which is at most $\log_2(n+1)$. The collection is referred to as a *binomial heap*. The items in each binomial tree are stored in heap-order. There is no specific relationship between the items stored in different binomial trees. The item with minimum key is thus stored in one of the logarithmically many roots, but it is not prescribed ahead of time in which one. An example is shown in Figure 45 where $11_{10} = 1011_2$ items are stored in three binomial trees with sizes 8, 2, and 1. In order to add a new item to the set, we create a new binomial tree of size 1 and we successively link binomial trees as dictated by the rules of adding 1 to the binary notation of n. In the example, we get $1011_2 + 1_2 = 1100_2$. The new collection thus consists of two binomial trees with sizes 8 and 4. The size 8 tree is the old one, and the size 4 tree is obtained by first linking the two size 1 trees and then linking the resulting size 2 tree to the old size 2 tree. All this is illustrated in Figure 45.

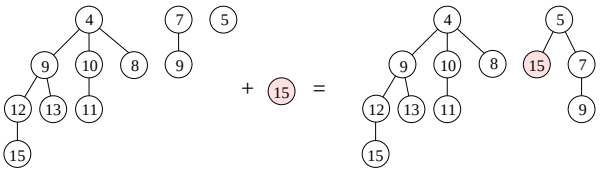

Figure 45: Adding the shaded node to a binomial heap consisting of three binomial trees.

Fibonacci heaps. A *Fibonacci heap* is a collection of heap-ordered trees. Ideally, we would like it to be a collection of binomial trees, but we need more flexibility. It will be important to understand how exactly the nodes of a Fibonacci heap are connected by pointers. Siblings are organized in doubly-linked cyclic lists, and each node has a pointer to its parent and a pointer to one of its children, as shown in Figure 46. Besides the pointers, each node stores a key, its degree, and a bit that can be used to mark or unmark the node. The roots of the heap-ordered trees are doubly-linked in a cycle, and there is an explicit pointer to the root that stores the item with the minimum key. Figure 47 illustrates a few basic operations we perform on a Fibonacci heap. Given two heap-ordered trees, we *link* them by making the root with the bigger key the child of the other root. To *unlink* a heap-ordered tree or subtree, we remove its root from the doubly-linked cycle. Finally, to *merge* two cycles, we cut both open and connect them at

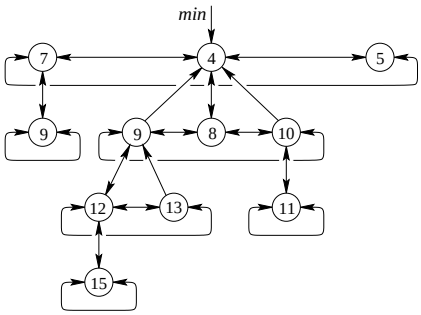

Figure 46: The Fibonacci heap representation of the first collection of heap-ordered trees in Figure 45.

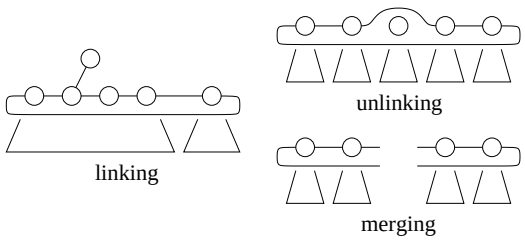

Figure 47: Cartoons for linking two trees, unlinking a tree, and merging two cycles.

their ends. Any one of these three operations takes only constant time.

Potential function. A Fibonacci heap supports a variety of operations, including the standard ones for priority queues. We use a potential function to analyze their amortized cost applied to an initially empty Fibonacci heap. Letting r_i be the number of roots in the root cycle and m_i the number of marked nodes, the *potential* after the i-th operation is $\Phi_i = r_i + 2m_i$. When we deal with a collection of Fibonacci heaps, we define its potential as the sum of individual potentials. The initial Fibonacci heap is empty, so $\Phi_0 = 0$. As usual, we let c_i be the actual cost and $a_i = c_i + \Phi_i - \Phi_{i-1}$ the amortized cost of the i-th operation. Since $\Phi_0 = 0$ and $\Phi_i \geq 0$ for all i, the actual cost is less than the amortized cost:

$$\sum_{i=1}^{n} c_i \;\leq\; \sum_{i=1}^{n} a_i \;=\; r_n + 2m_n + \sum_{i=1}^{n} c_i.$$

For some of the operations, it is fairly easy to compute the amortized cost. We get the **minimum** by returning the key in the marked root. This operation does not change the potential and its amortized and actual cost is $a_i = c_i = 1$. We **meld** two Fibonacci heaps, H_1 and H_2, by first merging the two root circles and second adjusting the pointer to the minimum key. We have

$$\begin{aligned} r_i(H) &= r_{i-1}(H_1) + r_{i-1}(H_2), \\ m_i(H) &= m_{i-1}(H_1) + m_{i-1}(H_2), \end{aligned}$$

which implies that there is no change in potential. The amortized and actual cost is therefore $a_i = c_i = 1$. We **insert** a key into a Fibonacci heap by first creating a new Fibonacci heap that stores only the new key and second melding the two heaps. We have one more node in the root cycle so the change in potential is $\Phi_i - \Phi_{i-1} = 1$. The amortized cost is therefore $a_i = c_i + 1 = 2$.

Deletemin. Next we consider the somewhat more involved operation of deleting the minimum key, which is done in four steps:

`Step 1.` Remove the node with minimum key from the root cycle.

`Step 2.` Merge the root cycle with the cycle of children of the removed node.

`Step 3.` As long as there are two roots with the same degree link them.

`Step 4.` Recompute the pointer to the minimum key.

For Step 3, we use a pointer array R. Initially, $R[i] =$ NULL for each i. For each root ϱ in the root cycle, we execute the following iteration.

$i = \varrho \rightarrow degree$;
`while` $R[i] \neq$ NULL `do`
  $\varrho' = R[i]$; $R[i] =$ NULL; $\varrho =$ LINK(ϱ, ϱ'); i`++`
`endwhile`;
$R[i] = \varrho$.

To analyze the amortized cost for deleting the minimum, let $D(n)$ be the maximum possible degree of any node in a Fibonacci heap of n nodes. The number of linking operations in Step 3 is the number of roots we start with, which is less than $r_{i-1} + D(n)$, minus the number of roots we end up with, which is r_i. After Step 3, all roots have different degrees, which implies $r_i \leq D(n)+1$. It follows that the actual cost for the four steps is

$$\begin{aligned} c_i &\leq 1 + 1 + (r_{i-1} + D(n) - r_i) + (D(n) + 1) \\ &= 3 + 2D(n) + r_{i-1} - r_i. \end{aligned}$$

The potential change is $\Phi_i - \Phi_{i-1} = r_i - r_{i-1}$. The amortized cost is therefore $a_i = c_i + \Phi_i - \Phi_{i-1} \leq 2D(n) + 3$. We will prove next time that the maximum possible degree is at most logarithmic in the size of the Fibonacci heap, $D(n) < 2\log_2(n+1)$. This implies that deleting the minimum has logarithmic amortized cost.

Decreasekey and delete. Besides deletemin, we also have operations that delete an arbitrary item and that decrease the key of an item. Both change the structure of the heap-ordered trees and are the reason why a Fibonacci heap is not a collection of binomial trees but of more general heap-ordered trees. The **decreasekey** operation replaces the item with key x stored in the node ν by $x - \Delta$, where $\Delta \geq 0$. We will see that this can be done more efficiently than to delete x and to insert $x - \Delta$. We decrease the key in four steps.

`Step 1.` Unlink the tree rooted at ν.

`Step 2.` Decrease the key in ν by Δ.

`Step 3.` Add ν to the root cycle and possibly update the pointer to the minimum key.

`Step 4.` Do cascading cuts.

We will explain cascading cuts shortly, after explaining the four steps we take to delete a node ν. Before we delete a node ν, we check whether $\nu = min$, and if it is then we delete the minimum as explained above. Assume therefore that $\nu \neq min$.

`Step 1.` Unlink the tree rooted at ν.

`Step 2.` Merge the root-cycle with the cycle of ν's children.

`Step 3.` Dispose of ν.

`Step 4.` Do cascading cuts.

Figure 48 illustrates the effect of decreasing a key and of deleting a node. Both operations create trees that are not

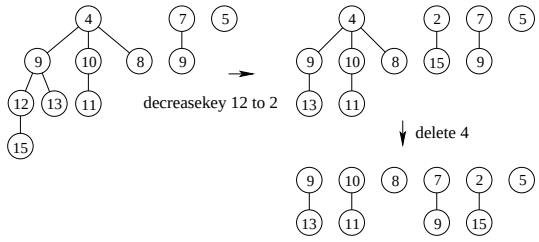

Figure 48: A Fibonacci heap initially consisting of three binomial trees modified by a decreasekey and a delete operation.

binomial, and we use cascading cuts to make sure that the shapes of these trees are not very different from the shapes of binomial trees.

Cascading cuts. Let ν be a node that becomes the child of another node at time t. We mark ν when it loses its first child after time t. Then we unmark ν, unlink it, and add it to the root-cycle when it loses its second child thereafter. We call this operation a *cut*, and it may cascade because one cut can cause another, and so on. Figure 49 illustrates the effect of cascading in a heap-ordered tree with two marked nodes. The first step decreases key 10 to 7, and the second step cuts first node 5 and then node 4.

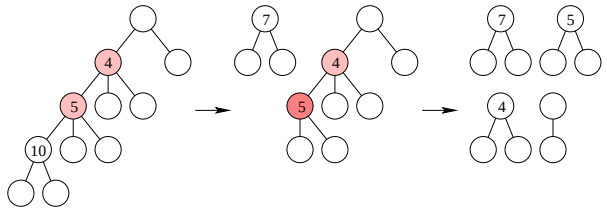

Figure 49: The effect of cascading after decreasing 10 to 7. Marked nodes are shaded.

Summary analysis. As mentioned earlier, we will prove $D(n) < 2\log_2(n+1)$ next time. Assuming this bound, we are able to compute the amortized cost of all operations. The actual cost of Step 4 in decreasekey or in delete is the number of cuts, c_i. The potential changes because there are c_i new roots and c_i fewer marked nodes. Also, the last cut may introduce a new mark. Thus

$$\begin{aligned} \Phi_i - \Phi_{i-1} &= r_i - r_{i-1} + 2m_i - 2m_{i-1} \\ &\leq c_i - 2c_i + 2 \\ &= -c_i + 2. \end{aligned}$$

The amortized cost is therefore $a_i = c_i + \Phi_i - \Phi_{i-1} \leq c_i - (2 - c_i) = 2$. The first three steps of a decreasekey operation take only a constant amount of actual time and increase the potential by at most a constant amount. It follows that the amortized cost of decreasekey, including the cascading cuts in Step 4, is only a constant. Similarly, the actual cost of a delete operation is at most a constant, but Step 2 may increase the potential of the Fibonacci heap by as much as $D(n)$. The rest is bounded from above by a constant, which implies that the amortized cost of the delete operation is O($\log n$). We summarize the amortized cost of the various operations supported by the Fibonacci heap:

find the minimum	O(1)
meld two heaps	O(1)
insert a new item	O(1)
delete the minimum	O($\log n$)
decrease the key of a node	O(1)
delete a node	O($\log n$)

We will later see graph problems for which the difference in the amortized cost of the decreasekey and delete operations implies a significant improvement in the running time.

12 Solving Recurrence Relations

Recurrence relations are perhaps the most important tool in the analysis of algorithms. We have encountered several methods that can sometimes be used to solve such relations, such as guessing the solution and proving it by induction, or developing the relation into a sum for which we find a closed form expression. We now describe a new method to solve recurrence relations and use it to settle the remaining open question in the analysis of Fibonacci heaps.

Annihilation of sequences. Suppose we are given an infinite sequence of numbers, $A = \langle a_0, a_1, a_2, \ldots \rangle$. We can multiply with a constant, shift to the left and add another sequence:

$$\begin{aligned} kA &= \langle ka_0, ka_1, ka_2, \ldots \rangle, \\ \mathrm{L}A &= \langle a_1, a_2, a_3, \ldots \rangle, \\ A + B &= \langle a_0 + b_0, a_1 + b_1, a_2 + b_2, \ldots \rangle. \end{aligned}$$

As an example, consider the sequence of powers of two, $a_i = 2^i$. Multiplying with 2 and shifting to the left give the same result. Therefore,

$$\mathrm{L}A - 2A = \langle 0, 0, 0, \ldots \rangle.$$

We write $\mathrm{L}A - 2A = (\mathrm{L} - 2)A$ and think of $\mathrm{L} - 2$ as an operator that *annihilates* the sequence of powers of 2. In general, $\mathrm{L} - k$ annihilates any sequence of the form $\langle ck^i \rangle$. What does $\mathrm{L} - k$ do to other sequences $A = \langle c\ell^i \rangle$, when $\ell \neq k$?

$$\begin{aligned} (\mathrm{L} - k)A &= \langle c\ell, c\ell^2, c\ell^3, \ldots \rangle - \langle ck, ck\ell, ck\ell^2, \ldots \rangle \\ &= (\ell - k)\langle c, c\ell, c\ell^2, \ldots \rangle \\ &= (\ell - k)A. \end{aligned}$$

We see that the operator $\mathrm{L} - k$ annihilates only one type of sequence and multiplies other similar sequences by a constant.

Multiple operators. Instead of just one, we can apply several operators to a sequence. We may multiply with two constants, $k(\ell A) = (k\ell)A$, multiply and shift, $\mathrm{L}(kA) = k(\mathrm{L}A)$, and shift twice, $\mathrm{L}(\mathrm{L}A) = \mathrm{L}^2 A$. For example, $(\mathrm{L} - k)(\mathrm{L} - \ell)$ annihilates all sequences of the form $\langle ck^i + d\ell^i \rangle$, where we assume $k \neq \ell$. Indeed, $\mathrm{L} - k$ annihilates $\langle ck^i \rangle$ and leaves behind $\langle (\ell - k)d\ell^i \rangle$, which is annihilated by $\mathrm{L} - \ell$. Furthermore, $(\mathrm{L} - k)(\mathrm{L} - \ell)$ annihilates no other sequences. More generally, we have

FACT. $(\mathrm{L} - k_1)(\mathrm{L} - k_2) \ldots (\mathrm{L} - k_n)$ annihilates all sequences of the form $\langle c_1 k_1^i + c_2 k_2^i + \ldots + c_n k_n^i \rangle$.

What if $k = \ell$? To answer this question, we consider

$$\begin{aligned} (\mathrm{L} - k)^2 \langle ik^i \rangle &= (\mathrm{L} - k)\langle (i+1)k^{i+1} - ik^{i+1} \rangle \\ &= (\mathrm{L} - k)\langle k^{i+1} \rangle \\ &= \langle 0 \rangle. \end{aligned}$$

More generally, we have

FACT. $(\mathrm{L} - k)^n$ annihilates all sequences of the form $\langle p(i)k^i \rangle$, with $p(i)$ a polynomial of degree $n - 1$.

Since operators annihilate only certain types of sequences, we can determine the sequence if we know the annihilating operator. The general method works in five steps:

1. Write down the annihilator for the recurrence.
2. Factor the annihilator.
3. Determine what sequence each factor annihilates.
4. Put the sequences together.
5. Solve for the constants of the solution by using initial conditions.

Fibonacci numbers. We put the method to a test by considering the Fibonacci numbers defined recursively as follows:

$$\begin{aligned} F_0 &= 0, \\ F_1 &= 1, \\ F_j &= F_{j-1} + F_{j-2}, \quad \text{for } j \geq 2. \end{aligned}$$

Writing a few of the initial numbers, we get the sequence $\langle 0, 1, 1, 2, 3, 5, 8, \ldots \rangle$. We notice that $\mathrm{L}^2 - \mathrm{L} - 1$ annihilates the sequence because

$$\begin{aligned} (\mathrm{L}^2 - \mathrm{L} - 1)\langle F_j \rangle &= \mathrm{L}^2\langle F_j \rangle - \mathrm{L}\langle F_j \rangle - \langle F_j \rangle \\ &= \langle F_{j+2} \rangle - \langle F_{j+1} \rangle - \langle F_j \rangle \\ &= \langle 0 \rangle. \end{aligned}$$

If we factor the operator into its roots, we get

$$\mathrm{L}^2 - \mathrm{L} - 1 = (\mathrm{L} - \varphi)(\mathrm{L} - \overline{\varphi}),$$

where

$$\varphi = \frac{1 + \sqrt{5}}{2} = 1.618\ldots,$$

$$\overline{\varphi} = 1 - \varphi = \frac{1 - \sqrt{5}}{2} = -0.618\ldots.$$

The first root is known as the *golden ratio* because it represents the aspect ratio of a rectangular piece of paper from which we may remove a square to leave a smaller rectangular piece of the same ratio: $\varphi : 1 = 1 : \varphi - 1$. Thus we know that $(\mathrm{L} - \varphi)(\mathrm{L} - \overline{\varphi})$ annihilates $\langle F_j \rangle$ and this means that the j-th Fibonacci number is of the form $F_j = c\varphi^j + \overline{c}\,\overline{\varphi}^j$. We get the constant factors from the initial conditions:

$$\begin{aligned} F_0 &= 0 = c + \overline{c}, \\ F_1 &= 1 = c\varphi + \overline{c}\,\overline{\varphi}. \end{aligned}$$

Solving the two linear equations in two unknowns, we get $c = 1/\sqrt{5}$ and $\overline{c} = -1/\sqrt{5}$. This implies that

$$F_j = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^j - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^j .$$

From this viewpoint, it seems surprising that F_j turns out to be an integer for all j. Note that $|\varphi| > 1$ and $|\overline{\varphi}| < 1$. It follows that for growing exponent j, φ^j goes to infinity and $\overline{\varphi}^j$ goes to zero. This implies that F_j is approximately $\varphi^j/\sqrt{5}$, and that this approximation becomes more and more accurate as j grows.

Maximum degree. Recall that $D(n)$ is the maximum possible degree of any one node in a Fibonacci heap of size n. We need two easy facts about the kind of trees that arise in Fibonacci heaps in order to show that $D(n)$ is at most logarithmic in n. Let ν be a node of degree j, and let $\mu_1, \mu_2, \ldots, \mu_j$ be its children ordered by the time they were linked to ν.

DEGREE LEMMA. The degree of μ_i is at least $i - 2$.

PROOF. Recall that nodes are linked only during the deletemin operation. Right before the linking happens, the two nodes are roots and have the same degree. It follows that the degree of μ_i was at least $i - 1$ at the time it was linked to ν. The degree of μ_i might have been even higher because it is possible that ν lost some of the older children after μ_i had been linked. After being linked, μ_i may have lost at most one of its children, for else it would have been cut. Its degree is therefore at least $i - 2$, as claimed. □

SIZE LEMMA. The number of descendents of ν (including ν) is at least F_{j+2}.

PROOF. Let s_j be the minimum number of descendents a node of degree j can have. We have $s_0 = 1$ and $s_1 = 2$. For larger j, we get s_j from s_{j-1} by adding the size of a minimum tree with root degree $j-2$, which is s_{j-2}. Hence $s_j = s_{j-1} + s_{j-2}$, which is the same recurrence relation that defines the Fibonacci numbers. The initial values are shifted two positions so we get $s_j = F_{j+2}$, as claimed. □

Consider a Fibonacci heap with n nodes and let ν be a node with maximum degree $D = D(n)$. The Size Lemma implies $n \geq F_{D+2}$. The Fibonacci number with index $D + 2$ is roughly $\varphi^{D+2}/\sqrt{5}$. Because $\overline{\varphi}^{D+2} < \sqrt{5}$, we have

$$n \geq \frac{1}{\sqrt{5}} \varphi^{D+2} - 1.$$

After rearranging the terms and taking the logarithm to the base φ, we get

$$D \leq \log_\varphi \sqrt{5}(n+1) - 2.$$

Recall that $\log_\varphi x = \log_2 x / \log_2 \varphi$ and use the calculator to verify that $\log_2 \varphi = 0.694\ldots > 0.5$ and $\log_\varphi \sqrt{5} = 1.672\ldots < 2$. Hence

$$\begin{aligned} D &\leq \frac{\log_2(n+1)}{\log_2 \varphi} + \log_\varphi \sqrt{5} - 2 \\ &< 2\log_2(n+1). \end{aligned}$$

Non-homogeneous terms. We now return to the annihilation method for solving recurrence relations and consider

$$a_j = a_{j-1} + a_{j-2} + 1.$$

This is similar to the recurrence that defines Fibonacci numbers and describes the minimum number of nodes in an *AVL tree*, also known as *height-balanced tree*. It is defined by the requirement that the height of the two subtrees of a node differ by at most 1. The smallest tree of height j thus consists of the root, a subtree of height $j - 1$ and another subtree of height $j - 2$. We refer to the terms involving a_i as the *homogeneous* terms of the relation and the others as the *non-homogeneous* terms. We know that $\mathrm{L}^2 - \mathrm{L} - 1$ annihilates the homogeneous part, $a_j = a_{j-1} + a_{j-2}$. If we apply it to the entire relation we get

$$\begin{aligned} (\mathrm{L}^2 - \mathrm{L} - 1)\langle a_j \rangle &= \langle a_{j+2} \rangle - \langle a_{j+1} \rangle - \langle a_j \rangle \\ &= \langle 1, 1, \ldots \rangle. \end{aligned}$$

The remaining sequence of 1s is annihilated by $\mathrm{L} - 1$. In other words, $(\mathrm{L} - \varphi)(\mathrm{L} - \overline{\varphi})(\mathrm{L} - 1)$ annihilates $\langle a_j \rangle$ implying that $a_j = c\varphi^j + \overline{c}\,\overline{\varphi}^j + c'1^j$. It remains to find

the constants, which we get from the boundary conditions $a_0 = 1$, $a_1 = 2$ and $a_2 = 4$:

$$\begin{array}{rcrcrcl} c & + & \overline{c} & + & c' & = & 1, \\ \varphi c & + & \overline{\varphi}\,\overline{c} & + & c' & = & 2, \\ \varphi^2 c & + & \overline{\varphi}^2\overline{c} & + & c' & = & 4. \end{array}$$

Noting that $\varphi^2 = \varphi + 1$, $\overline{\varphi}^2 = \overline{\varphi} + 1$, and $\varphi - \overline{\varphi} = \sqrt{5}$ we get $c = (5 + 2\sqrt{5})/5$, $\overline{c} = (5 - 2\sqrt{5})/5$, and $c' = -1$. The minimum number of nodes of a height-j AVL tree is therefore roughly the constant c times φ^j. Conversely, the maximum height of an AVL tree with $n = c\varphi^j$ nodes is roughly $j = \log_\varphi(n/c) = 1.440\ldots \cdot \log_2 n + \mathrm{O}(1)$. In words, the height-balancing condition implies logarithmic height.

Transformations. We extend the set of recurrences we can solve by employing transformations that produce relations amenable to the annihilation method. We demonstrate this by considering mergesort, which is another divide-and-conquer algorithm that can be used to sort a list of n items:

`Step 1.` Recursively sort the left half of the list.

`Step 2.` Recursively sort the right half of the list.

`Step 3.` Merge the two sorted lists by simultaneously scanning both from beginning to end.

The running time is described by the solution to the recurrence

$$\begin{array}{rcl} T(1) & = & 1, \\ T(n) & = & 2T(n/2) + n. \end{array}$$

We have no way to work with terms like $T(n/2)$ yet. However, we can transform the recurrence into a more manageable form. Defining $n = 2^i$ and $t_i = T(2^i)$ we get

$$\begin{array}{rcl} t_0 & = & 1, \\ t_i & = & 2t_{i-1} + 2^i. \end{array}$$

The homogeneous part is annihilated by $\mathrm{L} - 2$. Similarly, non-homogeneous part is annihilated by $\mathrm{L} - 2$. Hence, $(\mathrm{L} - 2)^2$ annihilates the entire relation and we get $t_i = (ci + \overline{c})2^i$. Expressed in the original notation we thus have $T(n) = (c \log_2 n + \overline{c})n = \mathrm{O}(n \log n)$. This result is of course no surprise and reconfirms what we learned earlier about sorting.

The Master Theorem. It is sometimes more convenient to look up the solution to a recurrence relation than playing with different techniques to see whether any one can make it to yield. Such a cookbook method for recurrence relations of the form

$$T(n) \quad = \quad aT(n/b) + f(n)$$

is provided by the following theorem. Here we assume that $a \geq 1$ and $b > 1$ are constants and that f is a well-behaved positive function.

MASTER THEOREM. Define $c = \log_b a$ and let ε be an arbitrarily small positive constant. Then

$$T(n) \quad = \quad \begin{cases} \mathrm{O}(n^c) & \text{if } f(n) = \mathrm{O}(n^{c-\varepsilon}), \\ \mathrm{O}(n^c \log n) & \text{if } f(n) = \mathrm{O}(n^c), \\ \mathrm{O}(f(n)) & \text{if } f(n) = \Omega(n^{c+\varepsilon}). \end{cases}$$

The last of the three cases also requires a usually satisfied technical condition, namely that $af(n/b) < \delta f(n)$ for some constant δ strictly less than 1. For example, this condition is satisfied in $T(n) = 2T(n/2) + n^2$ which implies $T(n) = \mathrm{O}(n^2)$.

As another example consider the relation $T(n) = 2T(n/2) + n$ that describes the running time of mergesort. We have $c = \log_2 2 = 1$ and $f(n) = n = \mathrm{O}(n^c)$. The middle case of the Master Theorem applies and we get $T(n) = \mathrm{O}(n \log n)$, as before.

Third Homework Assignment

Write the solution to each problem on a single page. The deadline for handing in solutions is October 14.

Problem 1. ($20 = 10 + 10$ points). Consider a lazy version of heapsort in which each item in the heap is either smaller than or equal to every other item in its subtree, or the item is identified as *uncertified*. To *certify* an item, we certify its children and then exchange it with the smaller child provided it is smaller than the item itself. Suppose $A[1..n]$ is a lazy heap with all items uncertified.

(a) How much time does it take to certify $A[1]$?

(b) Does certifying $A[1]$ turn A into a proper heap in which every item satisfies the heap property? (Justify your answer.)

Problem 2. (20 points). Recall that Fibonacci numbers are defined recursively as $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$. Prove the square of the n-th Fibonacci number differs from the product of the two adjacent numbers by one: $F_n^2 = F_{n-1} \cdot F_{n+1} + (-1)^{n+1}$.

Problem 3. (20 points). Professor Pinocchio claims that the height of an n-node Fibonacci heap is at most some constant times $\log_2 n$. Show that the Professor is mistaken by exhibiting, for any integer n, a sequence of operations that create a Fibonacci heap consisting of just one tree that is a linear chain of n nodes.

Problem 4. ($20 = 10 + 10$ points). To search in a sorted array takes time logarithmic in the size of the array, but to insert a new items takes linear time. We can improve the running time for insertions by storing the items in several instead of just one sorted arrays. Let n be the number of items, let $k = \lceil \log_2(n+1) \rceil$, and write $n = n_{k-1}n_{k-2}\ldots n_0$ in binary notation. We use k sorted arrays A_i (some possibly empty), where A_i stores $n_i 2^i$ items. Each item is stored exactly once, and the total size of the arrays is indeed $\sum_{i=0}^{k} n_i 2^i = n$. Although each individual array is sorted, there is no particular relationship between the items in different arrays.

(a) Explain how to search in this data structure and analyze your algorithm.

(b) Explain how to insert a new item into the data structure and analyze your algorithm, both in worst-case and in amortized time.

Problem 5. ($20 = 10 + 10$ points). Consider a full binary tree with n leaves. The *size* of a node, $s(\nu)$, is the number of leaves in its subtree and the *rank* is the floor of the binary logarithm of the size, $r(\nu) = \lfloor \log_2 s(\nu) \rfloor$.

(a) Is it true that every internal node ν has a child whose rank is strictly less than the rank of ν?

(b) Prove that there exists a leaf whose depth (length of path to the root) is at most $\log_2 n$.

IV GRAPH ALGORITHMS

13 Graph Search

We can think of graphs as generalizations of trees: they consist of nodes and edges connecting nodes. The main difference is that graphs do not in general represent hierarchical organizations.

Types of graphs. Different applications require different types of graphs. The most basic type is the *simple undirected graph* that consists of a set V of *vertices* and a set E of *edges*. Each edge is an unordered pair (a set) of two vertices. We always assume V is finite, and we write

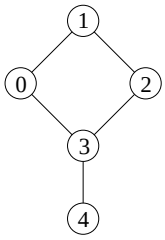

Figure 50: A simple undirected graph with vertices $0, 1, 2, 3, 4$ and edges $\{0,1\}, \{1,2\}, \{2,3\}, \{3,0\}, \{3,4\}$.

$\binom{V}{2}$ for the collection of all unordered pairs. Hence E is a subset of $\binom{V}{2}$. Note that because E is a set, each edge can occur only once. Similarly, because each edge is a set (of two vertices), it cannot connect to the same vertex twice. Vertices u and v are *adjacent* if $\{u, v\} \in E$. In this case u and v are called *neighbors*. Other types of graphs are

directed:	$E \subseteq V \times V$.
weighted:	has a weighting function $w : E \to \mathbb{R}$.
labeled:	has a labeling function $\ell : V \to \mathbb{Z}$.
non-simple:	there are loops and multi-edges.

A *loop* is like an edge, except that it connects to the same vertex twice. A *multi-edge* consists of two or more edges connecting the same two vertices.

Representation. The two most popular data structures for graphs are direct representations of adjacency. Let $V = \{0, 1, \ldots, n-1\}$ be the set of vertices. The *adjacency matrix* is the n-by-n matrix $A = (a_{ij})$ with

$$a_{ij} = \begin{cases} 1 & \text{if } \{i,j\} \in E, \\ 0 & \text{if } \{i,j\} \notin E. \end{cases}$$

For undirected graphs, we have $a_{ij} = a_{ji}$, so A is symmetric. For weighted graphs, we encode more information than just the existence of an edge and define a_{ij} as the weight of the edge connecting i and j. The adjacency matrix of the graph in Figure 50 is

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix},$$

which is symmetric. Irrespective of the number of edges,

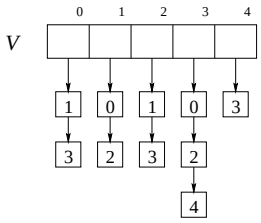

Figure 51: The adjacency list representation of the graph in Figure 50. Each edge is represented twice, once for each endpoint.

the adjacency matrix has n^2 elements and thus requires a quadratic amount of space. Often, the number of edges is quite small, maybe not much larger than the number of vertices. In these cases, the adjacency matrix wastes memory, and a better choice is a sparse matrix representation referred to as *adjacency lists*, which is illustrated in Figure 51. It consists of a linear array V for the vertices and a list of neighbors for each vertex. For most algorithms, we assume that vertices and edges are stored in structures containing a small number of fields:

```
struct Vertex {int d, f, π; Edge *adj};
struct Edge {int v; Edge *next}.
```

The d, f, π fields will be used to store auxiliary information used or created by the algorithms.

Depth-first search. Since graphs are generally not ordered, there are many sequences in which the vertices can be visited. In fact, it is not entirely straightforward to make sure that each vertex is visited once and only once. A useful method is depth-first search. It uses a global variable, *time*, which is incremented and used to leave time-stamps behind to avoid repeated visits.

```
    void VISIT(int i)
1     time++; V[i].d = time;
      forall outgoing edges ij do
2       if V[j].d = 0 then
3         V[j].π = i; VISIT(j)
        endif
      endfor;
4     time++; V[i].f = time.
```

The test in line 2 checks whether the neighbor j of i has already been visited. The assignment in line 3 records that the vertex is visited *from* vertex i. A vertex is first stamped in line 1 with the time at which it is encountered. A vertex is second stamped in line 4 with the time at which its visit has been completed. To prepare the search, we initialize the global time variable to 0, label all vertices as not yet visited, and call VISIT for all yet unvisited vertices.

```
time = 0;
forall vertices i do V[i].d = 0 endfor;
forall vertices i do
  if V[i].d = 0 then V[i].π = 0; VISIT(i) endif
endfor.
```

Let n be the number of vertices and m the number of edges in the graph. Depth-first search visits every vertex once and examines every edge twice, once for each endpoint. The running time is therefore $O(n + m)$, which is proportional to the size of the graph and therefore optimal.

DFS forest. Figure 52 illustrates depth-first search by showing the time-stamps d and f and the pointers π indicating the predecessors in the traversal. We call an edge $\{i, j\} \in E$ a *tree edge* if $i = V[j].\pi$ or $j = V[i].\pi$ and a *back edge*, otherwise. The tree edges form the *DFS forest* of the graph. The forest is a tree if the graph is connected and a collection of two or more trees if it is not connected. Figure 53 shows the DFS forest of the graph in Figure 52 which, in this case, consists of a single tree. The time-stamps d are consistent with the preorder traversal of the DFS forest. The time-stamps f are consistent with the postorder traversal. The two stamps can be used to decide, in constant time, whether two nodes in the forest live in different subtrees or one is a descendent of the other.

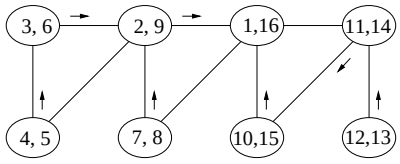

Figure 52: The traversal starts at the vertex with time-stamp 1. Each node is stamped twice, once when it is first encountered and another time when its visit is complete.

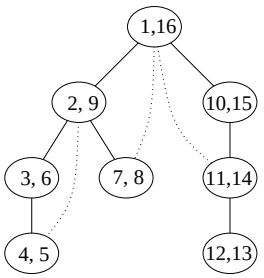

Figure 53: Tree edges are solid and back edges are dotted.

NESTING LEMMA. Vertex j is a proper descendent of vertex i in the DFS forest iff $V[i].d < V[j].d$ as well as $V[j].f < V[i].f$.

Similarly, if you have a tree and the preorder and postorder numbers of the nodes, you can determine the relation between any two nodes in constant time.

Directed graphs and relations. As mentioned earlier, we have a *directed graph* if all edges are directed. A directed graph is a way to think and talk about a mathematical relation. A typical problem where relations arise is scheduling. Some tasks are in a definite order while others are unrelated. An example is the scheduling of undergraduate computer science courses, as illustrated in Figure 54. Abstractly, a *relation* is a pair (V, E), where $V = \{0, 1, \ldots, n-1\}$ is a finite set of elements and $E \subseteq V \times V$ is a finite set of ordered pairs. Instead of

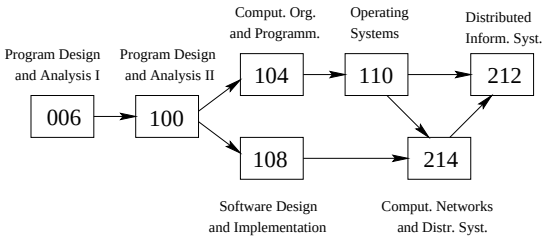

Figure 54: A subgraph of the CPS course offering. The courses CPS104 and CPS108 are incomparable, CPS104 is a predecessor of CPS110, and so on.

$(i,j) \in E$ we write $i \prec j$ and instead of (V,E) we write $(V,\prec)$. If $i \prec j$ then i is a *predecessor* of j and j is a *successor* of i. The terms relation, directed graph, digraph, and network are all synonymous.

Directed acyclic graphs. A *cycle* in a relation is a sequence $i_0 \prec i_1 \prec \ldots \prec i_k \prec i_0$. Even $i_0 \prec i_0$ is a cycle. A *linear extension* of $(V,\prec)$ is an ordering $j_0, j_1, \ldots, j_{n-1}$ of the elements that is consistent with the relation. Formally this means that $j_k \prec j_\ell$ implies $k < \ell$. A directed graph without cycle is a *directed acyclic graph*.

EXTENSION LEMMA. $(V,\prec)$ has a linear extension iff it contains no cycle.

PROOF. "$\Longrightarrow$" is obvious. We prove "$\Longleftarrow$" by induction. A vertex $s \in V$ is called a *source* if it has no predecessor. Assuming $(V,\prec)$ has no cycle, we can prove that V has a source by following edges against their direction. If we return to a vertex that has already been visited, we have a cycle and thus a contradiction. Otherwise we get stuck at a vertex s, which can only happen because s has no predecessor, which means s is a source.

Let $U = V - \{s\}$ and note that $(U,\prec)$ is a relation that is smaller than $(V,\prec)$. Hence $(U,\prec)$ has a linear extension by induction hypothesis. Call this extension X and note that s, X is a linear extension of $(V,\prec)$. □

Topological sorting with queue. The problem of constructing a linear extension is called *topological sorting*. A natural and fast algorithm follows the idea of the proof: find a source s, print s, remove s, and repeat. To expedite the first step of finding a source, each vertex maintains its number of predecessors and a queue stores all sources. First, we initialize this information.

```
forall vertices j do V[j].d = 0 endfor;
forall vertices i do
  forall successors j of i do V[j].d++ endfor
endfor;
forall vertices j do
  if V[j].d = 0 then ENQUEUE(j) endif
endfor.
```

Next, we compute the linear extension by repeated deletion of a source.

```
while queue is non-empty do
  s = DEQUEUE;
  forall successors j of s do
    V[j].d--;
    if V[j].d = 0 then ENQUEUE(j) endif
  endfor
endwhile.
```

The running time is linear in the number of vertices and edges, namely $O(n+m)$. What happens if there is a cycle in the digraph? We illustrate the above algorithm for the directed acyclic graph in Figure 55. The sequence of vertices added to the queue is also the linear extension computed by the algorithm. If the process starts at vertex a and if the successors of a vertex are ordered by name then we get a, f, d, g, c, h, b, e, which we can check is indeed a linear extension of the relation.

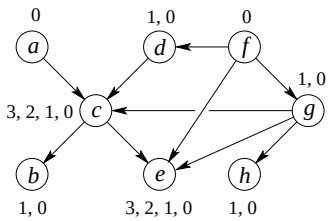

Figure 55: The numbers next to each vertex count the predecessors, which decreases during the algorithm.

Topological sorting with DFS. Another algorithm that can be used for topological sorting is depth-first search. We output a vertex when its visit has been completed, that is, when all its successors and their successors and so on have already been printed. The linear extension is therefore generated from back to front. Figure 56 shows the same digraph as Figure 55 and labels vertices with time

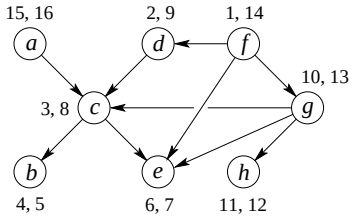

Figure 56: The numbers next to each vertex are the two time stamps applied by the depth-first search algorithm. The first number gives the time the vertex is encountered, and the second when the visit has been completed.

stamps. Consider the sequence of vertices in the order of decreasing second time stamp:

$$a(16), f(14), g(13), h(12), d(9), c(8), e(7), b(5).$$

Although this sequence is different from the one computed by the earlier algorithm, it is also a linear extension of the relation.

14 Shortest Paths

One of the most common operations in graphs is finding shortest paths between vertices. This section discusses three algorithms for this problem: breadth-first search for unweighted graphs, Dijkstra's algorithm for weighted graphs, and the Floyd-Warshall algorithm for computing distances between all pairs of vertices.

Breadth-first search. We call a graph *connected* if there is a path between every pair of vertices. A *(connected) component* is a maximal connected subgraph. Breadth-first search, or BFS, is a way to search a graph. It is similar to depth-first search, but while DFS goes as deep as quickly as possible, BFS is more cautious and explores a broad neighborhood before venturing deeper. The starting point is a vertex s. An example is shown in Figure 57. As

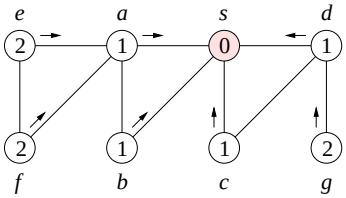

Figure 57: A sample graph with eight vertices and ten edges labeled by breath-first search. The label increases from a vertex to its successors in the search.

before, we call and edge a *tree edge* if it is traversed by the algorithm. The tree edges define the *BFS tree*, which we can use to redraw the graph in a hierarchical manner, as in Figure 58. In the case of an undirected graph, no non-tree edge can connect a vertex to an ancestor in the BFS tree. Why? We use a queue to turn the idea into an algorithm.

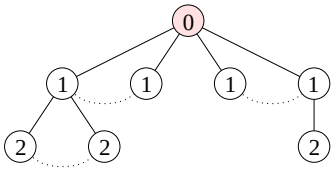

Figure 58: The tree edges in the redrawing of the graph in Figure 57 are solid, and the non-tree edges are dotted.

First, the graph and the queue are initialized.

`forall` vertices i `do` $V[i].d = -1$ `endfor`;
$V[s].d = 0$;
MAKEQUEUE; ENQUEUE(s); SEARCH.

A vertex is processed by adding its unvisited neighbors to the queue. They will be processed in turn.

```
void SEARCH
  while queue is non-empty do
    i = DEQUEUE;
    forall neighbors j of i do
      if V[j].d = −1 then
        V[j].d = V[i].d + 1; V[j].π = i;
        ENQUEUE(j)
      endif
    endfor
  endwhile.
```

The label $V[i].d$ assigned to vertex i during the traversal is the minimum number of edges of any path from s to i. In other words, $V[i].d$ is the length of the shortest path from s to i. The running time of BFS for a graph with n vertices and m edges is $O(n + m)$.

Single-source shortest path. BFS can be used to find shortest paths in unweighted graphs. We now extend the algorithm to weighted graphs. Assume V and E are the sets of vertices and edges of a simple, undirected graph with a positive weighting function $w : E \to \mathbb{R}_+$. The *length* or *weight* of a path is the sum of the weights of its edges. The *distance* between two vertices is the length of the shortest path connecting them. For a given source $s \in V$, we study the problem of finding the distances and shortest paths to all other vertices. Figure 59 illustrates the problem by showing the shortest paths to the source s. In

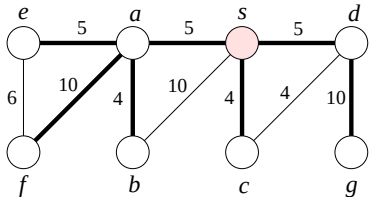

Figure 59: The bold edges form shortest paths and together the shortest path tree with root s. It differs by one edge from the breadth-first tree shown in Figure 57.

the non-degenerate case, in which no two paths have the same length, the union of all shortest paths to s is a tree, referred to as the *shortest path tree*. In the degenerate case, we can break ties such that the union of paths is a tree.

As before, we grow a tree starting from s. Instead of a queue, we use a priority queue to determine the next vertex to be added to the tree. It stores all vertices not yet in the

tree and uses $V[i].d$ for the priority of vertex i. First, we initialize the graph and the priority queue.

$V[s].d = 0$; $V[s].\pi = -1$; INSERT(s);
`forall` vertices $i \neq s$ `do`
  $V[i].d = \infty$; INSERT(i)
`endfor`.

After initialization the priority queue stores s with priority 0 and all other vertices with priority ∞.

Dijkstra's algorithm. We mark vertices in the tree to distinguish them from vertices that are not yet in the tree. The priority queue stores all unmarked vertices i with priority equal to the length of the shortest path that goes from i in one edge to a marked vertex and then to s using only marked vertices.

`while` priority queue is non-empty `do`
  $i =$ EXTRACTMIN; mark i;
  `forall` neighbors j of i `do`
    `if` j is unmarked `then`
      $V[j].d = \min\{w(ij) + V[i].d, V[j].d\}$
    `endif`
  `endfor`
`endwhile`.

Table 3 illustrates the algorithm by showing the information in the priority queue after each iteration of the while-loop operating on the graph in Figure 59. The marking mechanism is not necessary but clarifies the process. The algorithm performs n EXTRACTMIN operations and at most m DECREASEKEY operations. We compare the running time under three different data structures used to represent the priority queue. The first is a linear array, as originally proposed by Dijkstra, the second is a heap, and the third is a Fibonacci heap. The results are shown in Table 4. We get the best result with Fibonacci heaps for which the total running time is O($n \log n + m$).

s	**0**							
a	∞	5	**5**					
b	∞	10	10	9	**9**			
c	∞	**4**						
d	∞	5	5	**5**				
e	∞	∞	∞	10	10	**10**		
f	∞	∞	∞	15	15	15	**15**	
g	∞	∞	∞	∞	15	15	15	**15**

Table 3: Each column shows the contents of the priority queue. Time progresses from left to right.

	array	heap	F-heap
EXTRACTMINS	n^2	$n \log n$	$n \log n$
DECREASEKEYS	m	$m \log m$	m

Table 4: Running time of Dijkstra's algorithm for three different implementations of the priority queue holding the yet unmarked vertices.

Correctness. It is not entirely obvious that Dijkstra's algorithm indeed finds the shortest paths to s. To show that it does, we inductively prove that it maintains the following two invariants.

(A) For every unmarked vertex j, $V[j].d$ is the length of the shortest path from j to s that uses only marked vertices other than j.

(B) For every marked vertex i, $V[i].d$ is the length of the shortest path from i to s.

PROOF. Invariant (A) is true at the beginning of Dijkstra's algorithm. To show that it is maintained throughout the process, we need to make sure that shortest paths are computed correctly. Specifically, if we assume Invariant (B) for vertex i then the algorithm correctly updates the priorities $V[j].d$ of all neighbors j of i, and no other priorities change.

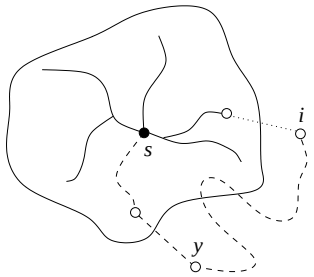

Figure 60: The vertex y is the last unmarked vertex on the hypothetically shortest, dashed path that connects i to s.

At the moment vertex i is marked, it minimizes $V[j].d$ over all unmarked vertices j. Suppose that, at this moment, $V[i].d$ is not the length of the shortest path from i to s. Because of Invariant (A), there is at least one other unmarked vertex on the shortest path. Let the last such vertex be y, as shown in Figure 60. But then $V[y].d < V[i].d$, which is a contradiction to the choice of i. □

We used (B) to prove (A) and (A) to prove (B). To make sure we did not create a circular argument, we parametrize the two invariants with the number k of vertices that are

marked and thus belong to the currently constructed portion of the shortest path tree. To prove (A_k) we need (B_k) and to prove (B_k) we need (A_{k-1}). Think of the two invariants as two recursive functions, and for each pair of calls, the parameter decreases by one and thus eventually becomes zero, which is when the argument arrives at the base case.

All-pairs shortest paths. We can run Dijkstra's algorithm n times, once for each vertex as the source, and thus get the distance between every pair of vertices. The running time is $O(n^2 \log n + nm)$ which, for dense graphs, is the same as $O(n^3)$. Cubic running time can be achieved with a much simpler algorithm using the adjacency matrix to store distances. The idea is to iterate n times, and after the k-th iteration, the computed distance between vertices i and j is the length of the shortest path from i to j that, other than i and j, contains only vertices of index k or less.

```
for k = 1 to n do
  for i = 1 to n do
    for j = 1 to n do
      A[i,j] = min{A[i,j], A[i,k] + A[k,j]}
    endfor
  endfor
endfor.
```

The only information needed to update $A[i, j]$ during the k-th iteration of the outer for-loop are its old value and values in the k-th row and the k-th column of the prior adjacency matrix. This row remains unchanged in this iteration and so does this column. We therefore do not have to use two arrays, writing the new values right into the old matrix. We illustrate the algorithm by showing the adjacency, or distance matrix before the algorithm in Figure 61 and after one iteration in Figure 62.

	s	*a*	*b*	*c*	*d*	*e*	*f*	*g*
s	0	5	10	4	5			
a	5	0	4			5	10	
b	10	4	0					
c	4			0	4			
d	5			4	0			10
e		5				0	6	
f		10				6	0	
g					10			0

Figure 61: Adjacency, or distance matrix of the graph in Figure 57. All blank entries store ∞.

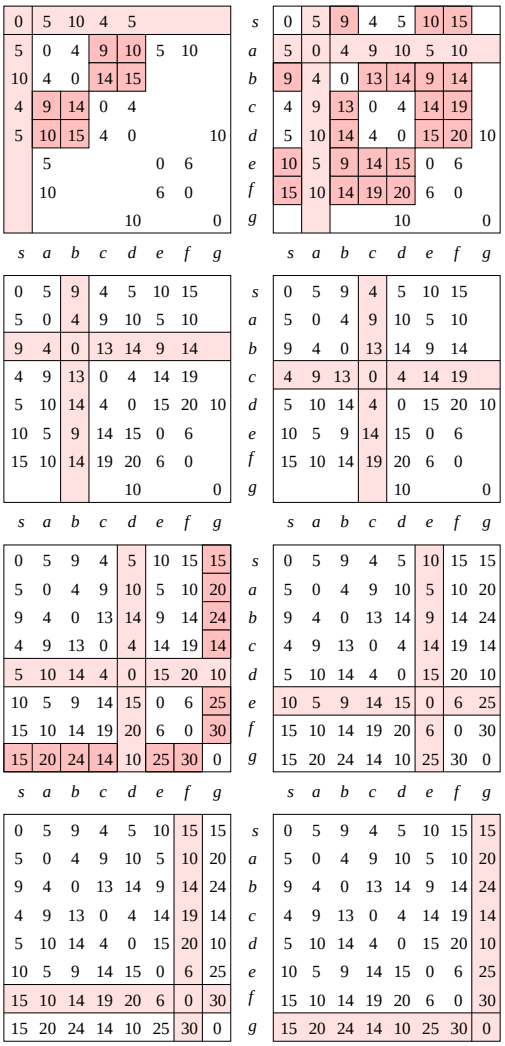

Figure 62: Matrix after each iteration. The k-th row and colum are shaded and the new, improved distances are high-lighted.

The algorithm works for weighted undirected as well as for weighted directed graphs. Its correctness is easily verified inductively. The running time is $O(n^3)$.

15 Minimum Spanning Trees

When a graph is connected, we may ask how many edges we can delete before it stops being connected. Depending on the edges we remove, this may happen sooner or later. The slowest strategy is to remove edges until the graph becomes a tree. Here we study the somewhat more difficult problem of removing edges with a maximum total weight. The remaining graph is then a tree with minimum total weight. Applications that motivate this question can be found in life support systems modeled as graphs or networks, such as telephone, power supply, and sewer systems.

Free trees. An undirected graph (U, T) is a *free tree* if it is connected and contains no cycle. We could impose a hierarchy by declaring any one vertex as the root and thus obtain a *rooted tree*. Here, we have no use for a hierarchical organization and exclusively deal with free trees. The

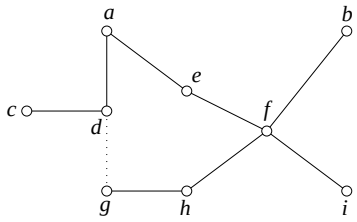

Figure 63: Adding the edge dg to the tree creates a single cycle with vertices d, g, h, f, e, a.

number of edges of a free tree is always one less than the number of vertices. Whenever we add a new edge (connecting two old vertices) we create exactly one cycle. This cycle can be destroyed by deleting any one of its edges, and we get a new free tree, as in Figure 63. Let (V, E) be a connected and undirected graph. A *subgraph* is another graph (U, T) with $U \subseteq V$ and $T \subseteq E$. It is a *spanning tree* if it is a free tree with $U = V$.

Minimum spanning trees. For the remainder of this section, we assume that we also have a weighting function, $w : E \to \mathbb{R}$. The *weight* of subgraph is then the total weight of its edges, $w(T) = \sum_{e \in T} w(e)$. A *minimum spanning tree*, or *MST* of G is a spanning tree that minimizes the weight. The definitions are illustrated in Figure 64 which shows a graph of solid edges with a minimum spanning tree of bold edges. A generic algorithm for constructing an MST grows a tree by adding more and

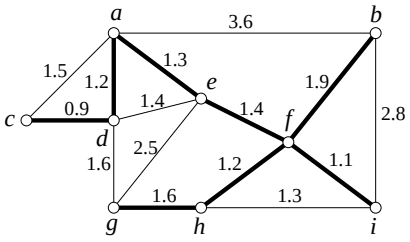

Figure 64: The bold edges form a spanning tree of weight $0.9 + 1.2 + 1.3 + 1.4 + 1.1 + 1.2 + 1.6 + 1.9 = 10.6$.

more edges. Let $A \subseteq E$ be a subset of some MST of a connected graph (V, E). An edge $uv \in E - A$ is *safe for* A if $A \cup \{uv\}$ is also subset of some MST. The generic algorithm adds safe edges until it arrives at an MST.

$A = \emptyset$;
`while` (V, A) is not a spanning tree `do`
  find a safe edge uv; $A = A \cup \{uv\}$
`endwhile`.

As long as A is a proper subset of an MST there are safe edges. Specifically, if (V, T) is an MST and $A \subseteq T$ then all edges in $T - A$ are safe for A. The algorithm will therefore succeed in constructing an MST. The only thing that is not yet clear is how to find safe edges quickly.

Cuts. To develop a mechanism for identifying safe edges, we define a *cut*, which is a partition of the vertex set into two complementary sets, $V = W \dot\cup (V - W)$. It is *crossed* by an edge $uv \in E$ if $u \in W$ and $v \in V - W$, and it *respects* an edge set A if A contains no crossing edge. The definitions are illustrated in Figure 65.

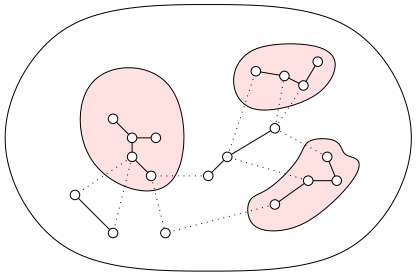

Figure 65: The vertices inside and outside the shaded regions form a cut that respects the collection of solid edges. The dotted edges cross the cut.

CUT LEMMA. Let A be subset of an MST and consider a cut $W \dot\cup (V - W)$ that respects A. If uv is a crossing edge with minimum weight then uv is safe for A.

PROOF. Consider a minimum spanning tree (V, T) with $A \subseteq T$. If $uv \in T$ then we are done. Otherwise, let $T' = T \cup \{uv\}$. Because T is a tree, there is a unique path from u to v in T. We have $u \in W$ and $v \in V - W$, so the path switches at least once between the two sets. Suppose it switches along xy, as in Figure 66. Edge xy

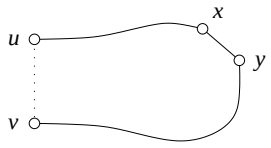

Figure 66: Adding uv creates a cycle and deleting xy destroys the cycle.

crosses the cut, and since A contains no crossing edges we have $xy \notin A$. Because uv has minimum weight among crossing edges we have $w(uv) \leq w(xy)$. Define $T'' = T' - \{xy\}$. Then (V, T'') is a spanning tree and because

$$w(T'') \;=\; w(T) - w(xy) + w(uv) \;\leq\; w(T)$$

it is a minimum spanning tree. The claim follows because $A \cup \{uv\} \subseteq T''$. □

A typical application of the Cut Lemma takes a component of (V, A) and defines W as the set of vertices of that component. The complementary set $V - W$ contains all other vertices, and crossing edges connect the component with its complement.

Prim's algorithm. Prim's algorithm chooses safe edges to grow the tree as a single component from an arbitrary first vertex s. Similar to Dijkstra's algorithm, the vertices that do not yet belong to the tree are stored in a priority queue. For each vertex i outside the tree, we define its priority $V[i].d$ equal to the minimum weight of any edge that connects i to a vertex in the tree. If there is no such edge then $V[i].d = \infty$. In addition to the priority, we store the index of the other endpoint of the minimum weight edge. We first initialize this information.

```
V[s].d = 0; V[s].π = −1; INSERT(s);
forall vertices i ≠ s do
  V[i].d = ∞; INSERT(i)
endfor.
```

The main algorithm expands the tree by one edge at a time. It uses marks to distinguish vertices in the tree from vertices outside the tree.

```
while priority queue is non-empty do
  i = EXTRACTMIN; mark i;
  forall neighbors j of i do
    if j is unmarked and w(ij) < V[j].d then
      V[j].d = w(ij); V[j].π = i
    endif
  endfor
endwhile.
```

After running the algorithm, the MST can be recovered from the π-fields of the vertices. The algorithm together with its initialization phase performs $n = |V|$ insertions into the priority queue, n extractmin operations, and at most $m = |E|$ decreasekey operations. Using the Fibonacci heap implementation, we get a running time of $O(n \log n + m)$, which is the same as for constructing the shortest-path tree with Dijkstra's algorithm.

Kruskal's algorithm. Kruskal's algorithm is another implementation of the generic algorithm. It adds edges in a sequence of non-decreasing weight. At any moment, the chosen edges form a collection of trees. These trees merge to form larger and fewer trees, until they eventually combine into a single tree. The algorithm uses a priority queue for the edges and a set system for the vertices. In this context, the term 'system' is just another word for 'set', but we will use it exclusively for sets whose elements are themselves sets. Implementations of the set system will be discussed in the next lecture. Initially, $A = \emptyset$, the priority queue contains all edges, and the system contains a singleton set for each vertex, $C = \{\{u\} \mid u \in V\}$. The algorithm finds an edge with minimum weight that connects two components defined by A. We set W equal to the vertex set of one component and use the Cut Lemma to show that this edge is safe for A. The edge is added to A and the process is repeated. The algorithm halts when only one tree is left, which is the case when A contains $n - 1 = |V| - 1$ edges.

```
A = ∅;
while |A| < n − 1 do
  uv = EXTRACTMIN;
  find P, Q ∈ C with u ∈ P and v ∈ Q;
  if P ≠ Q then
    A = A ∪ {uv}; merge P and Q
  endif
endwhile.
```

The running time is $O(m \log m)$ for the priority queue operations plus some time for maintaining C. There are two operations for the set system, namely finding the set that contains a given element, and merging two sets into one.

An example. We illustrate Kruskal's algorithm by applying it to the weighted graph in Figure 64. The sequence of edges sorted by weight is cd, fi, fh, ad, ae, hi, de, ef, ac, gh, dg, bf, eg, bi, ab. The evolution of the set system

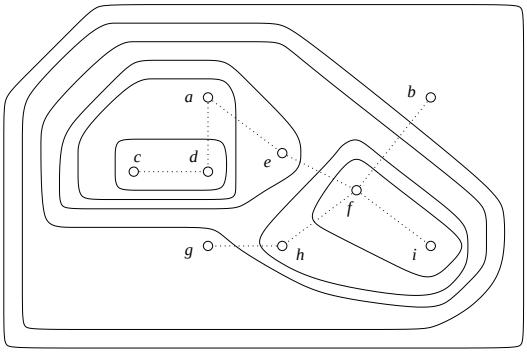

Figure 67: Eight union operations merge the nine singleton sets into one set.

is illustrated in Figure 67, and the MST computed with Kruskal's algorithm and indicated with dotted edges is the same as in Figure 64. The edges cd, fi, fh, ad, ae are all added to the tree. The next two edge, hi and de, are not added because they each have both endpoints in the same component, and adding either edge would create a cycle. Edge ef is added to the tree giving rise to a set in the system that contains all vertices other than g and b. Edge ac is not added, gh is added, dg is not, and finally bf is added to the tree. At this moment the system consists of a single set that contains all vertices of the graph.

As suggested by Figure 67, the evolution of the construction can be interpreted as a hierarchical clustering of the vertices. The specific method that corresponds to the evolution created by Kruskal's algorithm is referred to as single-linkage clustering.

16 Union-Find

In this lecture, we present two data structures for the disjoint set system problem we encountered in the implementation of Kruskal's algorithm for minimum spanning trees. An interesting feature of the problem is that m operations can be executed in a time that is only ever so slightly more than linear in m.

Abstract data type. A *disjoint set system* is an abstract data type that represents a partition C of a set $[n] = \{1, 2, \ldots, n\}$. In other words, C is a set of pairwise disjoint subsets of $[n]$ such that the union of all sets in C is $[n]$. The data type supports

`set` FIND(i): `return` $P \in C$ with $i \in P$;
`void` UNION(P, Q) : $C = C - \{P, Q\} \cup \{P \cup Q\}$.

In most applications, the sets themselves are irrelevant, and it is only important to know when two elements belong to the same set and when they belong to different sets in the system. For example, Kruskal's algorithm executes the operations only in the following sequence:

$P =$ FIND(i); $Q =$ FIND(j);
`if` $P \neq Q$ `then` UNION(P, Q) `endif`.

This is similar to many everyday situations where it is usually not important to know what it is as long as we recognize when two are the same and when they are different.

Linked lists. We construct a fairly simple and reasonably efficient first solution using linked lists for the sets. We use a table of length n, and for each $i \in [n]$, we store the name of the set that contains i. Furthermore, we link the elements of the same set and use the name of the first element as the name of the set. Figure 68 shows a sample set system and its representation. It is convenient to also store the size of the set with the first element.

To perform a UNION operation, we need to change the name for all elements in one of the two sets. To save time, we do this only for the smaller set. To merge the two lists without traversing the longer one, we insert the shorter list between the first two elements of the longer list.

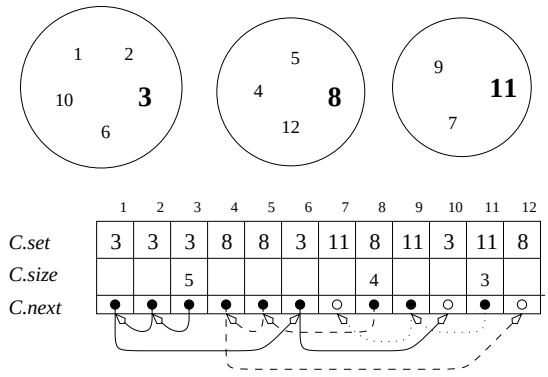

Figure 68: The system consists of three sets, each named by the bold element. Each element stores the name of its set, possibly the size of its set, and possibly a pointer to the next element in the same set.

```
void UNION(int P, Q)
  if C[P].size < C[Q].size then P ↔ Q endif;
  C[P].size = C[P].size + C[Q].size;
  second = C[P].next; C[P].next = Q; t = Q;
  while t ≠ 0 do
    C[t].set = P; u = t; t = C[t].next
  endwhile; C[u].next = second.
```

In the worst case, a single UNION operation takes time $\Theta(n)$. The amortized performance is much better because we spend time only on the elements of the smaller set.

WEIGHTED UNION LEMMA. $n - 1$ UNION operations applied to a system of n singleton sets take time $O(n \log n)$.

PROOF. For an element, i, we consider the cardinality of the set that contains it, $\sigma(i) = C[\text{FIND}(i)].size$. Each time the name of the set that contains i changes, $\sigma(i)$ at least doubles. After changing the name k times, we have $\sigma(i) \geq 2^k$ and therefore $k \leq \log_2 n$. In other words, i can be in the smaller set of a UNION operation at most $\log_2 n$ times. The claim follows because a UNION operation takes time proportional to the cardinality of the smaller set. □

Up-trees. Thinking of names as pointers, the above data structure stores each set in a tree of height one. We can use more general trees and get more efficient UNION operations at the expense of slower FIND operations. We consider a class of algorithms with the following commonalities:

- each set is a tree and the name of the set is the index of the root;
- FIND traverses a path from a node to the root;
- UNION links two trees.

It suffices to store only one pointer per node, namely the pointer to the parent. This is why these trees are called *up-trees*. It is convenient to let the root point to itself.

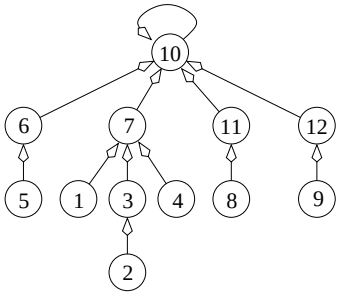

Figure 69: The UNION operations create a tree by linking the root of the first set to the root of the second set.

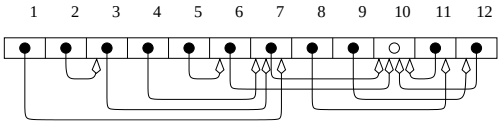

Figure 70: The table stores indices which function as pointers as well as names of elements and of sets. The white dot represents a pointer to itself.

Figure 69 shows the up-tree generated by executing the following eleven UNION operations on a system of twelve singleton sets: $2 \cup 3$, $4 \cup 7$, $2 \cup 4$, $1 \cup 2$, $4 \cup 10$, $9 \cup 12$, $12 \cup 2$, $8 \cup 11$, $8 \cup 2$, $5 \cup 6$, $6 \cup 1$. Figure 70 shows the embedding of the tree in a table. UNION takes constant time and FIND takes time proportional to the length of the path, which can be as large as $n - 1$.

Weighted union. The running time of FIND can be improved by linking smaller to larger trees. This is the ide of *weighted union* again. Assume a field $C[i].p$ for the index of the parent ($C[i].p = i$ if i is a root), and a field $C[i].$*size* for the number of elements in the tree rooted at i. We need the size field only for the roots and we need the index to the parent field everywhere except for the roots. The FIND and UNION operations can now be implemented as follows:

```
int FIND(int i)
  if C[i].p ≠ i then return FIND(C[i].p) endif;
  return i.

void UNION(int i, j)
  if C[i].size < C[j].size then i ↔ j endif;
  C[i].size = C[i].size + C[j].size; C[j].p = i.
```

The size of a subtree increases by at least a factor of 2 from a node to its parent. The depth of a node can therefore not exceed $\log_2 n$. It follows that FIND takes at most time $O(\log n)$. We formulate the result on the height for later reference.

HEIGHT LEMMA. An up-tree created from n singleton nodes by $n - 1$ weighted union operations has height at most $\log_2 n$.

Path compression. We can further improve the time for FIND operations by linking traversed nodes directly to the root. This is the idea of *path compression*. The UNION operation is implemented as before and there is only one modification in the implementation of the FIND operation:

```
int FIND(int i)
  if C[i].p ≠ i then C[i].p = FIND(C[i].p) endif;
  return C[i].p.
```

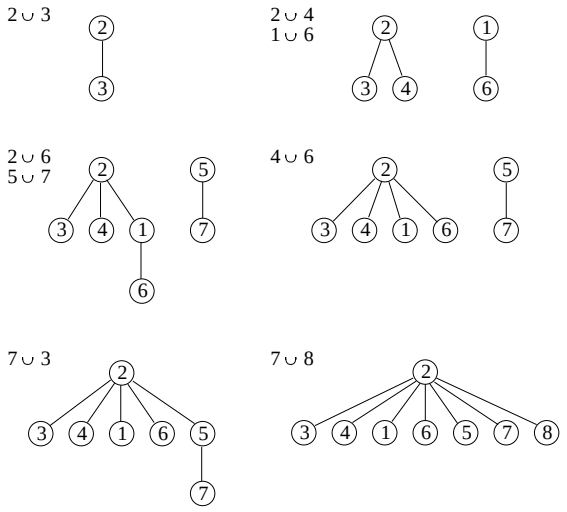

Figure 71: The operations and up-trees develop from top to bottom and within each row from left to right.

If i is not root then the recursion makes it the child of a root, which is then returned. If i is a root, it returns itself

because in this case $C[i].p = i$, by convention. Figure 71 illustrates the algorithm by executing a sequence of eight operations $i \cup j$, which is short for finding the sets that contain i and j, and performing a UNION operation if the sets are different. At the beginning, every element forms its own one-node tree. With path compression, it is difficult to imagine that long paths can develop at all.

Iterated logarithm. We will prove shortly that the iterated logarithm is an upper bound on the amortized time for a FIND operation. We begin by defining the function from its inverse. Let $F(0) = 1$ and $F(i+1) = 2^{F(i)}$. We have $F(1) = 2$, $F(2) = 2^2$, and $F(3) = 2^{2^2}$. In general, $F(i)$ is the tower of i 2s. Table 5 shows the values of F for the first six arguments. For $i \leq 3$, F is very small, but

i	0	1	2	3	4	5
F	1	2	4	16	$65,536$	$2^{65,536}$

Table 5: Values of F.

for $i = 5$ it already exceeds the number of atoms in our universe. Note that the binary logarithm of a tower of i 2s is a tower of $i-1$ 2s. The *iterated logarithm* is the number of times we can take the binary logarithm before we drop down to one or less. In other words, the iterated logarithm is the inverse of F,

$$\begin{aligned} \log^* n &= \min\{i \mid F(i) \geq n\} \\ &= \min\{i \mid \log_2 \log_2 \ldots \log_2 n \leq 1\}, \end{aligned}$$

where the binary logarithm is taken i times. As n goes to infinity, $\log^* n$ goes to infinity, but very slowly.

Levels and groups. The analysis of the path compression algorithm uses two Census Lemmas discussed shortly. Let $A_1, A_2, \ldots, A_m$ be a sequence of UNION and FIND operations, and let T be the collection of up-trees we get by executing the sequence, but *without* path compression. In other words, the FIND operations have no influence on the trees. The *level* $\lambda(\mu)$ of a node μ is its height of its subtree in T plus one.

LEVEL CENSUS LEMMA. There are at most $n/2^{\ell-1}$ nodes at level ℓ.

PROOF. We use induction to show that a node at level ℓ has a subtree of at least $2^{\ell-1}$ nodes. The claim follows because subtrees of nodes on the same level are disjoint. □

Note that if μ is a proper descendent of another node ν at some moment during the execution of the operation sequence then μ is a proper descendent of ν in T. In this case $\lambda(\mu) < \lambda(\nu)$.

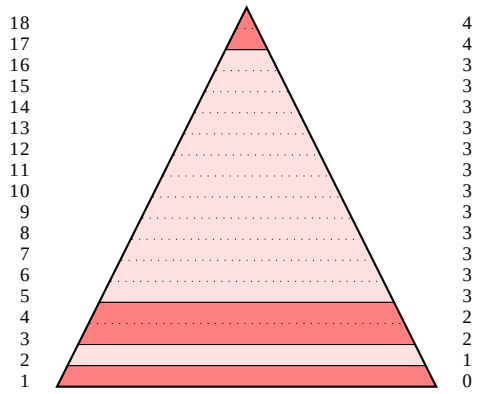

Figure 72: A schematic drawing of the tree T between the column of level numbers on the left and the column of group numbers on the right. The tree is decomposed into five groups, each a sequences of contiguous levels.

Define the *group number* of a node μ as the iterated logarithm of the level, $g(\mu) = \log^* \lambda(\mu)$. Because the level does not exceed n, we have $g(\mu) \leq \log^* n$, for every node μ in T. The definition of g decomposes an up-tree into at most $1 + \log^* n$ groups, as illustrated in Figure 72. The number of levels in group g is $F(g) - F(g-1)$, which gets large very fast. On the other hand, because levels get smaller at an exponential rate, the number of nodes in a group is not much larger than the number of nodes in the lowest level of that group.

GROUP CENSUS LEMMA. There are at most $2n/F(g)$ nodes with group number g.

PROOF. Each node with group number g has level between $F(g-1)+1$ and $F(g)$. We use the Level Census Lemma to bound their number:

$$\begin{aligned} \sum_{\ell=F(g-1)+1}^{F(g)} \frac{n}{2^{\ell-1}} &\leq \frac{n \cdot (1 + \frac{1}{2} + \frac{1}{4} + \ldots)}{2^{F(g-1)}} \\ &= \frac{2n}{F(g)}, \end{aligned}$$

as claimed. □

Analysis. The analysis is based on the interplay between the up-trees obtained with and without path compression.

The latter are constructed by the weighted union operations and eventually form a single tree, which we denote as T. The former can be obtained from the latter by the application of path compression. Note that in T, the level strictly increases from a node to its parent. Path compression preserves this property, so levels also increase when we climb a path in the actual up-trees.

We now show that any sequence of $m \geq n$ UNION and FIND operations on a ground set $[n]$ takes time at most $O(m \log^* n)$ if weighted union and path compression is used. We can focus on FIND because each UNION operation takes only constant time. For a FIND operation A_i, let X_i be the set of nodes along the traversed path. The total time for executing all FIND operations is proportional to

$$x = \sum_i |X_i|.$$

For $\mu \in X_i$, let $p_i(\mu)$ be the parent during the execution of A_i. We partition X_i into the topmost two nodes, the nodes just below boundaries between groups, and the rest:

$$\begin{aligned} Y_i &= \{\mu \in X_i \mid \mu \text{ is root or child of root}\}, \\ Z_i &= \{\mu \in X_i - Y_i \mid g(\mu) < g(p_i(\mu))\}, \\ W_i &= \{\mu \in X_i - Y_i \mid g(\mu) = g(p_i(\mu))\}. \end{aligned}$$

Clearly, $|Y_i| \leq 2$ and $|Z_i| \leq \log^* n$. It remains to bound the total size of the W_i, $w = \sum_i |W_i|$. Instead of counting, for each A_i, the nodes in W_i, we count, for each node μ, the FIND operations A_j for which $\mu \in W_j$. In other words, we count how often μ can change parent until its parent has a higher group number than μ. Each time μ changes parent, the new parent has higher level than the old parent. If follows that the number of changes is at most $F(g(\mu)) - F(g(\mu) - 1)$. The number of nodes with group number g is at most $2n/F(g)$ by the Group Census Lemma. Hence

$$\begin{aligned} w &\leq \sum_{g=0}^{\log^* n} \frac{2n}{F(g)} \cdot (F(g) - F(g-1)) \\ &\leq 2n \cdot (1 + \log^* n). \end{aligned}$$

This implies that

$$\begin{aligned} x &\leq 2m + m \log^* n + 2n(1 + \log^* n) \\ &= O(m \log^* n), \end{aligned}$$

assuming $m \geq n$. This is an upper bound on the total time it takes to execute m FIND operations. The amortized cost per FIND operation is therefore at most $O(\log^* n)$, which for all practical purposes is a constant.

Summary. We proved an upper bound on the time needed for $m \geq n$ UNION and FIND operations. The bound is more than constant per operation, although for all practical purposes it is constant. The $\log^* n$ bound can be improved to an even smaller function, usually referred to as $\alpha(n)$ or the inverse of the Ackermann function, that goes to infinity even slower than the iterated logarithm. It can also be proved that (under some mild assumptions) there is no algorithm that can execute general sequences of UNION and FIND operations in amortized time that is asymptotically less than $\alpha(n)$.

Fourth Homework Assignment

Write the solution to each problem on a single page. The deadline for handing in solutions is October 30.

Problem 1. ($20 = 10 + 10$ points). Consider a free tree and let $d(u, v)$ be the number of edges in the path connecting u to v. The *diameter* of the tree is the maximum $d(u, v)$ over all pairs of vertices in the tree.

(a) Give an efficient algorithm to compute the diameter of a tree.

(b) Analyze the running time of your algorithm.

Problem 2. (20 points). Design an efficient algorithm to find a spanning tree for a connected, weighted, undirected graph such that the weight of the maximum weight edge in the spanning tree is minimized. Prove the correctness of your algorithm.

Problem 3. ($7 + 6 + 7$ points). A weighted graph $G = (V, E)$ is a *near-tree* if it is connected and has at most $n + 8$ edges, where n is the number of vertices. Give an $O(n)$-time algorithm to find a minimum weight spanning tree for G.

Problem 4. ($10 + 10$ points). Given an undirected weighted graph and vertices s, t, design an algorithm that computes the number of shortest paths from s to t in the case:

(a) All weights are positive numbers.

(b) All weights are real numbers.

Analyze your algorithm for both (a) and (b).

Problem 5. ($20 = 10 + 10$ points). The *off-line minimum problem* is about maintaining a subset of $[n] = \{1, 2, \ldots, n\}$ under the operations INSERT and EXTRACTMIN. Given an interleaved sequence of n insertions and m min-extractions, the goal is to determine which key is returned by which min-extraction. We assume that each element $i \in [n]$ is inserted exactly once. Specifically, we wish to fill in an array $E[1..m]$ such that $E[i]$ is the key returned by the i-th min-extraction. Note that the problem is *off-line*, in the sense that we are allowed to process the entire sequence of operations before determining any of the returned keys.

(a) Describe how to use a union-find data structure to solve the problem efficiently.

(b) Give a tight bound on the worst-case running time of your algorithm.

V TOPOLOGICAL ALGORITHMS

17 Geometric Graphs

In the abstract notion of a graph, an edge is merely a pair of vertices. The geometric (or topological) notion of a graph is closer to our intuition in which we think of an edge as a curve that connects two vertices.

Embeddings. Let $G = (V, E)$ be a simple, undirected graph and write $\mathbb{R}^2$ for the two-dimensional real plane. A *drawing* maps every vertex $v \in V$ to a point $\varepsilon(v)$ in $\mathbb{R}^2$, and it maps every edge $\{u, v\} \in E$ to a curve with endpoints $\varepsilon(u)$ and $\varepsilon(v)$. The drawing is an *embedding* if

1. different vertices map to different points;
2. the curves have no self-intersections;
3. the only points of a curve that are images of vertices are its endpoints;
4. two curves intersect at most in their endpoints.

We can always map the vertices to points and the edges to curves in $\mathbb{R}^3$ so they form an embedding. On the other hand, not every graph has an embedding in $\mathbb{R}^2$. The graph G is *planar* if it has an embedding in $\mathbb{R}^2$. As illustrated in Figure 73, a planar graph has many drawings, not all of which are embeddings. A *straight-line* drawing or embedding is one in which each edge is mapped to a straight line segment. It is uniquely determined by the mapping of the vertices, $\varepsilon : V \to \mathbb{R}^2$. We will see later that every planar graph has a straight-line embedding.

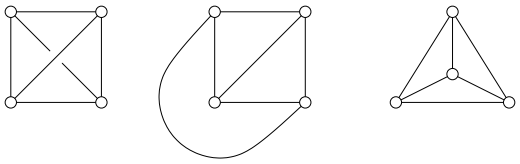

Figure 73: Three drawings of K_4, the complete graph with four vertices. From left to right: a drawing that is not an embedding, an embedding with one curved edge, a straight-line embedding.

Euler's formula. A *face* of an embedding ε of G is a component of the thus defined decomposition of $\mathbb{R}^2$. We write $n = |V|$, $m = |E|$, and ℓ for the number of faces. Euler's formula says these numbers satisfy a linear relation.

EULER'S FORMULA. If G is connected and ε is an embedding of G in $\mathbb{R}^2$ then $n - m + \ell = 2$.

PROOF. Choose a spanning tree (V, T) of $G = (V, E)$. It has n vertices, $|T| = n - 1$ edges, and one (unbounded) face. We have $n - (n-1) + 1 = 2$, which proves the formula if G is a tree. Otherwise, draw the remaining edges, one at a time. Each edge decomposes one face into two. The number of vertices does not change, m increases by one, and ℓ increases by one. Since the graph satisfies the linear relation before drawing the edge, it satisfies the relation also after drawing the edge. □

A planar graph is *maximally connected* if adding any one new edge violates planarity. Not surprisingly, a planar graph of three or more vertices is maximally connected iff every face in an embedding is bounded by three edges. Indeed, suppose there is a face bounded by four or more edges. Then we can find two vertices in its boundary that are not yet connected and we can connect them by drawing a curve that passes through the face; see Figure 74. For obvious reasons, we call an embedding of a maximally connected planar graph with $n \geq 3$ vertices a *triangulation*. For such graphs, we have an additional linear relation, namely $3\ell = 2m$. We can thus rewrite Euler's formula and get $n - m + \frac{2m}{3} = 2$ and $n - \frac{3\ell}{2} + \ell = 2$ and therefore

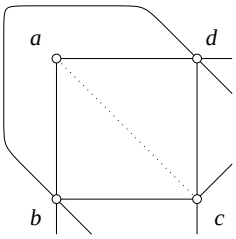

Figure 74: Drawing the edge from a to c decomposes the quadrangle into two triangles. Note that we cannot draw the edge from b to d since it already exists outside the quadrangle.

$$\begin{aligned} m &= 3n - 6; \\ \ell &= 2n - 4, \end{aligned}$$

Every planar graph can be completed to a maximally connected planar graph. For $n \geq 3$ this implies that the planar graph has at most $3n - 6$ edges and at most $2n - 4$ faces.

Forbidden subgraphs. We can use Euler's relation to prove that the complete graph of five vertices is not planar. It has $n = 5$ vertices and $m = 10$ edges, contradicting the upper bound of at most $3n - 6 = 9$ edges. Indeed, every drawing of K_5 has at least two edges crossing; see Figure 75. Similarly, we can prove that the complete bipartite

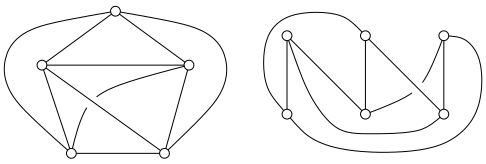

Figure 75: A drawing of K_5 on the left and of $K_{3,3}$ on the right.

graph with three plus three vertices is not planar. It has $n = 6$ vertices and $m = 9$ edges. Every cycle in a bipartite graph has an even number of edges. Hence, $4\ell \leq 2m$. Plugging this into Euler's formula, we get $n - m + \frac{m}{2} \geq 2$ and therefore $m \leq 2n - 4 = 8$, again a contradiction.

In a sense, K_5 and $K_{3,3}$ are the quintessential non-planar graphs. To make this concrete, we still need an operation that creates or removes degree-2 vertices. Two graphs are *homeomorphic* if one can be obtained from the other by a sequence of operations, each deleting a degree-2 vertex and replacing its two edges by the one that connects its two neighbors, or the other way round.

KURATOWSKI'S THEOREM. A graph G is planar iff no subgraph of G is homeomorphic to K_5 or to $K_{3,3}$.

The proof of this result is a bit lengthy and omitted.

Pentagons are star-convex. Euler's formula can also be used to show that every planar graph has a straight-line embedding. Note that the sum of vertex degrees counts each edge twice, that is, $\sum_{v \in V} \deg(v) = 2m$. For planar graphs, twice the number of edges is less than $6n$ which implies that the average degree is less than six. It follows that every planar graph has at least one vertex of degree 5 or less. This can be strengthened by saying that every planar graph with $n \geq 4$ vertices has at least four vertices of degree at most 5 each. To see this, assume the planar graph is maximally connected and note that every vertex has degree at least 3. The deficiency from degree 6 is thus at most 3. The total deficiency is $6n - \sum_{v \in V} \deg(v) = 12$ which implies that we have at least four vertices with positive deficiency.

We need a little bit of geometry to prepare the construction of a straight-line embedding. A region $R \subseteq \mathbb{R}^2$ is *convex* if $x, y \in R$ implies that the entire line segment connecting x and y is contained in R. Figure 76 shows regions of either kind. We call R *star-convex* of there is a point $z \in R$ such that for every point $x \in R$ the line segment connecting x with z is contained in R. The set of

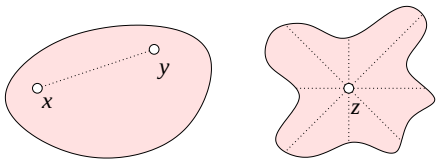

Figure 76: A convex region on the left and a non-convex star-convex region on the right.

such points z is the *kernel* of R. Clearly, every convex region is star-convex but not every star-convex region is convex. Similarly, there are regions that are not star-convex, even rather simple ones such as the hexagon in Figure 77. However, every pentagon is star-convex. Indeed, the pentagon can be decomposed into three triangles by drawing two diagonals that share an endpoint. Extending the incident sides into the pentagon gives locally the boundary of the kernel. It follows that the kernel is non-empty and has interior points.

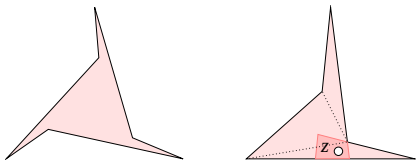

Figure 77: A non-star-convex hexagon on the left and a star-convex pentagon on the right. The dark region inside the pentagon is its kernel.

Fáry's construction. We construct a straight-line embedding of a planar graph $G = (V, E)$ assuming G is maximally connected. Choose three vertices, a, b, c, connected by three edges to form the outer triangle. If G has only $n = 3$ vertices we are done. Else it has at least one vertex $u \in V = \{a, b, c\}$ with $\deg(u) \leq 5$.

`Step 1.` Remove u together with the $k = \deg(u)$ edges incident to u. Add $k - 3$ edges to make the graph maximally connected again.

`Step 2.` Recursively construct a straight-line embedding of the smaller graph.

`Step 3.` Remove the added $k - 3$ edges and map u to a point $\varepsilon(u)$ in the interior of the kernel of the resulting k-gon. Connect $\varepsilon(u)$ with line segments to the vertices of the k-gon.

Figure 78 illustrates the recursive construction. It is straightforward to implement but there are numerical issues in the choice of $\varepsilon(u)$ that limit the usefulness of this construction.

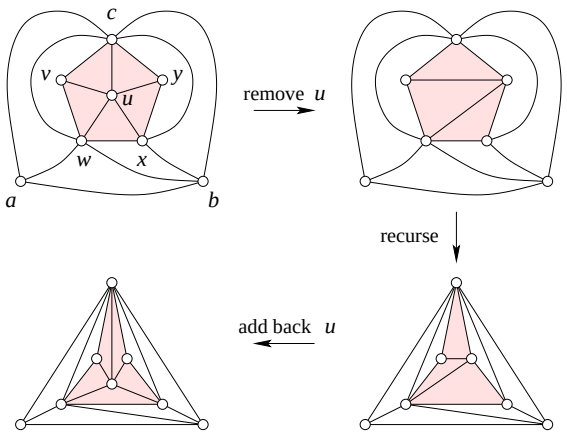

Figure 78: We fix the outer triangle, remove the degree-5 vertex, recursively construct a straight-line embedding of the rest, and finally add the vertex back.

Tutte's construction. A more useful construction of a straight-line embedding goes back to the work of Tutte. We begin with a definition. Given a finite set of points, $x_1, x_2, \ldots, x_j$, the *average* is

$$x \quad = \quad \frac{1}{n} \sum_{i=1}^{j} x_i.$$

For $j = 2$, it is the midpoint of the edge and for $j = 3$, it is the centroid of the triangle. In general, the average is a point somewhere between the x_i. Let $G = (V, E)$ be a maximally connected planar graph and a, b, c three vertices connected by three edges. We now follow Tutte's construction to get a mapping $\varepsilon : V \to \mathbb{R}^2$ so that the straight-line drawing of G is a straight-line embedding.

`Step 1.` Map a, b, c to points $\varepsilon(a), \varepsilon(b), \varepsilon(c)$ spanning a triangle in $\mathbb{R}^2$.

`Step 2.` For each vertex $u \in V - \{a, b, c\}$, let N_u be the set of neighbors of u. Map u to the average of the images of its neighbors, that is,

$$\varepsilon(u) \quad = \quad \frac{1}{|N_u|} \sum_{v \in N_u} \varepsilon(v).$$

The fact that the resulting mapping $\varepsilon : V \to \mathbb{R}^2$ gives a straight-line embedding of G is known as Tutte's Theorem. It holds even if G is not quite maximally connected and if the points are not quite the averages of their neighbors. The proof is a bit involved and omitted.

The points $\varepsilon(u)$ can be computed by solving a system of linear equations. We illustrate this for the graph in Figure 78. We set $\varepsilon(a) = \binom{-1}{-1}$, $\varepsilon(b) = \binom{1}{-1}$, $\varepsilon(c) = \binom{0}{1}$. The other five points are computed by solving the system of linear equations $\mathbf{A}\mathbf{v} = 0$, where

$$\mathbf{A} \quad = \quad \begin{bmatrix} 0 & 0 & 1 & -5 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & -3 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & -6 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & -5 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & -3 \end{bmatrix}$$

and $\mathbf{v}$ is the column vector of points $\varepsilon(a)$ to $\varepsilon(y)$. There are really two linear systems, one for the horizontal and the other for the vertical coordinates. In each system, we have $n - 3$ equations and a total of $n - 3$ unknowns. This gives a unique solution provided the equations are linearly independent. Proving that they are is part of the proof of Tutte's Theorem. Solving the linear equations is a numerical problem that is studies in detail in courses on numerical analysis.

18 Surfaces

Graphs may be drawn in two, three, or higher dimensions, but they are still intrinsically only 1-dimensional. One step up in dimensions, we find surfaces, which are 2-dimensional.

Topological 2-manifolds. The simplest kind of surfaces are the ones that on a small scale look like the real plane. A space $\mathbb{M}$ is a 2-*manifold* if every point $x \in \mathbb{M}$ is locally homeomorphic to $\mathbb{R}^2$. Specifically, there is an open neighborhood N of x and a continuous bijection $h : N \to \mathbb{R}^2$ whose inverse is also continuous. Such a bicontinuous map is called a *homeomorphism*. Examples of 2-manifolds are the open disk and the sphere. The former is not compact because it has covers that do not have finite subcovers. Figure 79 shows examples of compact 2-manifolds. If we add the boundary circle to the open disk

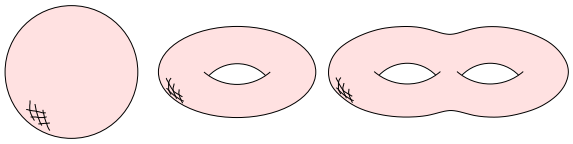

Figure 79: Three compact 2-manifolds, the sphere, the torus, and the double torus.

we get a closed disk which is compact but not every point is locally homeomorphic to $\mathbb{R}^2$. Specifically, a point on the circle has an open neighborhood homeomorphic to the closed half-plane, $\mathbb{H}^2 = \{(x_1, x_2) \in \mathbb{R}^2 \mid x_1 \geq 0\}$. A space whose points have open neighborhoods homeomorphic to $\mathbb{R}^2$ or $\mathbb{H}^2$ is called a 2-*manifolds with boundary*; see Figure 80 for examples. The *boundary* is the subset

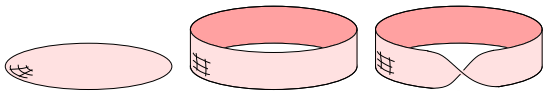

Figure 80: Three 2-manifolds with boundary, the closed disk, the cylinder, and the Möbius strip.

of points with neighborhoods homeomorphic to $\mathbb{H}^2$. It is a 1-manifold (without boundary), that is, every point is locally homeomorphic to $\mathbb{R}$. There is only one type of compact, connected 1-manifold, namely the closed curve. In topology, we do not distinguish spaces that are homeomorphic to each other. Hence, every closed curve is like every other one and they are all homeomorphic to the unit circle, $\mathbb{S}^1 = \{x \in \mathbb{R}^2 \mid \|x\| = 1\}$.

Triangulations. A standard representation of a compact 2-manifold uses triangles that are glued to each other along shared edges and vertices. A collection K of triangles, edges, and vertices is a *triangulation* of a compact 2-manifold if

I. for every triangle in K, its three edges belong to K, and for every edge in K, its two endpoints are vertices in K;

II. every edge belongs to exactly two triangles and every vertex belongs to a single ring of triangles.

An example is shown in Figure 81. To simplify language, we call each element of K a *simplex*. If we need to be specific, we add the dimension, calling a vertex a 0-simplex, an edge a 1-simplex, and a triangle a 2-simplex. A *face* of a simplex τ is a simplex $\sigma \subseteq \tau$. For example, a triangle has seven faces, its three vertices, its two edges, and itself. We can now state Condition I more succinctly: if σ is a face of τ and $\tau \in K$ then $\sigma \in K$. To talk about

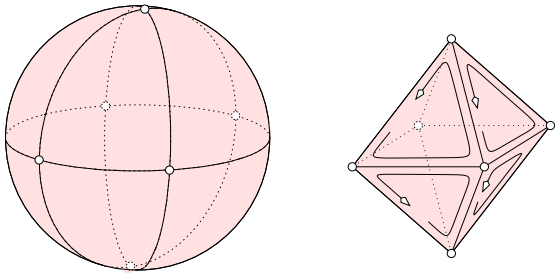

Figure 81: A triangulation of the sphere. The eight triangles are glued to form the boundary of an octahedron which is homeomorphic to the sphere.

the inverse of the face relation, we define the *star* of a simplex σ as the set of simplices that contain σ as a face, $\operatorname{St}\sigma = \{\tau \in K \mid \sigma \subseteq \tau\}$. Sometimes we think of the star as a set of simplices and sometimes as a set of points, namely the union of interiors of the simplices in the star. With the latter interpretation, we can now express Condition II more succinctly: the star of every simplex in K is homeomorphic to $\mathbb{R}^2$.

Data structure. When we store a 2-manifold, it is useful to keep track of which side we are facing and where we are going so that we can move around efficiently. The core piece of our data structure is a representation of the symmetry group of a triangle. This group is isomorphic to the group of permutations of three elements,

the vertices of the triangle. We call each permutation an *ordered triangle* and use cyclic shifts and transpositions to move between them; see Figure 82. We store

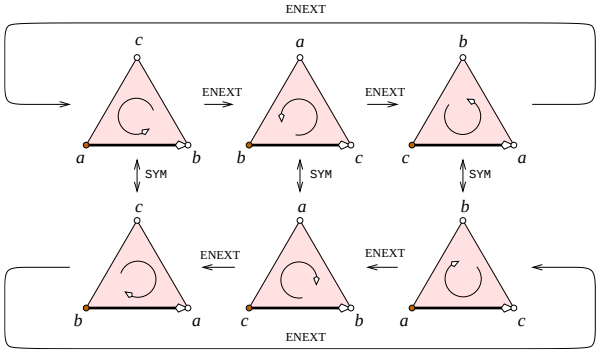

Figure 82: The symmetry group of the triangle consists of six ordered versions. Each ordered triangle has a lead vertex and a lead directed edge.

the entire symmetry group in a single node of an abstract graph, with arcs between neighboring triangles. Furthermore, we store the vertices in a linear array, $V[1..n]$. For each ordered triangle, we store the index of the lead vertex and a pointer to the neighboring triangle that shares the same directed lead edge. A pointer in this context is the address of a node together with a three-bit integer, ι, that identifies the ordered version of the triangle we refer to. Suppose for example that we identify the ordered versions $abc, bca, cab, bac, cba, acb$ of a triangle with $\iota = 0, 1, 2, 4, 5, 6$, in this sequence. Then we can move between different ordered versions of the same triangle using the following functions.

```
ordTri ENEXT(μ, ι)
  if ι ≤ 2 then return (μ, (ι + 1) mod 3)
           else return (μ, (ι + 1) mod 3 + 4)
  endif.

ordTri SYM(μ, ι)
  return (μ, (ι + 4) mod 8).
```

To get the index of the lead vertex, we use the integer function $\text{ORG}(\mu, \iota)$ and to get the pointer to the neighboring triangle, we use $\text{FNEXT}(\mu, \iota)$.

Orientability. A 2-manifold is *orientable* if it has two distinct sides, that is, if we move around on one we stay there and never cross over to the other side. The one example of a non-orientable manifold we have seen so far is the Möbious strip in Figure 80. There are also non-orientable, compact 2-manifolds (without boundary), as we can see in Figure 83. We use the data structure to decide whether or

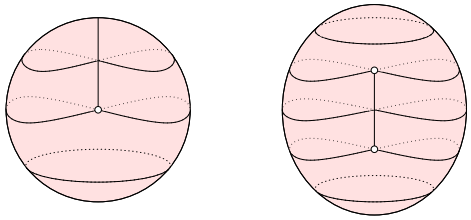

Figure 83: Two non-orientable, compact 2-manifolds, the projective plane on the left and the Klein bottle on the right.

not a 2-manifold is orientable. Note that the cyclic shift partitions the set of six ordered triangles into two *orientations*, each consisting of three triangles. We say two neighboring triangles are *consistently oriented* if they disagree on the direction of the shared edge, as in Figure 81. Using depth-first search, we visit all triangles and orient them consistently, if possible. At the first visit, we orient the triangle consistent with the preceding, neighboring triangle. At subsequence visits, we check for consistent orientation.

```
boolean ISORNTBL(μ, ι)
  if μ is unmarked then
    mark μ; choose the orientation that contains ι;
    b_x = ISORNTBL(FNEXT(SYM(μ, ι)));
    b_y = ISORNTBL(FNEXT(ENEXT(SYM(μ, ι))));
    b_z = ISORNTBL(FNEXT(ENEXT^2(SYM(μ, ι))));
    return b_x and b_y and b_z
  else
    return [orientation of μ contains ι]
  endif.
```

There are two places where we return a boolean value. At the second place, it indicates whether or not we have consistent orientation in spite of the visited triangle being oriented prior to the visit. At the first place, the boolean value indicates whether or not we have found a contradiction to orientablity so far. A value of FALSE anywhere during the computation is propagated to the root of the search tree telling us that the 2-manifold is non-orientable. The running time is proportional to the number of triangles in the triangulation of the 2-manifold.

Classification. For the sphere and the torus, it is easy to see how to make them out of a sheet of paper. Twisting the paper gives a non-orientable 2-manifold. Perhaps

most difficult to understand is the projective plane. It is obtained by gluing each point of the sphere to its antipodal point. This way, the entire northern hemisphere is glued to the southern hemisphere. This gives the disk except that we still need to glue points of the bounding circle (the equator) in pairs, as shown in the third paper construction in Figure 84. The Klein bottle is easier to imagine as it is obtained by twisting the paper just once, same as in the construction of the Möbius strip.

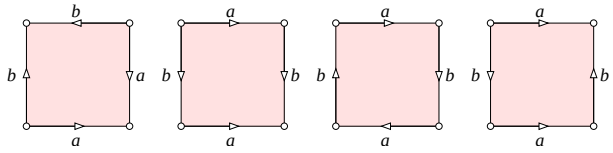

Figure 84: From left to right: the sphere, the torus, the projective plane, and the Klein bottle.

There is a general method here that can be used to classify the compact 2-manifolds. Given two of them, we construct a new one by removing an open disk each and gluing the 2-manifolds along the two circles. The result is called the *connected sum* of the two 2-manifolds, denoted as $\mathbb{M}\#\mathbb{N}$. For example, the double torus is the connected sum of two tori, $\mathbb{T}^2\#\mathbb{T}^2$. We can cut up the g-fold torus into a flat sheet of paper, and the canonical way of doing this gives a $4g$-gon with edges identified in pairs as shown in Figure 85 on the left. The number g is called the *genus* of the manifold. Similarly, we can get new non-orientable

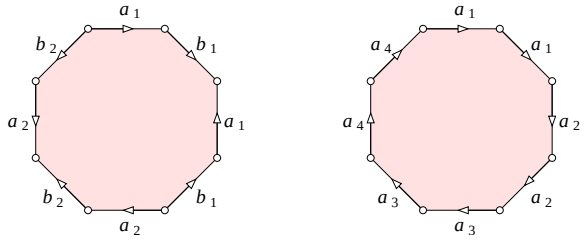

Figure 85: The polygonal schema in standard form for the double torus and the double Klein bottle.

manifolds from the projective plane, $\mathbb{P}^2$, by forming connected sums. Cutting up the g-fold projective plane gives a $2g$-gon with edges identified in pairs as shown in Figure 85 on the right. We note that the constructions of the projective plane and the Klein bottle in Figure 84 are both not in standard form. A remarkable result which is now more than a century old is that every compact 2-manifold can be cut up to give a standard polygonal schema. This implies a classification of the possibilities.

CLASSIFICATION THEOREM. The members of the families $\mathbb{S}^2, \mathbb{T}^2, \mathbb{T}^2\#\mathbb{T}^2, \ldots$ and $\mathbb{P}^2, \mathbb{P}^2\#\mathbb{P}^2, \ldots$ are non-homeomorphic and they exhaust the family of compact 2-manifolds.

Euler characteristic. Suppose we are given a triangulation, K, of a compact 2-manifold, $\mathbb{M}$. We already know how to decide whether or not $\mathbb{M}$ is orientable. To determine its type, we just need to find its genus, which we do by counting simplices. The *Euler characteristic* is

$$\chi \quad = \quad \#\text{vertices} - \#\text{edges} + \#\text{triangles}.$$

Let us look at the orientable case first. We have a $4g$-gon which we triangulate. This is a planar graph with $n - m + \ell = 2$. However, $2g$ edge are counted double, the $4g$ vertices of the $4g$-gon are all the same, and the outer face is not a triangle in K. Hence,

$$\begin{aligned} \chi &= (n - 4g + 1) - (m - 2g) + (\ell - 1) \\ &= (n - m + \ell) - 2g \end{aligned}$$

which is equal to $2 - 2g$. The same analysis can be used in the non-orientable case in which we get $\chi = (n - 2g + 1) - (m - g) + (\ell - 1) = 2 - g$. To decide whether two compact 2-manifolds are homeomorphic it suffices to determine whether they are both orientable or both non-orientable and, if they are, whether they have the same Euler characteristic. This can be done in time linear in the number of simplices in their triangulations.

This result is in sharp contrast to the higher-dimensional case. The classification of compact 3-manifolds has been a longstanding open problem in Mathematics. Perhaps the recent proof of the Poincaré conjecture by Perelman brings us close to a resolution. Beyond three dimensions, the situation is hopeless, that is, deciding whether or not two triangulated compact manifolds of dimension four or higher are homeomorphic is undecidable.

19 Homology

In topology, the main focus is not on geometric size but rather on how a space is connected. The most elementary notion distinguishes whether we can go from one place to another. If not then there is a gap we cannot bridge. Next we would ask whether there is a loop going around an obstacle, or whether there is a void missing in the space. Homology is a formalization of these ideas. It gives a way to define and count holes using algebra.

The cyclomatic number of a graph. To motivate the more general concepts, consider a connected graph, G, with n vertices and m edges. A spanning tree has $n-1$ edges and every additional edge forms a unique cycle together with edges in this tree; see Figure 86. Every other

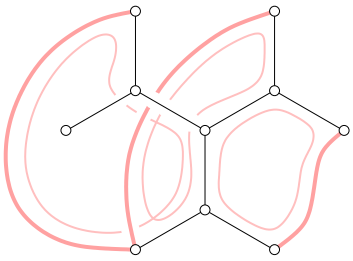

Figure 86: A tree with three additional edges defining the same number of cycles.

cycle in G can be written as a sum of these $m-(n-1)$ cycles. To make this concrete, we define a *cycle* as a subset of the edges such that every vertex belongs to an even number of these edges. A cycle does not need to be connected. The *sum* of two cycles is the symmetric difference of the two sets such that multiple edges erase each other in pairs. Clearly, the sum of two cycles is again a cycle. Every cycle, γ, in G contains some positive number of edges that do not belong to the spanning tree. Calling these edges $e_1, e_2, \ldots, e_k$ and the cycles they define $\gamma_1, \gamma_2, \ldots, \gamma_k$, we claim that

$$\gamma \quad = \quad \gamma_1 + \gamma_2 + \ldots + \gamma_k.$$

To see this assume that $\delta = \gamma_1 + \gamma_2 + \ldots + \gamma_k$ is different from γ. Then $\gamma + \delta$ is again a cycle but it contains no edges that do not belong to the spanning tree. Hence $\gamma + \delta = \emptyset$ and therefore $\gamma = \delta$, as claimed. This implies that the $m-n+1$ cycles form a basis of the group of cycles which motivates us to call $m-n+1$ the *cyclomatic number* of the graph. Note that the basis depends on the choice of spanning tree while the cyclomatic number is independent of that choice.

Simplicial complexes. We begin with a combinatorial representation of a topological space. Using a finite ground set of vertices, V, we call a subset $\sigma \subseteq V$ an *abstract simplex*. Its *dimension* is one less than the cardinality, $\dim \sigma = |\sigma| - 1$. A *face* is a subset $\tau \subseteq \sigma$.

DEFINITION. An *abstract simplicial complex* over V is a system $K \subseteq 2^V$ such that $\sigma \in K$ and $\tau \subseteq \sigma$ implies $\tau \in K$.

The *dimension* of K is the largest dimension of any simplex in K. A graph is thus a 1-dimensional abstract simplicial complex. Just like for graphs, we sometimes think of K as an abstract structure and at other times as a geometric object consisting of geometric simplices. In the latter interpretation, we glue the simplices along shared faces to form a *geometric realization* of K, denoted as $|K|$. We say K *triangulates* a space $\mathbb{X}$ if there is a homeomorphism $h : \mathbb{X} \to |K|$. We have seen 1- and 2-dimensional examples in the preceding sections. The *boundary* of a simplex σ is the collection of co-dimension one faces,

$$\partial\sigma \quad = \quad \{\tau \subseteq \sigma \mid \dim \tau = \dim \sigma - 1\}.$$

If $\dim \sigma = p$ then the boundary consists of $p+1$ $(p-1)$-simplices. Every $(p-1)$-simplex has p $(p-2)$-simplices in its own boundary. This way we get $(p+1)p$ $(p-2)$-simplices, counting each of the $\binom{p+1}{p-1} = \binom{p+1}{2}$ $(p-2)$-dimensional faces of σ twice.

Chain complexes. We now generalize the cycles in graphs to cycles of different dimensions in simplicial complexes. A *p-chain* is a set of p-simplices in K. The *sum* of two p-chains is their symmetric difference. We usually write the sets as formal sums,

$$\begin{aligned} c &= a_1\sigma_1 + a_2\sigma_2 + \ldots + a_n\sigma_n; \\ d &= b_1\sigma_1 + b_2\sigma_2 + \ldots + b_n\sigma_n, \end{aligned}$$

where the a_i and b_i are either 0 or 1. Addition can then be done using modulo 2 arithmetic,

$$c +_2 d \quad = \quad (a_1 +_2 b_1)\sigma_1 + \ldots + (a_n +_2 b_n)\sigma_n,$$

where $a_i +_2 b_i$ is the exclusive or operation. We simplify notation by dropping the subscript but note that the two plus signs are different, one modulo two and the other a formal notation separating elements in a set. The p-chains

form a group, which we denote as $(\mathsf{C}_p, +)$ or simply C_p. Note that the boundary of a p-simplex is a $(p-1)$-chain, an element of C_{p-1}. Extending this concept linearly, we define the boundary of a p-chain as the sum of boundaries of its simplices, $\partial c = a_1\partial\sigma_1 + \ldots + a_n\partial\sigma_n$. The boundary is thus a map between chain groups and we sometimes write the dimension as index for clarity,

$$\partial_p : \mathsf{C}_p \to \mathsf{C}_{p-1}.$$

It is a homomorphism since $\partial_p(c+d) = \partial_p c + \partial_p d$. The infinite sequence of chain groups connected by boundary homomorphisms is called the *chain complex* of K. All groups of dimension smaller than 0 and larger than the dimension of K are trivial. It is convenient to keep them around to avoid special cases at the ends. A *p-cycle* is a p-chain whose boundary is zero. The sum of two p-cycles is again a p-cycle so we get a subgroup, $\mathsf{Z}_p \subseteq \mathsf{C}_p$. A *$p$-boundary* is a p-chain that is the boundary of a $(p+1)$-chain. The sum of two p-boundaries is again a p-boundary so we get another subgroup, $\mathsf{B}_p \subseteq \mathsf{C}_p$, Taking the boundary twice in a row gives zero for every simplex and thus for every chain, that is, $(\partial_p(\partial_{p+1}d) = 0$. It follows that B_p is a subgroup of Z_p. We can therefore draw the chain complex as in Figure 87.

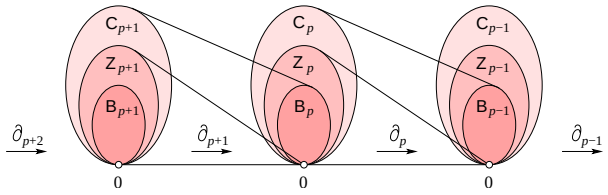

Figure 87: The chain complex consisting of a linear sequence of chain, cycle, and boundary groups connected by homomorphisms.

Homology groups. We would like to talk about cycles but ignore the boundaries since they do not go around a hole. At the same time, we would like to consider two cycles the same if they differ by a boundary. See Figure 88 for a few 1-cycles, some of which are 1-boundaries and some of which are not. This is achieved by taking the quotient of the cycle group and the boundary group. The result is the *p-th homology group*,

$$\mathsf{H}_p \quad = \quad \mathsf{Z}_p/\mathsf{B}_p.$$

Its elements are of the form $[c] = c + \mathsf{B}_p$, where c is a p-cycle. $[c]$ is called a *homology class*, c is a *representative* of $[c]$, and any two cycles in $[c]$ are *homologous* denoted as $c \sim c'$. Note that $[c] = [c']$ whenever $c \sim c'$. Also note that $[c+d] = [c'+d']$ whenever $c \sim c'$ and $d \sim d'$. We use this as a definition of addition for homology classes, so we again have a group. For example, the 1-st homology group of the torus consists of four elements, $[0] = \mathsf{B}_1$, $[\gamma] = \gamma + \mathsf{B}_1$, $[\delta] = \delta + \mathsf{B}_1$, and $[\gamma+\delta] = \gamma + \delta + \mathsf{B}_1$. We often draw the elements as the corners of a cube of some dimension; see Figure 89. If the dimension is β then it has

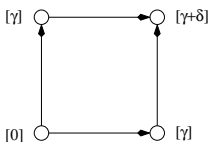

Figure 89: The four homology classes of H_1 are generated by two classes, $[\gamma]$ and $[\delta]$.

2^β corners. The dimension is also the number of classes needed to generate the group, the size of the basis. For the p-th homology group, this number is $\beta_p = \operatorname{rank} \mathsf{H}_p = \log_2 |\mathsf{H}_p|$, the *$p$-th Betti number*. For the torus we have

$$\begin{aligned} \beta_0 &= 1; \\ \beta_1 &= 2; \\ \beta_2 &= 1, \end{aligned}$$

and $\beta_p = 0$ for all $p \neq 0, 1, 2$. Every 0-chain is a 0-cycle. Two 0-cycles are homologous if they are both the sum of an even number or both of an odd number of vertices. Hence $\beta_0 = \log_2 2 = 1$. We have seen the reason for $\beta_1 = 2$ before. Finally, there are only two 2-cycles, namely 0 and the set of all triangles. The latter is not a boundary, hence $\beta_2 = \log_2 2 = 1$.

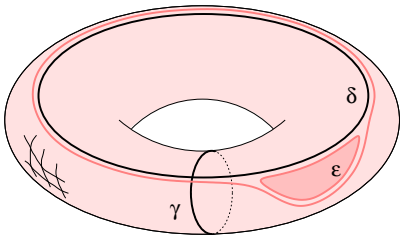

Figure 88: The 1-cycles γ and δ are not 1-boundaries. Adding the 1-boundary ε to δ gives a 1-cycle homologous to δ.

Boundary matrices. To compute homology groups and Betti numbers, we use a matrix representation of the simplicial complex. Specifically, we store the boundary homomorphism for each dimension, setting $\partial_p[i,j] = 1$ if

the i-th $(p-1)$-simplex is in the boundary of the j-th p-simplex, and $\partial_p[i,j] = 0$, otherwise. For example, if the complex consists of all faces of the tetrahedron, then the boundary matrices are

$$\begin{aligned}
\partial_0 &= \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}; \\
\partial_1 &= \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}; \\
\partial_2 &= \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}; \\
\partial_3 &= \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}.
\end{aligned}$$

Given a p-chain as a column vector, $\mathbf{v}$, its boundary is computed by matrix multiplication, $\partial_p \mathbf{v}$. The result is a combination of columns in the p-th boundary matrix, as specified by $\mathbf{v}$. Thus, $\mathbf{v}$ is a p-cycle iff $\partial_p \mathbf{v} = 0$ and $\mathbf{v}$ is a p-boundary iff there is $\mathbf{u}$ such that $\partial_{p+1}\mathbf{u} = \mathbf{v}$.

Matrix reduction. Letting n_p be the number of p-simplices in K, we note that it is also the rank of the p-th chain group, $n_p = \operatorname{rank} \mathsf{C}_p$. The p-th boundary matrix thus has n_{p-1} rows and n_p columns. To figure the sizes of the cycle and boundary groups, and thus of the homology groups, we reduce the matrix to normal form, as shown in Figure 90. The algorithm of choice uses column and

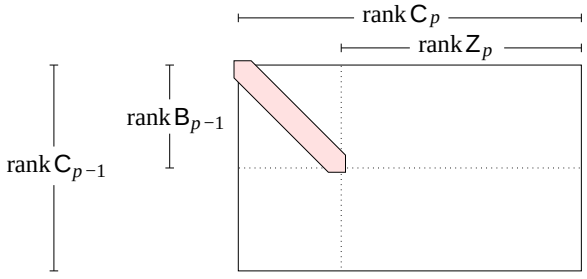

Figure 90: The p-th boundary matrix in normal form. The entries in the shaded portion of the diagonal are 1 and all other entries are 0.

row operations similar to Gaussian elimination for solving a linear system. We write it recursively, calling it with $m = 1$.

```
void REDUCE(m)
  if ∃k, l ≥ m with ∂_p[k, l] = 1 then
    exchange rows m and k and columns m and l;
    for i = m + 1 to n_{p−1} do
      if ∂_p[i, m] = 1 then
        add row m to row i
      endif
    endfor;
    for j = m + 1 to n_p do
      if ∂_p[m, j] = 1 then
        add column m to column j
      endif
    endfor;
    REDUCE(m + 1)
  endif.
```

For each recursive call, we have at most a linear number of row and column operations. The total running time is therefore at most cubic in the number of simplices. Figure 90 shows how we interpret the result. Specifically, the number of zero columns is the rank of the cycle group, Z_p, and the number of 1s in the diagonal is the rank of the boundary group, B_{p-1}. The Betti number is the difference,

$$\beta_p = \operatorname{rank} \mathsf{Z}_p - \operatorname{rank} \mathsf{B}_p,$$

taking the rank of the boundary group from the reduced matrix one dimension up. Working on our example, we get the following reduced matrices.

$$\begin{aligned}
\partial_0 &= \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}; \\
\partial_1 &= \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}; \\
\partial_2 &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}; \\
\partial_3 &= \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}.
\end{aligned}$$

Writing $z_p = \operatorname{rank} \mathsf{Z}_p$ and $b_p = \operatorname{rank} \mathsf{B}_p$, we get $z_0 = 4$ from the zeroth and $b_0 = 3$ from the first reduced boundary matrix. Hence $\beta_0 = z_0 = b_0 = 1$. Furthermore,

$z_1 = 3$ and $b_1 = 3$ giving $\beta_1 = 0$, $z_2 = 1$ and $b_2 = 1$ giving $\beta_2 = 0$, and $z_3 = 0$ giving $\beta_3 = 0$. These are the Betti numbers of the closed ball.

Euler-Poincaré Theorem. The *Euler characteristic* of a simplicial complex is the alternating sum of simplex numbers,

$$\chi = \sum_{p \geq 0} (-1)^p n_p.$$

Recalling that n_p is the rank of the p-th chain group and that it equals the rank of the p-th cycle group plus the rank of the $(p-1)$-st boundary group, we get

$$\begin{aligned} \chi &= \sum_{p \geq 0} (-1)^p (z_p + b_{p-1}) \\ &= \sum_{p \geq 0} (-1)^p (z_p - b_p), \end{aligned}$$

which is the same as the alternating sum of Betti numbers. To appreciate the beauty of this result, we need to know that the Betti numbers do not depend on the triangulation chosen for the space. The proof of this property is technical and omitted. This now implies that the Euler characteristic is an invariant of the space, same as the Betti numbers.

EULER-POINCARÉ THEOREM. $\chi = \sum (-1)^p \beta_p$.

Fifth Homework Assignment

Write the solution to each problem on a single page. The deadline for handing in solutions is November 13.

Problem 1. (20 points). Let $G = (V, E)$ be a maximally connected planar graph and recall that $[k] = \{1, 2, \ldots, k\}$. A *vertex k-coloring* is a mapping $\gamma : V \to [k]$ such that $\gamma(u) \neq \gamma(v)$ whenever $u \neq v$ are adjacent, and an *edge k-coloring* is a mapping $\eta : E \to [k]$ such that $\eta(e) \neq \eta(f)$ whenever $e \neq f$ bound a common triangle. Prove that if G has a vertex 4-coloring then it also has an edge 3-coloring.

Problem 2. ($20 = 10 + 10$ points). Let K be a set of triangles together with their edges and vertices. The vertices are represented by a linear array, as usual, but there is no particular ordering information in the way the edges and triangles are given. In other words, the edges are just a list of index pairs and the triangles are a list of index triplets into the vertex array.

(a) Give an algorithm that decides whether or not K is a triangulation of a 2-manifold.

(b) Analyze your algorithm and collect credit points if the running time of your algorithm is linear in the number of triangles.

Problem 3. ($20 = 5+7+8$ points). Determine the type of 2-manifold with boundary obtained by the following constructions.

(a) Remove a cylinder from a torus in such a way that the rest of the torus remains connected.

(b) Remove a disk from the projective plane.

(c) Remove a Möbius strip from a Klein bottle.

Whenever we remove a piece, we do this like cutting with scissors so that the remainder is still closed, in each case a 2-manifold with boundary.

Problem 4. ($20 = 5 + 5 + 5 + 5$ points). Recall that the sphere is the space of points at unit distance from the origin in three-dimensional Euclidean space, $\mathbb{S}^2 = \{x \in \mathbb{R}^3 \mid \|x\| = 1\}$.

(a) Give a triangulation of $\mathbb{S}^2$.

(b) Give the corresponding boundary matrices.

(c) Reduce the boundary matrices.

(d) Give the Betti numbers of $\mathbb{S}^2$.

Problem 5. ($20 = 10 + 10$ points). The *dunce cap* is obtained by gluing the three edges of a triangular sheet of paper to each other. [After gluing the first two edges you get a cone, with the glued edges forming a seam connecting the cone point with the rim. In the final step, wrap the seam around the rim, gluing all three edges to each other. To imagine how this work, it might help to think of the final result as similar to the shell of a snale.]

(a) Is the dunce cap a 2-manifold? Justify your answer.

(b) Give a triangulation of the dunce cap, making sure that no two edges connect the same two vertices and no two triangles connect the same three vertices.

VI GEOMETRIC ALGORITHMS

20 Plane-Sweep

Plane-sweep is an algorithmic paradigm that emerges in the study of two-dimensional geometric problems. The idea is to sweep the plane with a line and perform the computations in the sequence the data is encountered. In this section, we solve three problems with this paradigm: we construct the convex hull of a set of points, we triangulate the convex hull using the points as vertices, and we test a set of line segments for crossings.

Convex hull. Let S be a finite set of points in the plane, each given by its two coordinates. The *convex hull* of S, denoted by $\mathrm{conv}\, S$, is the smallest convex set that contains S. Figure 91 illustrates the definition for a set of nine points. Imagine the points as solid nails in a planar board. An intuitive construction stretches a rubber band around the nails. After letting go, the nails prevent the complete relaxation of the rubber band which will then trace the boundary of the convex hull.

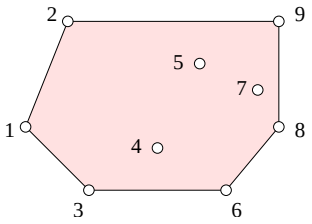

Figure 91: The convex hull of nine points, which we represent by the counterclockwise sequence of boundary vertices: 1, 3, 6, 8, 9, 2.

To construct the counterclockwise cyclic sequence of boundary vertices representing the convex hull, we sweep a vertical line from left to right over the data. At any moment in time, the points in front (to the right) of the line are untouched and the points behind (to the left) of the line have already been processed.

`Step 1.` Sort the points from left to right and relabel them in this sequence as $x_1, x_2, \ldots, x_n$.

`Step 2.` Construct a counterclockwise triangle from the first three points: $x_1x_2x_3$ or $x_1x_3x_2$.

`Step 3.` For i from 4 to n, add the next point x_i to the convex hull of the preceding points by finding the two lines that pass through x_i and support the convex hull.

The algorithm is illustrated in Figure 92, which shows the addition of the sixth point in the data set.

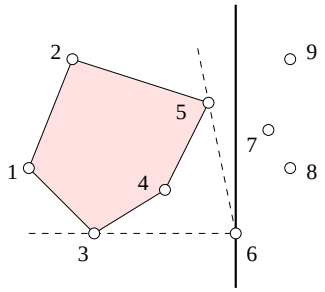

Figure 92: The vertical sweep-line passes through point 6. To add 6, we substitute 6 for the sequence of vertices on the boundary between 3 and 5.

Orientation test. A critical test needed to construct the convex hull is to determine the orientation of a sequence of three points. In other words, we need to be able to distinguish whether we make a left-turn or a right-turn as we go from the first to the middle and then the last point in the sequence. A convenient way to determine the orientation evaluates the determinant of a three-by-three matrix. More precisely, the points $a = (a_1, a_2)$, $b = (b_1, b_2)$, $c = (c_1, c_2)$ form a left-turn iff

$$\det \begin{bmatrix} 1 & a_1 & a_2 \\ 1 & b_1 & b_2 \\ 1 & c_1 & c_2 \end{bmatrix} > 0.$$

The three points form a right-turn iff the determinant is negative and they lie on a common line iff the determinant is zero.

```
boolean LEFT(Points a, b, c)
  return [a1(b2 − c2) + b1(c2 − a2)
                      + c1(a2 − b2) > 0].
```

To see that this formula is correct, we may convince ourselves that it is correct for three non-collinear points, e.g. $a = (0, 0)$, $b = (1, 0)$, and $c = (0, 1)$. Remember also that the determinant measures the area of the triangle and is therefore a continuous function that passes through zero only when the three points are collinear. Since we can continuously move every left-turn to every other left-turn without leaving the class of left-turns, it follows that the sign of the determinant is the same for all of them.

Finding support lines. We use a doubly-linked cyclic list of vertices to represent the convex hull boundary. Each

node in the list contains pointers to the next and the previous nodes. In addition, we have a pointer $last$ to the last vertex added to the list. This vertex is also the rightmost in the list. We add the i-th point by connecting it to the vertices $\mu \to pt$ and $\lambda \to pt$ identified in a counterclockwise and a clockwise traversal of the cycle starting at $last$, as illustrated in Figure 93. We simplify notation by using

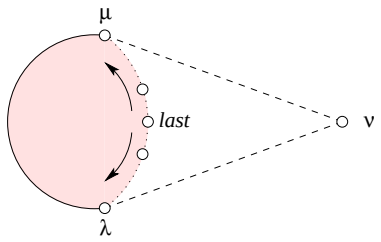

Figure 93: The upper support line passes through the first point $\mu \to pt$ that forms a left-turn from $\nu \to pt$ to $\mu \to next \to pt$.

nodes in the parameter list of the orientation test instead of the points they store.

$\mu = \lambda = last$; create new node with $\nu \to pt = i$;
`while` RIGHT$(\nu, \mu, \mu \to next)$ `do`
  $\mu = \mu \to next$
`endwhile`;
`while` LEFT$(\nu, \lambda, \lambda \to prev)$ `do`
  $\lambda = \lambda \to prev$
`endwhile`;
$\nu \to next = \mu$; $\nu \to prev = \lambda$;
$\mu \to prev = \lambda \to next = \nu$; $last = \nu$.

The effort to add the i-th point can be large, but if it is then we remove many previously added vertices from the list. Indeed, each iteration of the for-loop adds only one vertex to the cyclic list. We charge \$2 for the addition, one dollar for the cost of adding and the other to pay for the future deletion, if any. The extra dollars pay for all iterations of the while-loops, except for the first and the last. This implies that we spend only constant amortized time per point. After sorting the points from left to right, we can therefore construct the convex hull of n points in time O(n).

Triangulation. The same plane-sweep algorithm can be used to decompose the convex hull into triangles. All we need to change is that points and edges are never removed and a new point is connected to every point examined during the two while-loops. We define a *(geometric) triangulation* of a finite set of points S in the plane as a maximally connected straight-line embedding of a planar graph whose vertices are mapped to points in S. Figure 94 shows the triangulation of the nine points in Figure 91 constructed by the plane-sweep algorithm. A triangulation is

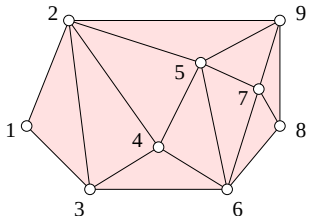

Figure 94: Triangulation constructed with the plane-sweep algorithm.

not necessarily a maximally connected planar graph since the prescribed placement of the points fixes the boundary of the outer face to be the boundary of the convex hull. Letting k be the number of edges of that boundary, we would have to add $k - 3$ more edges to get a maximally connected planar graph. It follows that the triangulation has $m = 3n - (k + 3)$ edges and $\ell = 2n - (k + 2)$ triangles.

Line segment intersection. As a third application of the plane-sweep paradigm, we consider the problem of deciding whether or not n given line segments have pairwise disjoint interiors. We allow line segments to share endpoints but we do not allow them to cross or to overlap. We may interpret this problem as deciding whether or not a straight-line drawing of a graph is an embedding. To simplify the description of the algorithm, we assume no three endpoints are collinear, so we only have to worry about crossings and not about other overlaps.

How can we decide whether or not a line segment with endpoint $u = (u_1, u_2)$ and $v = (v_1, v_2)$ crosses another line segment with endpoints $p = (p_1, p_2)$ and $q = (q_1, q_2)$? Figure 95 illustrates the question by showing the four different cases of how two line segments and the lines they span can intersect. The line segments cross iff uv intersects the line of pq and pq intersects the line of uv. This condition can be checked using the orientation test.

`boolean` CROSS(`Points` u, v, p, q)
  `return` [(LEFT(u, v, p) `xor` LEFT(u, v, q)) `and`
    (LEFT(p, q, u) `xor` LEFT(p, q, v))].

We can use the above function to test all $\binom{n}{2}$ pairs of line segments, which takes time O(n^2).

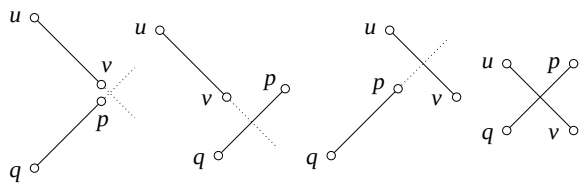

Figure 95: Three pairs of non-crossing and one pair of crossing line segments.

Plane-sweep algorithm. We obtain a faster algorithm by sweeping the plane with a vertical line from left to right, as before. To avoid special cases, we assume that no two endpoints are the same or lie on a common vertical line. During the sweep, we maintain the subset of line segments that intersect the sweep-line in the order they meet the line, as shown in Figure 96. We store this subset

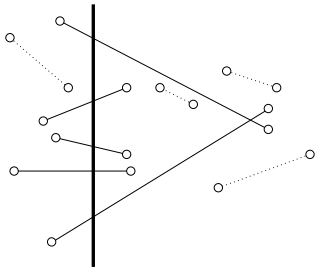

Figure 96: Five of the line segments intersect the sweep-line at its current position and two of them cross.

in a dictionary, which is updated at every endpoint. Only line segments that are adjacent in the ordering along the sweep-line are tested for crossings. Indeed, two line segments that cross are adjacent right before the sweep-line passes through the crossing, if not earlier.

`Step 1.` Sort the $2n$ endpoints from left to right and relabel them in this sequence as $x_1, x_2, \ldots, x_{2n}$. Each point still remembers the index of the other endpoint of its line segment.

`Step 2.` For i from 1 to $2n$, process the i-th endpoint as follows:

`Case 2.1` x_i is left endpoint of the line segment x_ix_j. Therefore, $i < j$. Insert x_ix_j into the dictionary and let uv and pq be its predecessor and successor. If $\textsc{Cross}(u, v, x_i, x_j)$ or $\textsc{Cross}(p, q, x_i, x_j)$ then report the crossing and stop.

`Case 2.2` x_i is right endpoint of the line segment x_ix_j. Therefore, $i > j$. Let uv and pq be the predecessor and the successor of x_ix_j. If $\textsc{Cross}(u, v, p, q)$ then report the crossing and stop. Delete x_ix_j from the dictionary.

We do an insertion into the dictionary for each left endpoint and a deletion from the dictionary for each right endpoint, both in time $O(\log n)$. In addition, we do at most two crossing tests per endpoint, which takes constant time. In total, the algorithm takes time $O(n \log n)$ to test whether a set of n line segments contains two that cross.

21 Delaunay Triangulations

The triangulations constructing by plane-sweep are typically of inferior quality, that is, there are many long and skinny triangles and therefore many small and many large angles. We study Delaunay triangulations which distinguish themselves from all other triangulations by a number of nice properties, including they have fast algorithms and they avoid small angles to the extent possible.

Plane-sweep versus Delaunay triangulation. Figures 97 and 98 show two triangulations of the same set of points, one constructed by plane-sweep and the other the Delaunay triangulation. The angles in the Delaunay triangulation seem consistently larger than those in the plane-sweep triangulation. This is not a coincidence and it can be proved that the Delaunay triangulation maximizes the minimum angle for every input set. Both triangulations contain the edges that bound the convex hull of the input set.

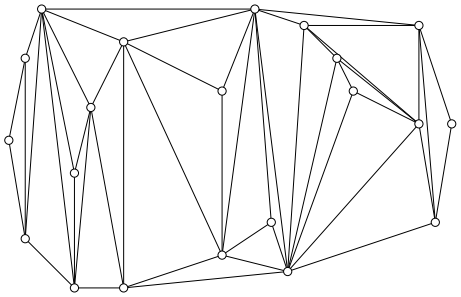

Figure 97: Triangulation constructed by plane-sweep. Points on the same vertical line are processed from bottom to top.

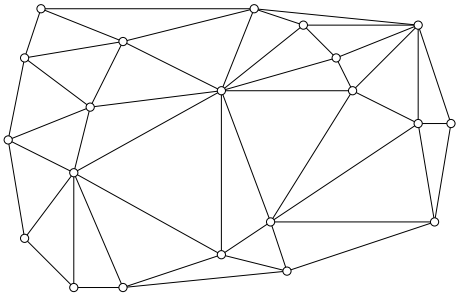

Figure 98: Delaunay triangulation of the same twenty-one points triangulated in Figure 97.

Voronoi diagram. We introduce the Delaunay triangulation indirectly, by first defining a particular decomposition of the plane into regions, one per point in the finite data set S. The region of the point u in S contains all points x in the plane that are at least as close to u as to any other point in S, that is,

$$V_u \quad = \quad \{x \in \mathbb{R}^2 \mid \|x-u\| \leq \|x-v\|, v \in S\},$$

where $\|x-u\| = [(x_1-u_1)^2 + (x_2-u_2)^2]^{1/2}$ is the Euclidean distance between the points x and u. We refer to V_u as the *Voronoi region* of u. It is closed and its boundary consists of *Voronoi edges* which V_u shares with neighboring Voronoi regions. A Voronoi edge ends in *Voronoi vertices* which it shares with other Voronoi edges. The *Voronoi diagram* of S is the collection of Voronoi regions, edges and vertices. Figure 99 illustrates the definitions. Let n be the number of points in S. We list some of the properties that will be important later.

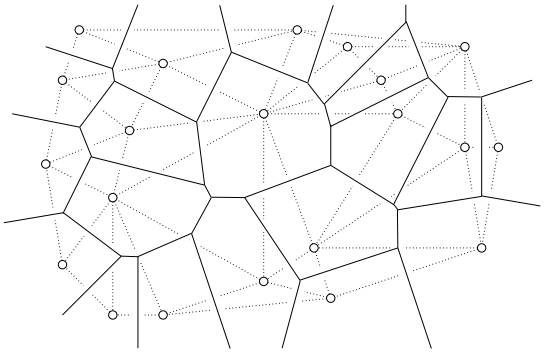

Figure 99: The (solid) Voronoi diagram drawn above the (dotted) Delaunay triangulation of the same twenty-one points triangulated in Figures 97 and 98. Some of the Voronoi edges are too far out to fit into the picture.

- Each Voronoi region is a convex polygon constructed as the intersection of $n-1$ closed half-planes.
- The Voronoi region V_u is bounded (finite) iff u lies in the interior of the convex hull of S.
- The Voronoi regions have pairwise disjoint interiors and together cover the entire plane.

Delaunay triangulation. We define the *Delaunay triangulation* as the straight-line dual of the Voronoi diagram. Specifically, for every pair of Voronoi regions V_u and V_v that share an edge, we draw the line segment from u to v. By construction, every Voronoi vertex, x, has $j \geq 3$ closest input points. Usually there are exactly three closest

points, u, v, w, in which case the triangle they span belongs to the Delaunay triangulation. Note that x is equally far from u, v, and w and further from all other points in S. This implies the *empty circle property* of Delaunay triangles: all points of $S - \{u, v, w\}$ lie outside the circumscribed circle of uvw. Similarly, for each Delaunay edge uv, there is a circle that passes through u and v such that all points of $S - \{u, v\}$ lie outside the circle. For example, the circle centered at the midpoint of the Voronoi edge shared by V_u and V_v is empty in this sense. This property can be used to prove that the edge skeleton of the Delaunay triangulation is a straight-line embedding of a planar graph.

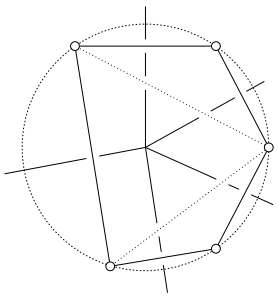

Figure 100: A Voronoi vertex of degree 5 and the corresponding pentagon in the Delaunay triangulation. The dotted edges complete the triangulation by decomposing the pentagon into three triangles.

Now suppose there is a vertex with degree $j > 3$. It corresponds to a polygon with $j > 3$ edges in the Delaunay triangulation, as illustrated in Figure 100. Strictly speaking, the Delaunay triangulation is no longer a triangulation but we can complete it to a triangulation by decomposing each j-gon into $j - 2$ triangles. This corresponds to perturbing the data points every so slightly such that the degree-j Voronoi vertices are resolved into trees in which $j - 2$ degree-3 vertices are connected by $j - 3$ tiny edges.

Local Delaunayhood. Given a triangulation of a finite point set S, we can test whether or not it is the Delaunay triangulation by testing each edge against the two triangles that share the edge. Suppose the edge uv in the triangulation T is shared by the triangles uvp and uvq. We call uv *locally Delaunay*, or *lD* for short, if q lies on or outside the circle that passes through u, v, p. The condition is symmetric in p and q because the circle that passes through u, v, p intersects the first circle in points u and v. It follows that p lies on or outside the circle of u, v, q iff q lies on or outside the circle of u, v, p. We also call uv locally Delaunay if it bounds the convex hull of S and thus belongs to only one triangle. The local condition on the edges implies a global property.

DELAUNAY LEMMA. If every edge in a triangulation K of S is locally Delaunay then K is the Delaunay triangulation of S.

Although every edge of the Delaunay triangulation is locally Delaunay, the Delaunay Lemma is not trivial. Indeed, K may contain edges that are locally Delaunay but do not belong to the Delaunay triangulation, as shown in Figure 101. We omit the proof of the lemma.

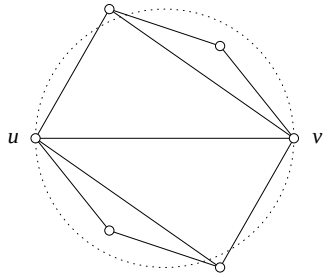

Figure 101: The edge uv is locally Delaunay but does not belong to the Delaunay triangulation.

Edge-flipping. The Delaunay Lemma suggests we construct the Delaunay triangulation by first constructing an arbitrary triangulation of the point set S and then modifying it locally to make all edges lD. The idea is to look for non-lD edges and to flip them, as illustrated in Figure 102. Indeed, if uv is a non-lD edge shared by triangles uvp and

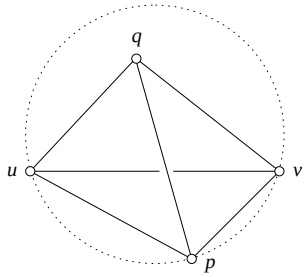

Figure 102: The edge uv is non-lD and can be flipped to the edge pq, which is lD.

uvq then $upvq$ is a convex quadrilateral and *flipping* uv means substituting one diagonal for the other, namely pq

for uv. Note that if uv is non-lD then pq is lD. It is important that the algorithm finds non-lD edges quickly. For this purpose, we use a stack of edges. Initially, we push all edges on the stack and mark them.

```
while stack is non-empty do
  pop edge uv from stack and unmark it;
  if uv is non-lD then
    substitute pq for uv;
    for ab ∈ {up, pv, vq, qu} do
      if ab is unmarked then
        push ab on the stack and mark it
      endif
    endfor
  endif
endwhile.
```

The marks avoid multiple copies of the same edge on the stack. This implies that at any one moment the size of the stack is less than $3n$. Note also that initially the stack contains all non-lD edges and that this property is maintained as an invariant of the algorithm. The Delaunay Lemma implies that when the algorithm halts, which is when the stack is empty, then the triangulation is the Delaunay triangulation. However, it is not yet clear that the algorithm terminates. Indeed, the stack can grow and shrink during the course of the algorithm, which makes it difficult to prove that it ever runs empty.

In-circle test. Before studying the termination of the algorithm, we look into the question of distinguishing lD from non-lD edges. As before we assume that the edge uv is shared by the triangles uvp and uvq in the current triangulation. Recall that uv is lD iff q lies outside the circle that passes through u, v, p. Let $f : \mathbb{R}^2 \to \mathbb{R}$ be defined by $f(x) = x_1^2 + x_2^2$. As illustrated in Figure 103, the graph of this function is a paraboloid in three-dimensional space and we write $x^+ = (x_1, x_2, f(x))$ for the vertical projection of the point x onto the paraboloid. Assuming the three points u, v, p do not lie on a common line then the points u^+, v^+, p^+ lie on a non-vertical plane that is the graph of a function $h(x) = \alpha x_1 + \beta x_2 + \gamma$. The projection of the intersection of the paraboloid and the plane back into $\mathbb{R}^2$ is given by

$$\begin{aligned} 0 &= f(x) - h(x) \\ &= x_1^2 + x_2^2 - \alpha x_1 - \beta x_2 - \gamma, \end{aligned}$$

which is the equation of a circle. This circle passes through u, v, p so it is the circle we have to compare q against. We note that q lies inside the circle iff q^+ lies below the plane. The latter test can be based on the sign of the determinant of the 4-by-4 matrix

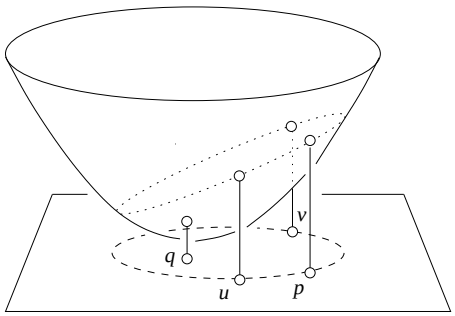

Figure 103: The plane passing through u^+, v^+, p^+ intersects the paraboloid in an ellipse whose projection into $\mathbb{R}^2$ passes through the points u, v, p. The point q^+ lies below the plane iff q lies inside the circle.

$$\Delta = \begin{bmatrix} 1 & u_1 & u_2 & u_1^2 + u_2^2 \\ 1 & v_1 & v_2 & v_1^2 + v_2^2 \\ 1 & p_1 & p_2 & p_1^2 + p_2^2 \\ 1 & q_1 & q_2 & q_1^2 + q_2^2 \end{bmatrix}.$$

Exchanging two rows in the matrix changes the sign. While the in-circle test should be insensitive to the order of the first three points, the sign of the determinant is not. We correct the change using the sign of the determinant of the 3-by-3 matrix that keeps track of the ordering of u, v, p along the circle,

$$\Gamma = \begin{bmatrix} 1 & u_1 & u_2 \\ 1 & v_1 & v_2 \\ 1 & p_1 & p_2 \end{bmatrix}.$$

Now we claim that s is inside the circle of u, v, p iff the two determinants have opposite signs:

```
boolean INCIRCLE(Points u, v, p, q)
  return det Γ · det Δ < 0.
```

We first show that the boolean function is correct for $u = (0, 0)$, $v = (1, 0)$, $p = (0, 1)$, and $q = (0, 0.5)$. The sign of the product of determinants remains unchanged if we continuously move the points and avoid the configurations that make either determinant zero, which are when u, v, p are collinear and when u, v, p, q are cocircular. We can change any configuration where q is inside the circle of u, v, p continuously into the special configuration without going through zero, which implies the correctness of the function for general input points.

Termination and running time. To prove the edge-flip algorithm terminates, we imagine the triangulation lifted to $\mathbb{R}^3$. We do this by projecting the vertices vertically onto the paraboloid, as before, and connecting them with straight edges and triangles in space. Let uv be an edge shared by triangles uvp and uvq that is flipped to pq by the algorithm. It follows the line segments uv and pq cross and their endpoints form a convex quadrilateral, as shown in Figure 104. After lifting the two line segments, we get

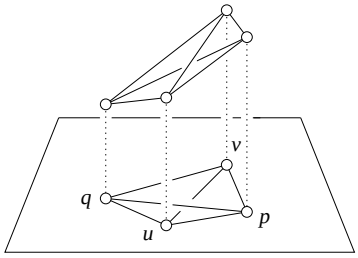

Figure 104: A flip in the plane lifts to a tetrahedron in space in which the lD edge passes below the non-lD edge.

u^+v^+ passing above p^+q^+. We may thus think of the flip as gluing the tetrahedron $u^+v^+p^+q^+$ underneath the surface obtained by lifting the triangulation. The surface is pushed down by each flip and never pushed back up. The removed edge is now above the new surface and can therefore not be reintroduced by a later flip. It follows that the algorithm performs at most $\binom{n}{2}$ flips and thus takes at most time O(n^2) to construct the Delaunay triangulation of S. There are faster algorithms that work in time O($n \log n$) but we prefer the suboptimal method because it is simpler and it reveals more about Delaunay triangulations than the other algorithms.

The lifting of the input points to $\mathbb{R}^3$ leads to an interesting interpretation of the edge-flip algorithm. Starting with a monotone triangulated surface passing through the lifted points, we glue tetrahedra below the surface until we reach the unique convex surface that passes through the points. The projection of this convex surface is the Delaunay triangulation of the points in the plane. This also gives a reinterpretation of the Delaunay Lemma in terms of convex and concave edges of the surface.

22 Alpha Shapes

Many practical applications of geometry have to do with the intuitive but vague concept of the shape of a finite point set. To make this idea concrete, we use the distances between the points to identify subcomplexes of the Delaunay triangulation that represent that shape at different levels of resolution.

Union of disks. Let S be a set of n points in $\mathbb{R}^2$. For each $r \geq 0$, we write $B_u(r) = \{x \in \mathbb{R}^2 \mid \|x - u\| \leq r\}$ for the closed disk with center u and radius r. Let $\mathbb{U}(r) = \bigcup_{u \in S} B_u(r)$ be the union of the n disks. We decompose this union into convex sets of the form $R_u(r) = B_u(r) \cap V_u$. Then

(i) $R_u(r)$ is closed and convex for every point $u \in S$ and every radius $r \geq 0$;

(ii) $R_u(r)$ and $R_v(r)$ have disjoint interiors whenever the two points, u and v, are different;

(iii) $\mathbb{U}(r) = \bigcup_{u \in S} R_u(r)$.

We illustrate this decomposition in Figure 105. Each region $R_u(r)$ is the intersection of $n - 1$ closed half-planes and a closed disk. All these sets are closed and convex, which implies (i). The Voronoi regions have disjoint interiors, which implies (ii). Finally, take a point $x \in \mathbb{U}(r)$ and let u be a point in S with $x \in V_u$. Then $x \in B_u(r)$ and therefore $x \in R_u(x)$. This implies (iii).

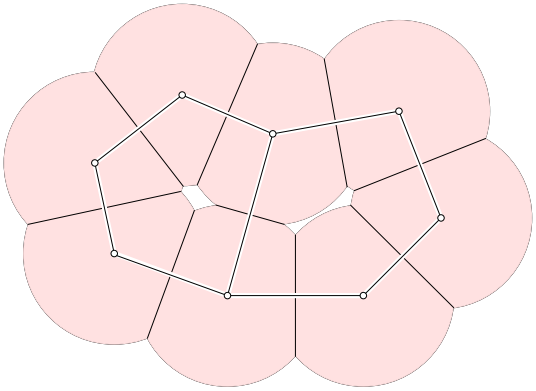

Figure 105: The Voronoi decomposition of a union of eight disks in the plane and superimposed dual alpha complex.

Nerve. Similar to defining the Delaunay triangulation as the dual of the Voronoi diagram, we define the alpha complex as the dual of the Voronoi decomposition of the union of disks. This time around, we do this more formally. Letting C be a finite collection of sets, the *nerve* of C is the system of subcollections that have a non-empty common intersection,

$$\mathrm{Nrv}\, C \quad = \quad \{X \subseteq C \mid \bigcap X \neq \emptyset\}.$$

This is an abstract simplicial complex since $\bigcap X \neq \emptyset$ and $Y \subseteq X$ implies $\bigcap Y \neq \emptyset$. For example, if C is the collection of Voronoi regions then $\mathrm{Nrv}\, C$ is an abstract version of the Delaunay triangulation. More specifically, this is true provide the points are in general position and in particular no four points lie on a common circle. We will assume this for the remainder of this section. We say the Delaunay triangulation is a *geometric realization* of $\mathrm{Nrv}\, C$, namely the one obtained by mapping each Voronoi region (a vertex in the abstract simplicial complex) to the generating point. All edges and triangles are just convex hulls of their incident vertices. To go from the Delaunay triangulation to the alpha complex, we substitute the regions $R_u(r)$ for the V_u. Specifically,

$$\mathrm{Alpha}(r) \quad = \quad \mathrm{Nrv}\, \{R_u(r) \mid u \in S\}.$$

Clearly, this is isomorphic to a subcomplex of the nerve of Voronoi regions. We can therefore draw $\mathrm{Alpha}(r)$ as a subcomplex of the Delaunay triangulation; see Figure 105. We call this geometric realization of $\mathrm{Alpha}(r)$ the *alpha complex* for radius r, denoted as $A(r)$. The *alpha shape* for the same radius is the underlying space of the alpha complex, $|A(r)|$.

The nerve preserves the way the union is connected. In particular, their Betti numbers are the same, that is, $\beta_p(\mathbb{U}(r)) = \beta_p(A(r))$ for all dimensions p and all radii r. This implies that the union and the alpha shape have the same number of components and the same number of holes. For example, in Figure 105 both have one component and two holes. We omit the proof of this property.

Filtration. We are interested in the sequence of alpha shapes as the radius grows from zero to infinity. Since growing r grows the regions $R_u(r)$, the nerve can only get bigger. In other words, $A(r) \subseteq A(s)$ whenever $r \leq s$. There are only finitely many subcomplexes of the Delaunay triangulation. Hence, we get a finite sequence of alpha complexes. Writing A_i for the i-th alpha complex, we get the following nested sequence,

$$S = A_1 \subset A_2 \subset \ldots \subset A_k = D,$$

where D denotes the Delaunay triangulation of S. We call such a sequence of complexes a *filtration*. We illustrate this construction in Figure 106. The sequence of al-

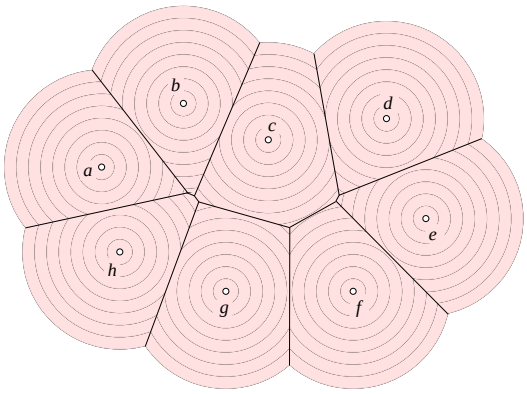

Figure 106: A finite sequence of unions of disks, all decomposed by the same Voronoi diagram.

pha complexes begins with a set of n isolated vertices, the points in S. To go from one complex to the next, we either add an edge, we add a triangle, or we add a pair consisting of a triangle with one of its edges. In Figure 106, we begin with eight vertices and get the following sequence of alpha complexes.

$$\begin{aligned}
A_1 &= \{a, b, c, d, e, f, g, h\};\\
A_2 &= A_1 \cup \{ah\};\\
A_3 &= A_2 \cup \{bc\};\\
A_4 &= A_3 \cup \{ab, ef\};\\
A_5 &= A_4 \cup \{de\};\\
A_6 &= A_5 \cup \{gh\};\\
A_7 &= A_6 \cup \{cd\};\\
A_8 &= A_7 \cup \{fg\};\\
A_9 &= A_8 \cup \{cg\}.
\end{aligned}$$

Going from A_7 to A_8, we get for the first time a 1-cycle, which bounds a hole in the embedding. In A_9, this hole is cut into two. This is the alpha complex depicted in Figure 105. We continue.

$$\begin{aligned}
A_{10} &= A_9 \cup \{cf\};\\
A_{11} &= A_{10} \cup \{abh, bh\};\\
A_{12} &= A_{11} \cup \{cde, ce\};\\
A_{13} &= A_{12} \cup \{cfg\};\\
A_{14} &= A_{13} \cup \{cef\};\\
A_{15} &= A_{14} \cup \{bch, ch\};\\
A_{16} &= A_{15} \cup \{cgh\}.
\end{aligned}$$

At this moment, we have a triangulated disk but not yet the entire Delaunay triangulation since the triangle bcd and the edge bd are still missing. Each step is generic except when we add two equally long edges to A_3.

Compatible ordering of simplices. We can represent the entire filtration of alpha complexes compactly by sorting the simplices in the order they join the growing complex. An ordering $\sigma_1, \sigma_2, \ldots, \sigma_m$ of the Delaunay simplices is *compatible* with the filtration if

1. the simplices in A_i precede the ones not in A_i for each i;
2. the faces of a simplex precede the simplex.

For example, the sequence

$$\begin{aligned}
&a, b, c, d, e, f, g, h; ah; bc; ab, ef;\\
&de; gh; cd; fg; cg; cf; bh, abh; ce,\\
&cde; cfg; cef; ch, bch; cgh; bd; bcd
\end{aligned}$$

is compatible with the filtration in Figure 106. Every alpha complex is a prefix of the compatible sequence but not necessarily the other way round. Condition 2 guarantees that every prefix is a complex, whether an alpha complex or not. We thus get a finer filtration of complexes

$$\emptyset = K_0 \subset K_1 \subset \ldots \subset K_m = D,$$

where K_i is the set of simplices from σ_1 to σ_i. To construct the compatible ordering, we just need to compute for each Delaunay simplex the radius $r_i = r(\sigma_i)$ such that $\sigma_i \in A(r)$ iff $r \geq r_i$. For a vertex, this radius is zero. For a triangle, this is the radius of the circumcircle. For

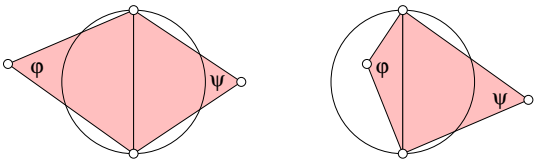

Figure 107: Left: the middle edge belongs to two acute triangles. Right: it belongs to an obtuse and an acute triangle.

an edge, we have two cases. Let φ and ψ be the angles opposite the edge σ_i inside the two incident triangles. We have $\varphi + \psi > 180°$ because of the empty circle property.

CASE 1. $\varphi < 90°$ and $\psi < 90°$. Then $r_i = r(\sigma_i)$ is half the length of the edge.

CASE 2. $\varphi \geq 90^\circ$. Then $r_i = r_j$, where σ_j is the incident triangle with angle φ.

Both cases are illustrated in Figure 107. In Case 2, the edge σ_i enters the growing alpha complex together with the triangle σ_j. The total number of simplices in the Delaunay triangulation is $m < 6n$. The threshold radii can be computed in time O(n). Sorting the simplices into the compatible ordering can therefore be done in time O($n \log n$).

Betti numbers. In two dimensions, Betti numbers can be computed directly, without resorting to boundary matrices. The only two possibly non-zero Betti numbers are β_0, the number of components, and β_1, the number of holes. We compute the Betti numbers of K_j by adding the simplices in order.

```
β0 = β1 = 0;
for i = 1 to j do
  switch dim σi:
    case 0: β0 = β0 + 1;
    case 1: let u, v be the endpoints of σi;
      if FIND(u) = FIND(v) then β1 = β1 + 1
                           else β0 = β0 − 1;
                                UNION(u, v)
      endif
    case 2: β1 = β1 − 1
  endswitch
endfor.
```

All we need is tell apart the two cases when σ_i is an edge. This is done using a union-find data structure maintaining the components of the alpha complex in amortized time $\alpha(n)$ per simplex. The total running time of the algorithm for computing Betti numbers is therefore O($n\alpha(n)$).

Sixth Homework Assignment

Write the solution to each problem on a single page. The deadline for handing in solutions is November 25.

Problem 1. (20 points). Let S be a set of n unit disks in the Euclidean plane, each given by its center and radius, which is one. Give an algorithm that decides whether any two of the disks in S intersect.

Problem 2. ($20 = 10 + 10$ points). Let S be a set of n points in the Euclidean plane. The *Gabriel graph* connects points $u, v \in S$ with a straight edge if

$$\|u - v\|^2 \quad \leq \quad \|u - p\|^2 + \|v - p\|^2$$

for every point p in S.

(a) Show that the Grabriel graph is a subgraph of the edge skeleton of the Delaunay triangulation.

(b) Is the Gabriel graph necessarily connected? Justify your answer.

Problem 3. ($20 = 10 + 10$ points). Consider a set of $n \geq 3$ closed disks in the Euclidean plane. The disks are allowed to touch but no two of them have an interior point in common.

(a) Show that the number of touching pairs of disks is at most $3n - 6$.

(b) Give a construction that achieves the upper bound in (a) for any $n \geq 3$.

Problem 4. ($20 = 10 + 10$ points). Let K be a triangulation of a set of $n \geq 3$ points in the plane. Let L be a line that avoids all the points.

(a) Prove that L intersects at most $2n - 4$ of the edges in K.

(b) Give a construction for which L achieves the upper bound in (a) for any $n \geq 3$.

Problem 5. (20 points). Let S be a set of n points in the Euclidean plane, consider its Delaunay triangulation and the corresponding filtration of alpha complexes,

$$S = A_1 \subset A_2 \subset \ldots \subset A_k.$$

Under what conditions is it true that A_i and A_{i+1} differ by a single simplex for every $1 \leq i \leq m - 1$?

VII NP-COMPLETENESS

23 Easy and Hard Problems

The theory of NP-completeness is an attempt to draw a line between tractable and intractable problems. The most important question is whether there is indeed a difference between the two, and this question is still unanswered. Typical results are therefore relative statements such as "if problem B has a polynomial-time algorithm then so does problem C" and its equivalent contra-positive "if problem C has no polynomial-time algorithm then neither has problem B". The second formulation suggests we remember hard problems C and for a new problem B we first see whether we can prove the implication. If we can then we may not want to even try to solve problem B efficiently. A good deal of formalism is necessary for a proper description of results of this kind, of which we will introduce only a modest amount.

What is a problem? An *abstract decision problem* is a function $I \to \{0,1\}$, where I is the set of problem instances and 0 and 1 are interpreted to mean FALSE and TRUE, as usual. To completely formalize the notion, we encode the problem instances in strings of zeros and ones: $I \to \{0,1\}^*$. A *concrete decision problem* is then a function $Q : \{0,1\}^* \to \{0,1\}$. Following the usual convention, we map bit-strings that do not correspond to meaningful problem instances to 0.

As an example consider the shortest-path problem. A problem instance is a graph and a pair of vertices, u and v, in the graph. A solution is a shortest path from u and v, or the length of such a path. The decision problem version specifies an integer k and asks whether or not there exists a path from u to v whose length is at most k. The theory of NP-completeness really only deals with decision problems. Although this is a loss of generality, the loss is not dramatic. For example, given an algorithm for the decision version of the shortest-path problem, we can determine the length of the shortest path by repeated decisions for different values of k. Decision problems are always easier (or at least not harder) than the corresponding optimization problems. So in order to prove that an optimization problem is hard it suffices to prove that the corresponding decision problem is hard.

Polynomial time. An algorithm *solves* a concrete decision problem Q *in time* $T(n)$ if for every instance $x \in \{0,1\}^*$ of length n the algorithm produces $Q(x)$ in time at most $T(n)$. Note that this is the worst-case notion of time-complexity. The problem Q is *polynomial-time solvable* if $T(n) = \mathrm{O}(n^k)$ for some constant k independent of n. The first important complexity class of problems is

$$\mathsf{P} = \text{set of concrete decision problems that are polynomial-time solvable.}$$

The problems $Q \in \mathsf{P}$ are called *tractable* or *easy* and the problems $Q \notin \mathsf{P}$ are called *intractable* or *hard*. Algorithms that take only polynomial time are called *efficient* and algorithms that require more than polynomial time are *inefficient*. In other words, until now in this course we only talked about efficient algorithms and about easy problems. This terminology is adapted because the rather fine grained classification of algorithms by complexity we practiced until now is not very useful in gaining insights into the rather coarse distinction between polynomial and non-polynomial.

It is convenient to recast the scenario in a formal language framework. A *language* is a set $L \subseteq \{0,1\}^*$. We can think of it as the set of problem instances, x, that have an affirmative answer, $Q(x) = 1$. An algorithm $A : \{0,1\}^* \to \{0,1\}$ *accepts* $x \in \{0,1\}^*$ if $A(x) = 1$ and it *rejects* x if $A(x) = 0$. The language *accepted* by A is the set of strings $x \in \{0,1\}^*$ with $A(x) = 1$. There is a subtle difference between accepting and *deciding* a language L. The latter means that A accepts every $x \in L$ and rejects every $x \notin L$. For example, there is an algorithm that accepts every program that halts, but there is no algorithm that decides the language of such programs. Within the formal language framework we redefine the class of polynomial-time solvable problems as

$$\begin{aligned}\mathsf{P} &= \{L \subseteq \{0,1\}^* \mid L \text{ is accepted by a polynomial-time algorithm}\} \\ &= \{L \subseteq \{0,1\}^* \mid L \text{ is decided by a polynomial-time algorithm}\}.\end{aligned}$$

Indeed, a language that can be accepted in polynomial time can also be decided in polynomial time: we keep track of the time and if too much goes by without x being accepted, we turn around and reject x. This is a non-constructive argument since we may not know the constants in the polynomial. However, we know such constants exist which suffices to show that a simulation as sketched exists.

Hamiltonian cycles. We use a specific graph problem to introduce the notion of verifying a solution to a problem, as opposed to solving it. Let $G = (V, E)$ be an undirected graph. A *hamiltonian cycle* contains every vertex

$v \in V$ exactly once. The graph G is *hamiltonian* if it has a hamiltonian cycle. Figure 108 shows a hamiltonian cycle of the edge graph of a Platonic solid. How about the edge graphs of the other four Platonic solids? Define $L =$

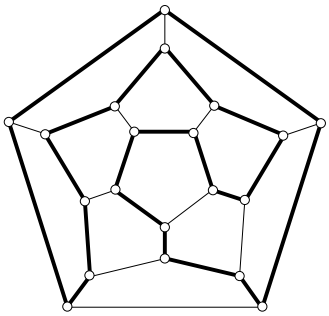

Figure 108: The edge graph of the dodecahedron and one of its hamiltonian cycles.

$\{G \mid G \text{ is hamiltonian}\}$. We can thus ask whether or not $L \in \mathsf{P}$, that is, whether or not there is a polynomial-time algorithm that decides whether or not a graph is hamiltonian. The answer to this question is currently not known, but there is evidence that the answer might be negative. On the other hand, suppose y is a hamiltonian cycle of G. The language $L' = \{(G, y) \mid y \text{ is a hamiltonian cycle of } G\}$ is certainly in P because we just need to make sure that y and G have the same number of vertices and every edge of y is also an edge of G.

Non-deterministic polynomial time. More generally, it seems easier to verify a given solution than to come up with one. In a nutshell, this is what NP-completeness is about, namely finding out whether this is indeed the case and whether the difference between accepting and verifying can be used to separate hard from easy problems.

Call $y \in \{0,1\}^*$ a *certificate*. An algorithm A *verifies* a problem instance $x \in \{0,1\}^*$ if there exists a certificate y with $A(x, y) = 1$. The language *verified* by A is the set of strings $x \in \{0,1\}^*$ verified by A. We now define a new class of problems,

$$\mathsf{NP} \quad = \quad \{L \subseteq \{0,1\}^* \mid L \text{ is verified by a polynomial-time algorithm}\}.$$

More formally, L is in NP if for every problem instance $x \in L$ there is a certificate y whose length is bounded from above by a polynomial in the length of x such that $A(x, y) = 1$ and A runs in polynomial time. For example, deciding whether or not G is hamiltonian is in NP.

The name NP is an abbreviation for **n**on-deterministic **p**olynomial time, because a non-deterministic computer can guess a certificate and then verify that certificate. In a parallel emulation, the computer would generate all possible certificates and then verify them in parallel. Generating one certificate is easy, because it only has polynomial length, but generating all of them is hard, because there are exponentially many strings of polynomial length.

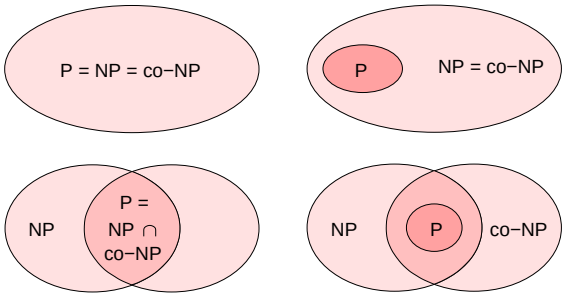

Figure 109: Four possible relations between the complexity classes P, NP, and co-NP.

Non-deterministic machine are at least as powerful as deterministic machines. It follows that every problem in P is also in NP, $\mathsf{P} \subseteq \mathsf{NP}$. Define

$$\text{co-NP} \quad = \quad \{L \mid \overline{L} = \{x \notin L\} \in \mathsf{NP}\},$$

which is the class of languages whose complement can be verified in non-deterministic polynomial time. It is not known whether or not $\mathsf{NP} = \text{co-NP}$. For example, it seems easy to verify that a graph is hamiltonian but it seems hard to verify that a graph is not hamiltonian. We said earlier that if $L \in \mathsf{P}$ then $\overline{L} \in \mathsf{P}$. Therefore, $\mathsf{P} \subseteq \text{co-NP}$. Hence, only the four relationships between the three complexity classes shown in Figure 109 are possible, but at this time we do not know which one is correct.

Problem reduction. We now develop the concept of reducing one problem to another, which is key in the construction of the class of NP-complete problems. The idea is to map or transform an instance of a first problem to an instance of a second problem and to map the solution to the second problem back to a solution to the first problem. For decision problems, the solutions are the same and need no transformation.

Language L_1 is *polynomial-time reducible* to language L_2, denoted $L_1 \leq_P L_2$, if there is a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that $x \in L_1$ iff $f(x) \in L_2$, for all $x \in \{0,1\}^*$. Now suppose that

L_1 is polynomial-time reducible to L_2 and that L_2 has a polynomial-time algorithm A_2 that decides L_2,

$$x \xrightarrow{f} f(x) \xrightarrow{A_2} \{0,1\}.$$

We can compose the two algorithms and obtain a polynomial-time algorithm $A_1 = A_2 \circ f$ that decides L_1. In other words, we gained an efficient algorithm for L_1 just by reducing it to L_2.

REDUCTION LEMMA. If $L_1 \leq_P L_2$ and $L_2 \in \mathsf{P}$ then $L_1 \in \mathsf{P}$.

In words, if L_1 is polynomial-time reducible to L_2 and L_2 is easy then L_1 is also easy. Conversely, if we know that L_1 is hard then we can conclude that L_2 is also hard. This motivates the following definition. A language $L \subseteq \{0,1\}^*$ is NP-*complete* if

(1) $L \in \mathsf{NP}$;

(2) $L' \leq_P L$, for every $L' \in \mathsf{NP}$.

Since *every* $L' \in \mathsf{NP}$ is polynomial-time reducible to L, *all* L' have to be easy for L to have a chance to be easy. The L' thus only provide evidence that L might indeed be hard. We say L is NP-*hard* if it satisfies (2) but not necessarily (1). The problems that satisfy (1) and (2) form the complexity class

$$\mathsf{NPC} \quad = \quad \{L \mid L \text{ is NP-complete}\}.$$

All these definitions would not mean much if we could not find any problems in NPC. The first step is the most difficult one. Once we have one problem in NPC we can get others using reductions.

Satisfying boolean formulas. Perhaps surprisingly, a first NP-complete problem has been found, namely the problem of satisfiability for logical expressions. A *boolean formula*, φ, consists of variables, $x_1, x_2, \ldots$, operators, $\neg, \wedge, \vee, \Longrightarrow, \ldots$, and parentheses. A *truth assignment* maps each variable to a boolean value, 0 or 1. The truth assignment *satisfies* if the formula evaluates to 1. The formula is *satisfiable* if there exists a satisfying truth assignment. Define SAT $= \{\varphi \mid \varphi \text{ is satisfiable}\}$. As an example consider the formula

$$\psi \quad = \quad (x_1 \Longrightarrow x_2) \Longleftrightarrow (x_2 \vee \neg x_1).$$

If we set $x_1 = x_2 = 1$ we get $(x_1 \Longrightarrow x_2) = 1$, $(x_2 \vee \neg x_1) = 1$ and therefore $\psi = 1$. It follows that $\psi \in$ SAT. In fact, all truth assignments evaluate to 1, which means that ψ is really a tautology. More generally, a boolean formula, φ, is satisfyable iff $\neg\varphi$ is not a tautology.

SATISFIABILITY THEOREM. We have SAT $\in$ NP and $L' \leq_P$ SAT for every $L' \in \mathsf{NP}$.

That SAT is in the class NP is easy to prove: just guess an assignment and verify that it satisfies. However, to prove that every $L' \in \mathsf{NP}$ can be reduced to SAT in polynomial time is quite technical and we omit the proof. The main idea is to use the polynomial-time algorithm that verifies L' and to construct a boolean formula from this algorithm. To formalize this idea, we would need a formal model of a computer, a Touring machine, which is beyond the scope of this course.

24 NP-Complete Problems

In this section, we discuss a number of NP-complete problems, with the goal to develop a feeling for what hard problems look like. Recognizing hard problems is an important aspect of a reliable judgement for the difficulty of a problem and the most promising approach to a solution. Of course, for NP-complete problems, it seems futile to work toward polynomial-time algorithms and instead we would focus on finding approximations or circumventing the problems altogether. We begin with a result on different ways to write boolean formulas.

Reduction to 3-satisfiability. We call a boolean variable or its negation a *literal*. The *conjunctive normal form* is a sequence of clauses connected by $\wedge$s, and each *clause* is a sequence of literals connected by $\vee$s. A formula is in *3-CNF* if it is in conjunctive normal form and each clause consists of three literals. It turns out that deciding the satisfiability of a boolean formula in 3-CNF is no easier than for a general boolean formula. Define 3-SAT $= \{\varphi \in \text{SAT} \mid \varphi \text{ is in 3-CNF}\}$. We prove the above claim by reducing SAT to 3-SAT.

SATISFIABILITY LEMMA. SAT $\leq_P$ 3-SAT.

PROOF. We take a boolean formula φ and transform it into 3-CNF in three steps.

`Step 1.` Think of φ as an expression and represent it as a binary tree. Each node is an operation that gets the input from its two children and forwards the output to its parent. Introduce a new variable for the output and define a new formula φ' for each node, relating the two input edges with the one output edge. Figure 110 shows the tree representation of the formula $\varphi = (x_1 \Longrightarrow x_2) \Longleftrightarrow (x_2 \vee \neg x_1)$. The new formula is

$$\begin{aligned}\varphi' \quad = \quad & (y_2 \Longleftrightarrow (x_1 \Longrightarrow x_2)) \\ & \wedge (y_3 \Longleftrightarrow (x_2 \vee \neg x_1)) \\ & \wedge (y_1 \Longleftrightarrow (y_2 \Longleftrightarrow y_3)) \wedge y_1.\end{aligned}$$

It should be clear that there is a satisfying assignment for φ iff there is one for φ'.

`Step 2.` Convert each clause into disjunctive normal form. The most mechanical way uses the truth table for each clause, as illustrated in Table 6. Each clause has at most three literals. For example, the negation of $y_2 \Longleftrightarrow (x_1 \Longrightarrow x_2)$ is equivalent to the disjunction of the conjunctions in the rightmost column. It follows that $y_2 \Longleftrightarrow (x_1 \Longrightarrow x_2)$ is equivalent to the negation of that disjunction, which by de Morgan's law is $(y_2 \vee x_1 \vee x_2) \wedge (y_2 \vee x_1 \vee \neg x_2) \wedge (y_2 \vee \neg x_1 \vee \neg x_2) \wedge (\neg y_2 \vee \neg x_1 \vee x_2)$.

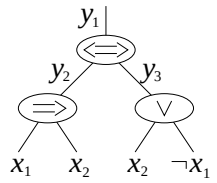

Figure 110: The tree representation of the formula φ. Incidentally, φ is a tautology, which means it is satisfied by every truth assignment. Equivalently, $\neg\varphi$ is not satisfiable.

y_2	x_1	x_2	$y_2 \Leftrightarrow (x_1 \Rightarrow x_2)$	prohibited
0	0	0	0	$\neg y_2 \wedge \neg x_1 \wedge \neg x_2$
0	0	1	0	$\neg y_2 \wedge \neg x_1 \wedge x_2$
0	1	0	1	
0	1	1	0	$\neg y_2 \wedge x_1 \wedge x_2$
1	0	0	1	
1	0	1	1	
1	1	0	0	$y_2 \wedge x_1 \wedge \neg x_2$
1	1	1	1	

Table 6: Conversion of a clause into a disjunction of conjunctions of at most three literals each.

`Step 3.` The clauses with fewer than three literals can be expanded by adding new variables. For example $a \vee b$ is expanded to $(a \vee b \vee p) \wedge (a \vee b \vee \neg p)$ and (a) is expanded to $(a \vee p \vee q) \wedge (a \vee p \vee \neg q) \wedge (a \vee \neg p \vee q) \wedge (a \vee \neg p \vee \neg q)$.

Each step takes only polynomial time. At the end, we get an equivalent formula in 3-conjunctive normal form. □

We note that clauses of length three are necessary to make the satisfiability problem hard. Indeed, there is a polynomial-time algorithm that decides the satisfiability of a formula in 2-CNF.

NP-completeness proofs. Using polynomial-time reductions, we can show fairly mechanically that problems are NP-complete, if they are. A key property is the transitivity of $\leq_P$, that is, if $L' \leq_P L_1$ and $L_1 \leq_P L_2$ then $L' \leq_P L_2$, as can be seen by composing the two polynomial-time computable functions to get a third one.

REDUCTION LEMMA. Let $L_1, L_2 \subseteq \{0,1\}^*$ and assume $L_1 \leq_P L_2$. If L_1 is NP-hard and $L_2 \in$ NP then $L_2 \in$ NPC.

A generic NP-completeness proof thus follows the steps outline below.

`Step 1.` Prove that $L_2 \in$ NP.

`Step 2.` Select a known NP-hard problem, L_1, and find a polynomial-time computable function, f, with $x \in L_1$ iff $f(x) \in L_2$.

This is what we did for $L_2 =$ 3-SAT and $L_1 =$ SAT. Therefore 3-SAT $\in$ NPC. Currently, there are thousands of problems known to be NP-complete. This is often considered evidence that P $\neq$ NP, which can be the case only if P $\cap$ NPC $= \emptyset$, as drawn in Figure 111.

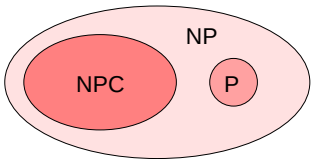

Figure 111: Possible relation between P, NPC, and NP.

Cliques and independent sets. There are many NP-complete problems on graphs. A typical such problem asks for the largest complete subgraph. Define a *clique* in an undirected graph $G = (V, E)$ as a subgraph (W, F) with $F = \binom{W}{2}$. Given G and an integer k, the CLIQUE problem asks whether or not there is a clique of k or more vertices.

CLAIM. CLIQUE $\in$ NPC.

PROOF. Given k vertices in G, we can verify in polynomial time whether or not they form a complete graph. Thus CLIQUE $\in$ NP. To prove property (2), we show that 3-SAT $\leq_P$ CLIQUE. Let φ be a boolean formula in 3-CNF consisting of k clauses. We construct a graph as follows:

(i) each clause is replaced by three vertices;

(ii) two vertices are connected by an edge if they do not belong to the same clause and they are not negations of each other.

In a satisfying truth assignment, there is at least one true literal in each clause. The true literals form a clique. Conversely, a clique of k or more vertices covers all clauses and thus implies a satisfying truth assignment. □

It is easy to decide in time $O(k^2 n^{k+2})$ whether or not a graph of n vertices has a clique of size k. If k is a constant, the running time of this algorithm is polynomial in n. For the CLIQUE problem to be NP-complete it is therefore essential that k be a variable that can be arbitrarily large. We use the NP-completeness of finding large cliques to prove the NP-completeness of large sets of pairwise non-adjacent vertices. Let $G = (V, E)$ be an undirected graph. A subset $W \subseteq V$ is *independent* if none of the vertices in W are adjacent or, equivalently, if $E \cap \binom{W}{2} = \emptyset$. Given G and an integer k, the INDEPENDENT SET problem asks whether or not there is an independent set of k or more vertices.

CLAIM. INDEPENDENT SET $\in$ NPC.

PROOF. It is easy to verify that there is an independent set of size k: just guess a subset of k vertices and verify that no two are adjacent.

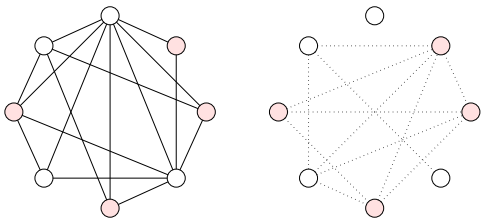

Figure 112: The four shaded vertices form an independent set in the graph on the left and a clique in the complement graph on the right.

We complete the proof by reducing the CLIQUE to the INDEPENDENT SET problem. As illustrated in Figure 112, $W \subseteq V$ is independent iff W defines a clique in the complement graph, $\overline{G} = (V, \binom{V}{2} - E)$. To prove CLIQUE $\leq_P$ INDEPENDENT SET, we transform an instance H, k of the CLIQUE problem to the instance $G = \overline{H}, k$ of the INDEPENDENT SET problem. G has an independent set of size k or larger iff H has a clique of size k or larger. □

Various NP-complete graph problems. We now describe a few NP-complete problems for graphs without proving that they are indeed NP-complete. Let $G = (V, E)$ be an undirected graph with n vertices and k a positive integer, as before. The following problems defined for G and k are NP-complete.

An *ℓ-coloring* of G is a function $\chi : V \to [\ell]$ with $\chi(u) \neq \chi(v)$ whenever u and v are adjacent. The CHROMATIC NUMBER problem asks whether or not G has an ℓ-coloring with $\ell \leq k$. The problem remains NP-complete

for fixed $k \geq 3$. For $k = 2$, the CHROMATIC NUMBER problem asks whether or not G is bipartite, for which there is a polynomial-time algorithm.

The *bandwidth* of G is the minimum ℓ such that there is a bijection $\beta : V \to [n]$ with $|\beta(u) - \beta(v)| \leq \ell$ for all adjacent vertices u and v. The BANDWIDTH problem asks whether or not the bandwidth of G is k or less. The problem arises in linear algebra, where we permute rows and columns of a matrix to move all non-zero elements of a square matrix as close to the diagonal as possible. For example, if the graph is a simple path then the bandwidth is 1, as can be seen in Figure 113. We can transform the

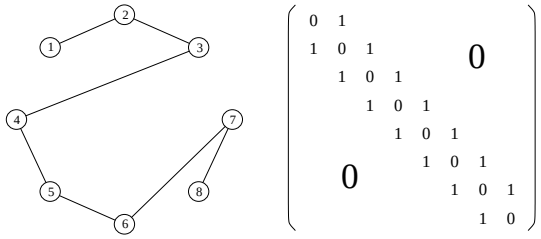

Figure 113: Simple path and adjacency matrix with rows and columns ordered along the path.

adjacency matrix of G such that all non-zero diagonals are at most the bandwidth of G away from the main diagonal.

Assume now that the graph G is complete, $E = \binom{V}{2}$, and that each edge, uv, has a positive integer weight, $w(uv)$. The TRAVELING SALESMAN problem asks whether there is a permutation $u_0, u_1, \ldots, u_{n-1}$ of the vertices such that the sum of edges connecting contiguous vertices (and the last vertex to the first) is k or less,

$$\sum_{i=0}^{n-1} w(u_i u_{i+1}) \quad \leq \quad k,$$

where indices are taken modulo n. The problem remains NP-complete if $w : E \to \{1, 2\}$ (reduction to HAMILTONIAN CYCLE problem), and also if the vertices are points in the plane and the weight of an edge is the Euclidean distance between the two endpoints.

Set systems. Simple graphs are set systems in which the sets contain only two elements. We now list a few NP-complete problems for more general set systems. Letting V be a finite set, $C \subseteq 2^V$ a set system, and k a positive integer, the following problems are NP-complete.

The PACKING problem asks whether or not C has k or more mutually disjoint sets. The problem remains NP-complete if no set in C contains more than three elements, and there is a polynomial-time algorithm if every set contains two elements. In the latter case, the set system is a graph and a maximum packing is a maximum matching.

The COVERING problem asks whether or not C has k or fewer subsets whose union is V. The problem remains NP-complete if no set in C contains more than three elements, and there is a polynomial-time algorithm if every sets contains two elements. In the latter case, the set system is a graph and the minimum cover can be constructed in polynomial time from a maximum matching.

Suppose every element $v \in V$ has a positive integer weight, $w(v)$. The PARTITION problem asks whether there is a subset $U \subseteq V$ with

$$\sum_{u \in U} w(u) \quad = \quad \sum_{v \in V - U} w(v).$$

The problem remains NP-complete if we require that U and $V - U$ have the same number of elements.

25 Approximation Algorithms

Many important problems are NP-hard and just ignoring them is not an option. There are indeed many things one can do. For problems of small size, even exponential-time algorithms can be effective and special subclasses of hard problems sometimes have polynomial-time algorithms. We consider a third coping strategy appropriate for optimization problems, which is computing almost optimal solutions in polynomial time. In case the aim is to maximize a positive cost, a $\varrho(n)$-*approximation algorithm* is one that guarantees to find a solution with cost $C \geq C^*/\varrho(n)$, where C^* is the maximum cost. For minimization problems, we would require $C \leq C^*\varrho(n)$. Note that $\varrho(n) \geq 1$ and if $\varrho(n) = 1$ then the algorithm produces optimal solutions. Ideally, ϱ is a constant but sometime even this is not achievable in polynomial time.

Vertex cover. The first problem we consider is finding the minimum set of vertices in a graph $G = (V, E)$ that covers all edges. Formally, a subset $V' \subseteq V$ is a *vertex cover* if every edge has at least one endpoint in V'. Observe that V' is a vertex cover iff $V - V'$ is an independent set. Finding a minimum vertex cover is therefore equivalent to finding a maximum independent set. Since the latter problem is NP-complete, we conclude that finding a minimum vertex cover is also NP-complete. Here is a straightforward algorithm that achieves approximation ratio $\varrho(n) = 2$, for all $n = |V|$.

$V' = \emptyset$; $E' = E$;
`while` $E' \neq \emptyset$ `do`
  select an arbitrary edge uv in E';
  add u and v to V';
  remove all edges incident to u or v from E'
`endwhile`.

Clearly, V' is a vertex cover. Using adjacency lists with links between the two copies of an edge, the running time is O($n + m$), where m is the number of edges. Furthermore, we have $\varrho = 2$ because every cover must pick at least one vertex of each edge uv selected by the algorithm, hence $C \leq 2C^*$. Observe that this result does not imply a constant approximation ratio for the maximum independent set problem. We have $|V - V'| = n - C \geq n - 2C^*$, which we have to compare with $n - C^*$, the size of the maximum independent set. For $C^* = \frac{n}{2}$, the approximation ratio is unbounded.

Let us contemplate the argument we used to relate C and C^*. The set of edges uv selected by the algorithm is a *matching*, that is, a subset of the edges so that no two share a vertex. The size of the minimum vertex cover is at least the size of the largest possible matching. The algorithm finds a matching and since it picks two vertices per edge, we are guaranteed at most twice as many vertices as needed. This pattern of bounding C^* by the size of another quantity (in this case the size of the largest matching) is common in the analysis of approximation algorithms. Incidentally, for bipartite graphs, the size of the largest matching is equal to the size of the smallest vertex cover. Furthermore, there is a polynomial-time algorithm for computing them.

Traveling salesman. Second, we consider the traveling salesman problem, which is formulated for a complete graph $G = (V, E)$ with a positive integer cost function $c : E \to \mathbb{Z}_+$. A *tour* in this graph is a Hamiltonian cycle and the problem is finding the tour, A, with minimum total cost, $c(A) = \sum_{uv \in A} c(uv)$. Let us first assume that the cost function satisfies the triangle inequality, $c(uw) \leq c(uv) + c(vw)$ for all $u, v, w \in V$. It can be shown that the problem of finding the shortest tour remains NP-complete even if we restrict it to weighted graphs that satisfy this inequality. We formulate an algorithm based on the observation that the cost of every tour is at least the cost of the minimum spanning tree, $C^* \geq c(T)$.

1 Construct the minimum spanning tree T of G.

2 Return the preorder sequence of vertices in T.

Using Prim's algorithm for the minimum spanning tree, the running time is O(n^2). Figure 114 illustrates the algorithm. The preorder sequence is only defined if we have

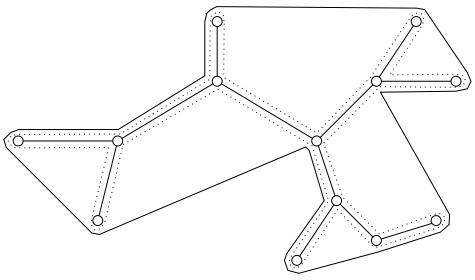

Figure 114: The solid minimum spanning tree, the dotted traversal using each edge of the tree twice, and the solid tour obtained by taking short-cuts.

a root and the neighbors of each vertex are ordered, but

we may choose both arbitrarily. The cost of the returned tour is at most twice the cost of the minimum spanning tree. To see this, consider traversing each edge of the minimum spanning tree twice, once in each direction. Whenever a vertex is visited more than once, we take the direct edge connecting the two neighbors of the second copy as a short-cut. By the triangle inequality, this substitution can only decrease the overall cost of the traversal. It follows that $C \leq 2c(T) \leq 2C^*$.

The triangle inequality is essential in finding a constant approximation. Indeed, without it we can construct instances of the problem for which finding a constant approximation is NP-hard. To see this, transform an unweighted graph $G' = (V', E')$ to the complete weighted graph $G = (V, E)$ with

$$c(uv) \quad = \quad \begin{cases} 1 & \text{if } uv \in E', \\ \varrho n + 1 & \text{otherwise.} \end{cases}$$

Any ϱ-approximation algorithm must return the Hamiltonian cycle of G', if there is one.

Set cover. Third, we consider the problem of covering a set X with sets chosen from a set system $\mathcal{F}$. We assume the set is the union of sets in the system, $X = \bigcup \mathcal{F}$. More precisely, we are looking for a smallest subsystem $\mathcal{F}' \subseteq \mathcal{F}$ with $X = \bigcup \mathcal{F}'$. The *cost* of this subsystem is the number of sets it contains, $|\mathcal{F}'|$. See Figure 115 for an illustration of the problem. The vertex cover problem

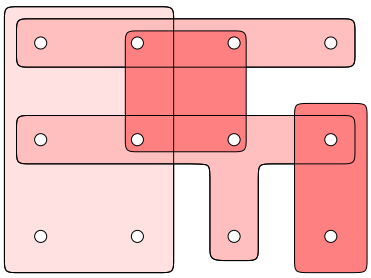

Figure 115: The set X of twelve dots can be covered with four of the five sets in the system.

is a special case: $X = E$ and $\mathcal{F}$ contains all subsets of edges incident to a common vertex. It is special because each element (edge) belongs to exactly two sets. Since we no longer have a bound on the number of sets containing a single element, it is not surprising that the algorithm for vertex covers does not extend to a constant-approximation algorithm for set covers. Instead, we consider the following greedy approach that selects, at each step, the set containing the maximum number of yet uncovered elements.

$\mathcal{F}' = \emptyset$; $X' = X$;
`while` $X' \neq \emptyset$ `do`
  select $S \in \mathcal{F}$ maximizing $|S \cap X'|$;
  $\mathcal{F}' = \mathcal{F}' \cup \{S\}$; $X' = X' - S$
`endwhile`.

Using a sparse matrix representation of the set system (similar to an adjacency list representation of a graph), we can run the algorithm in time proportional to the total size of the sets in the system, $n = \sum_{S \in \mathcal{F}} |S|$. We omit the details.

Analysis. More interesting than the running time is the analysis of the approximation ratio the greedy algorithm achieves. It is convenient to have short notation for the d-th harmonic number, $H_d = \sum_{i=1}^{d} \frac{1}{i}$ for $d \geq 0$. Recall that $H_d \leq 1 + \ln d$ for $d \geq 1$. Let the size of the largest set in the system be $m = \max\{|S| \mid S \in \mathcal{F}\}$.

CLAIM. The greedy method is an H_m-approximation algorithm for the set cover problem.

PROOF. For each set S selected by the algorithm, we distribute \$1 over the $|S \cap X'|$ elements covered for the first time. Let c_x be the cost allocated this way to $x \in X$. We have $|\mathcal{F}'| = \sum_{x \in X} c_x$. If x is covered the first time by the i-th selected set, S_i, then

$$c_x \quad = \quad \frac{1}{|S_i - (S_1 \cup \ldots \cup S_{i-1})|}.$$

We have $|\mathcal{F}'| \leq \sum_{S \in \mathcal{F}^*} \sum_{x \in S} c_x$ because the optimal cover, $\mathcal{F}^*$, contains each element x at least once. We will prove shortly that $\sum_{x \in S} c_x \leq H_{|S|}$ for every set $S \in \mathcal{F}$. It follows that

$$|\mathcal{F}'| \quad \leq \quad \sum_{S \in \mathcal{F}^*} H_{|S|} \quad \leq \quad H_m |\mathcal{F}^*|,$$

as claimed. □

For $m = 3$, we get $\varrho = H_3 = \frac{11}{6}$. This implies that for graphs with vertex-degrees at most 3, the greedy algorithm guarantees a vertex cover of size at most $\frac{11}{6}$ times the optimum, which is better than the ratio 2 guaranteed by our first algorithm.

We still need to prove that the sum of costs c_x over the elements of a set S in the system is bounded from above by $H_{|S|}$. Let u_i be the number of elements in S that are

not covered by the first i selected sets, $u_i = |S - (S_1 \cup \ldots \cup S_i)|$, and observe that the numbers do not increase. Let u_{k-1} be the last non-zero number in the sequence, so $|S| = u_0 \geq \ldots \geq u_{k-1} > u_k = 0$. Since $u_{i-1} - u_i$ is the number of elements in S covered the first time by S_i, we have

$$\sum_{x \in S} c_x = \sum_{i=1}^{k} \frac{u_{i-1} - u_i}{|S_i - (S_1 \cup \ldots \cup S_{i-1})|}.$$

We also have $u_{i-1} \leq |S_i - (S_1 \cup \ldots \cup S_{i-1})|$, for all $i \leq k$, because of the greedy choice of S_i. If this were not the case, the algorithm would have chosen S instead of S_i in the construction of $\mathcal{F}'$. The problem thus reduces to bounding the sum of ratios $\frac{u_{i-1}-u_i}{u_{i-1}}$. It is not difficult to see that this sum can be at least logarithmic in the size of S. Indeed, if we choose u_i about half the size of u_{i-1}, for all $i \geq 1$, then we have logarithmically many terms, each roughly $\frac{1}{2}$. We use a sequence of simple arithmetic manipulations to prove that this lower bound is asymptotically tight:

$$\begin{aligned} \sum_{x \in S} c_x &\leq \sum_{i=1}^{k} \frac{u_{i-1} - u_i}{u_{i-1}} \\ &= \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{u_{i-1}}. \end{aligned}$$

We now replace the denominator by $j \leq u_{i-1}$ to form a telescoping series of harmonic numbers and get

$$\begin{aligned} \sum_{x \in S} c_x &\leq \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j} \\ &= \sum_{i=1}^{k} \left(\sum_{j=1}^{u_{i-1}} \frac{1}{j} - \sum_{j=1}^{u_i} \frac{1}{j} \right) \\ &= \sum_{i=1}^{k} (H_{u_{i-1}} - H_{u_i}). \end{aligned}$$

This is equal to $H_{u_0} - H_{u_k} = H_{|S|}$, which fills the gap left in the analysis of the greedy algorithm.

Seventh Homework Assignment

The purpose of this assignment is to help you prepare for the final exam. Solutions will neither be graded nor even collected.

Problem 1. ($20 = 5 + 15$ points). Consider the class of satisfiable boolean formulas in conjunctive normal form in which each clause contains two literals, 2-SAT $= \{\varphi \in \text{SAT} \mid \varphi \text{ is 2-CNF}\}$.

(a) Is 2-SAT $\in$ NP?

(b) Is there a polynomial-time algorithm for deciding whether or not a boolean formula in 2-CNF is satisfiable? If your answer is yes, then describe and analyze your algorithm. If your answer is no, then show that 2-SAT $\in$ NPC.

Problem 2. (20 points). Let A be a finite set and f a function that maps every $a \in A$ to a positive integer $f(a)$. The PARTITION problem asks whether or not there is a subset $B \subseteq A$ such that

$$\sum_{b \in B} f(b) \quad = \quad \sum_{a \in A - B} f(a).$$

We have learned that the PARTITION problem is NP-complete. Given positive integers j and k, the SUM OF SQUARES problem asks whether or not A can be partitioned into j disjoint subsets, $A = B_1 \dot\cup B_2 \dot\cup \ldots \dot\cup B_j$, such that

$$\sum_{i=1}^{j} \left(\sum_{a \in B_i} f(a) \right)^2 \quad \leq \quad k.$$

Prove that the SUM OF SQUARES problem is NP-complete.

Problem 3. ($20 = 10+10$ points). Let G be an undirected graph. A path in G is *simple* if it contains each vertex at most once. Specifying two vertices u, v and a positive integer k, the LONGEST PATH problem asks whether or not there is a simple path connecting u and v whose length is k or longer.

(a) Give a polynomial-time algorithm for the LONGEST PATH problem or show that it is NP-hard.

(b) Revisit (a) under the assumption that G is directed and acyclic.

Problem 4. ($20 = 10 + 10$ points). Let $A \subseteq 2^V$ be an abstract simplicial complex over the finite set V and let k be a positive integer.

(a) Is it NP-hard to decide whether A has k or more disjoint simplices?

(b) Is it NP-hard to decide whether A has k or fewer simplices whose union is V?

Problem 5. (20 points). Let $G = (V, E)$ be an undirected, bipartite graph and recall that there is a polynomial-time algorithm for constructing a maximum matching. We are interested in computing a minimum set of matchings such that every edge of the graph is a member of at least one of the selected matchings. Give a polynomial-time algorithm constructing an $\mathrm{O}(\log n)$ approximation for this problem.

13 Graph Search

We can think of graphs as generalizations of trees: they consist of nodes and edges connecting nodes. The main difference is that graphs do not in general represent hierarchical organizations.

Types of graphs. Different applications require different types of graphs. The most basic type is the *simple undirected graph* that consists of a set V of *vertices* and a set E of *edges*. Each edge is an unordered pair (a set) of two vertices. We always assume V is finite, and we write

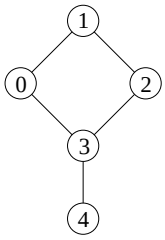

Figure 50: A simple undirected graph with vertices $0, 1, 2, 3, 4$ and edges $\{0,1\}, \{1,2\}, \{2,3\}, \{3,0\}, \{3,4\}$.

$\binom{V}{2}$ for the collection of all unordered pairs. Hence E is a subset of $\binom{V}{2}$. Note that because E is a set, each edge can occur only once. Similarly, because each edge is a set (of two vertices), it cannot connect to the same vertex twice. Vertices u and v are *adjacent* if $\{u, v\} \in E$. In this case u and v are called *neighbors*. Other types of graphs are

directed:	$E \subseteq V \times V$.
weighted:	has a weighting function $w : E \to \mathbb{R}$.
labeled:	has a labeling function $\ell : V \to \mathbb{Z}$.
non-simple:	there are loops and multi-edges.

A *loop* is like an edge, except that it connects to the same vertex twice. A *multi-edge* consists of two or more edges connecting the same two vertices.

Representation. The two most popular data structures for graphs are direct representations of adjacency. Let $V = \{0, 1, \ldots, n-1\}$ be the set of vertices. The *adjacency matrix* is the n-by-n matrix $A = (a_{ij})$ with

$$a_{ij} = \begin{cases} 1 & \text{if } \{i,j\} \in E, \\ 0 & \text{if } \{i,j\} \notin E. \end{cases}$$

For undirected graphs, we have $a_{ij} = a_{ji}$, so A is symmetric. For weighted graphs, we encode more information than just the existence of an edge and define a_{ij} as the weight of the edge connecting i and j. The adjacency matrix of the graph in Figure 50 is

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix},$$

which is symmetric. Irrespective of the number of edges,

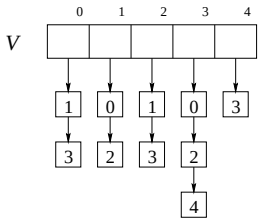

Figure 51: The adjacency list representation of the graph in Figure 50. Each edge is represented twice, once for each endpoint.

the adjacency matrix has n^2 elements and thus requires a quadratic amount of space. Often, the number of edges is quite small, maybe not much larger than the number of vertices. In these cases, the adjacency matrix wastes memory, and a better choice is a sparse matrix representation referred to as *adjacency lists*, which is illustrated in Figure 51. It consists of a linear array V for the vertices and a list of neighbors for each vertex. For most algorithms, we assume that vertices and edges are stored in structures containing a small number of fields:

```
struct Vertex {int d, f, π; Edge *adj};
struct Edge {int v; Edge *next}.
```

The d, f, π fields will be used to store auxiliary information used or created by the algorithms.

Depth-first search. Since graphs are generally not ordered, there are many sequences in which the vertices can be visited. In fact, it is not entirely straightforward to make sure that each vertex is visited once and only once. A useful method is depth-first search. It uses a global variable, *time*, which is incremented and used to leave time-stamps behind to avoid repeated visits.

```
    void VISIT(int i)
1     time++; V[i].d = time;
      forall outgoing edges ij do
2       if V[j].d = 0 then
3         V[j].π = i; VISIT(j)
        endif
      endfor;
4     time++; V[i].f = time.
```

The test in line 2 checks whether the neighbor j of i has already been visited. The assignment in line 3 records that the vertex is visited *from* vertex i. A vertex is first stamped in line 1 with the time at which it is encountered. A vertex is second stamped in line 4 with the time at which its visit has been completed. To prepare the search, we initialize the global time variable to 0, label all vertices as not yet visited, and call VISIT for all yet unvisited vertices.

```
time = 0;
forall vertices i do V[i].d = 0 endfor;
forall vertices i do
  if V[i].d = 0 then V[i].π = 0; VISIT(i) endif
endfor.
```

Let n be the number of vertices and m the number of edges in the graph. Depth-first search visits every vertex once and examines every edge twice, once for each endpoint. The running time is therefore $O(n + m)$, which is proportional to the size of the graph and therefore optimal.

DFS forest. Figure 52 illustrates depth-first search by showing the time-stamps d and f and the pointers π indicating the predecessors in the traversal. We call an edge $\{i, j\} \in E$ a *tree edge* if $i = V[j].\pi$ or $j = V[i].\pi$ and a *back edge*, otherwise. The tree edges form the *DFS forest* of the graph. The forest is a tree if the graph is connected and a collection of two or more trees if it is not connected. Figure 53 shows the DFS forest of the graph in Figure 52 which, in this case, consists of a single tree. The time-stamps d are consistent with the preorder traversal of the DFS forest. The time-stamps f are consistent with the postorder traversal. The two stamps can be used to decide, in constant time, whether two nodes in the forest live in different subtrees or one is a descendent of the other.

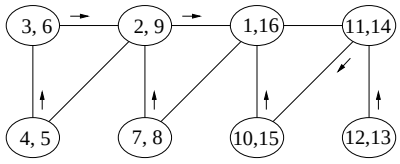

Figure 52: The traversal starts at the vertex with time-stamp 1. Each node is stamped twice, once when it is first encountered and another time when its visit is complete.

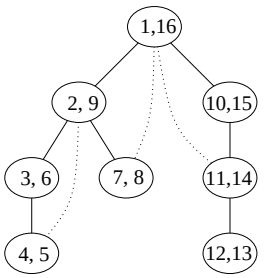

Figure 53: Tree edges are solid and back edges are dotted.

NESTING LEMMA. Vertex j is a proper descendent of vertex i in the DFS forest iff $V[i].d < V[j].d$ as well as $V[j].f < V[i].f$.

Similarly, if you have a tree and the preorder and postorder numbers of the nodes, you can determine the relation between any two nodes in constant time.

Directed graphs and relations. As mentioned earlier, we have a *directed graph* if all edges are directed. A directed graph is a way to think and talk about a mathematical relation. A typical problem where relations arise is scheduling. Some tasks are in a definite order while others are unrelated. An example is the scheduling of undergraduate computer science courses, as illustrated in Figure 54. Abstractly, a *relation* is a pair (V, E), where $V = \{0, 1, \ldots, n-1\}$ is a finite set of elements and $E \subseteq V \times V$ is a finite set of ordered pairs. Instead of

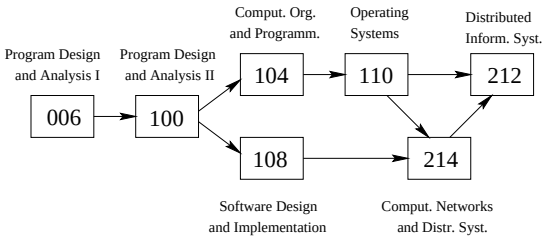

Figure 54: A subgraph of the CPS course offering. The courses CPS104 and CPS108 are incomparable, CPS104 is a predecessor of CPS110, and so on.

$(i,j) \in E$ we write $i \prec j$ and instead of (V,E) we write $(V,\prec)$. If $i \prec j$ then i is a *predecessor* of j and j is a *successor* of i. The terms relation, directed graph, digraph, and network are all synonymous.

Directed acyclic graphs. A *cycle* in a relation is a sequence $i_0 \prec i_1 \prec \ldots \prec i_k \prec i_0$. Even $i_0 \prec i_0$ is a cycle. A *linear extension* of $(V,\prec)$ is an ordering $j_0, j_1, \ldots, j_{n-1}$ of the elements that is consistent with the relation. Formally this means that $j_k \prec j_\ell$ implies $k < \ell$. A directed graph without cycle is a *directed acyclic graph*.

EXTENSION LEMMA. $(V,\prec)$ has a linear extension iff it contains no cycle.

PROOF. "$\Longrightarrow$" is obvious. We prove "$\Longleftarrow$" by induction. A vertex $s \in V$ is called a *source* if it has no predecessor. Assuming $(V,\prec)$ has no cycle, we can prove that V has a source by following edges against their direction. If we return to a vertex that has already been visited, we have a cycle and thus a contradiction. Otherwise we get stuck at a vertex s, which can only happen because s has no predecessor, which means s is a source.

Let $U = V - \{s\}$ and note that $(U,\prec)$ is a relation that is smaller than $(V,\prec)$. Hence $(U,\prec)$ has a linear extension by induction hypothesis. Call this extension X and note that s, X is a linear extension of $(V,\prec)$. □

Topological sorting with queue. The problem of constructing a linear extension is called *topological sorting*. A natural and fast algorithm follows the idea of the proof: find a source s, print s, remove s, and repeat. To expedite the first step of finding a source, each vertex maintains its number of predecessors and a queue stores all sources. First, we initialize this information.

```
forall vertices j do V[j].d = 0 endfor;
forall vertices i do
  forall successors j of i do V[j].d++ endfor
endfor;
forall vertices j do
  if V[j].d = 0 then ENQUEUE(j) endif
endfor.
```

Next, we compute the linear extension by repeated deletion of a source.

```
while queue is non-empty do
  s = DEQUEUE;
  forall successors j of s do
    V[j].d--;
    if V[j].d = 0 then ENQUEUE(j) endif
  endfor
endwhile.
```

The running time is linear in the number of vertices and edges, namely $O(n+m)$. What happens if there is a cycle in the digraph? We illustrate the above algorithm for the directed acyclic graph in Figure 55. The sequence of vertices added to the queue is also the linear extension computed by the algorithm. If the process starts at vertex a and if the successors of a vertex are ordered by name then we get a, f, d, g, c, h, b, e, which we can check is indeed a linear extension of the relation.

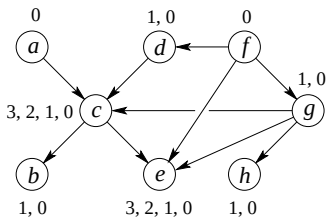

Figure 55: The numbers next to each vertex count the predecessors, which decreases during the algorithm.

Topological sorting with DFS. Another algorithm that can be used for topological sorting is depth-first search. We output a vertex when its visit has been completed, that is, when all its successors and their successors and so on have already been printed. The linear extension is therefore generated from back to front. Figure 56 shows the same digraph as Figure 55 and labels vertices with time

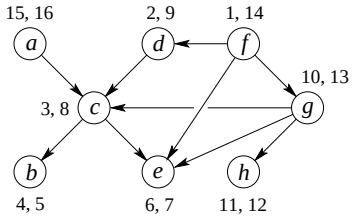

Figure 56: The numbers next to each vertex are the two time stamps applied by the depth-first search algorithm. The first number gives the time the vertex is encountered, and the second when the visit has been completed.

stamps. Consider the sequence of vertices in the order of decreasing second time stamp:

$$a(16), f(14), g(13), h(12), d(9), c(8), e(7), b(5).$$

Although this sequence is different from the one computed by the earlier algorithm, it is also a linear extension of the relation.

14 Shortest Paths

One of the most common operations in graphs is finding shortest paths between vertices. This section discusses three algorithms for this problem: breadth-first search for unweighted graphs, Dijkstra's algorithm for weighted graphs, and the Floyd-Warshall algorithm for computing distances between all pairs of vertices.

Breadth-first search. We call a graph *connected* if there is a path between every pair of vertices. A *(connected) component* is a maximal connected subgraph. Breadth-first search, or BFS, is a way to search a graph. It is similar to depth-first search, but while DFS goes as deep as quickly as possible, BFS is more cautious and explores a broad neighborhood before venturing deeper. The starting point is a vertex s. An example is shown in Figure 57. As

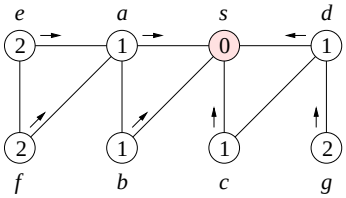

Figure 57: A sample graph with eight vertices and ten edges labeled by breath-first search. The label increases from a vertex to its successors in the search.

before, we call and edge a *tree edge* if it is traversed by the algorithm. The tree edges define the *BFS tree*, which we can use to redraw the graph in a hierarchical manner, as in Figure 58. In the case of an undirected graph, no non-tree edge can connect a vertex to an ancestor in the BFS tree. Why? We use a queue to turn the idea into an algorithm.

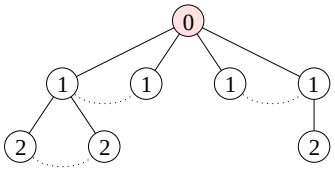

Figure 58: The tree edges in the redrawing of the graph in Figure 57 are solid, and the non-tree edges are dotted.

First, the graph and the queue are initialized.

`forall` vertices i `do` $V[i].d = -1$ `endfor`;
$V[s].d = 0$;
MAKEQUEUE; ENQUEUE(s); SEARCH.

A vertex is processed by adding its unvisited neighbors to the queue. They will be processed in turn.

```
void SEARCH
  while queue is non-empty do
    i = DEQUEUE;
    forall neighbors j of i do
      if V[j].d = −1 then
        V[j].d = V[i].d + 1; V[j].π = i;
        ENQUEUE(j)
      endif
    endfor
  endwhile.
```

The label $V[i].d$ assigned to vertex i during the traversal is the minimum number of edges of any path from s to i. In other words, $V[i].d$ is the length of the shortest path from s to i. The running time of BFS for a graph with n vertices and m edges is $O(n + m)$.

Single-source shortest path. BFS can be used to find shortest paths in unweighted graphs. We now extend the algorithm to weighted graphs. Assume V and E are the sets of vertices and edges of a simple, undirected graph with a positive weighting function $w : E \to \mathbb{R}_+$. The *length* or *weight* of a path is the sum of the weights of its edges. The *distance* between two vertices is the length of the shortest path connecting them. For a given source $s \in V$, we study the problem of finding the distances and shortest paths to all other vertices. Figure 59 illustrates the problem by showing the shortest paths to the source s. In

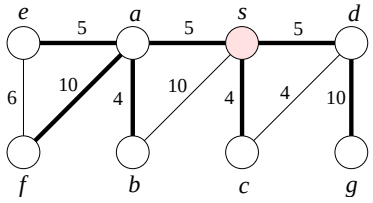

Figure 59: The bold edges form shortest paths and together the shortest path tree with root s. It differs by one edge from the breadth-first tree shown in Figure 57.

the non-degenerate case, in which no two paths have the same length, the union of all shortest paths to s is a tree, referred to as the *shortest path tree*. In the degenerate case, we can break ties such that the union of paths is a tree.

As before, we grow a tree starting from s. Instead of a queue, we use a priority queue to determine the next vertex to be added to the tree. It stores all vertices not yet in the

tree and uses $V[i].d$ for the priority of vertex i. First, we initialize the graph and the priority queue.

$V[s].d = 0$; $V[s].\pi = -1$; INSERT(s);
`forall` vertices $i \neq s$ `do`
  $V[i].d = \infty$; INSERT(i)
`endfor`.

After initialization the priority queue stores s with priority 0 and all other vertices with priority ∞.

Dijkstra's algorithm. We mark vertices in the tree to distinguish them from vertices that are not yet in the tree. The priority queue stores all unmarked vertices i with priority equal to the length of the shortest path that goes from i in one edge to a marked vertex and then to s using only marked vertices.

`while` priority queue is non-empty `do`
  $i =$ EXTRACTMIN; mark i;
  `forall` neighbors j of i `do`
    `if` j is unmarked `then`
      $V[j].d = \min\{w(ij) + V[i].d, V[j].d\}$
    `endif`
  `endfor`
`endwhile`.

Table 3 illustrates the algorithm by showing the information in the priority queue after each iteration of the while-loop operating on the graph in Figure 59. The marking mechanism is not necessary but clarifies the process. The algorithm performs n EXTRACTMIN operations and at most m DECREASEKEY operations. We compare the running time under three different data structures used to represent the priority queue. The first is a linear array, as originally proposed by Dijkstra, the second is a heap, and the third is a Fibonacci heap. The results are shown in Table 4. We get the best result with Fibonacci heaps for which the total running time is O($n \log n + m$).

s	**0**							
a	∞	5	**5**					
b	∞	10	10	9	**9**			
c	∞	**4**						
d	∞	5	5	**5**				
e	∞	∞	∞	10	10	**10**		
f	∞	∞	∞	15	15	15	**15**	
g	∞	∞	∞	∞	15	15	15	**15**

Table 3: Each column shows the contents of the priority queue. Time progresses from left to right.

	array	heap	F-heap
EXTRACTMINS	n^2	$n \log n$	$n \log n$
DECREASEKEYS	m	$m \log m$	m

Table 4: Running time of Dijkstra's algorithm for three different implementations of the priority queue holding the yet unmarked vertices.

Correctness. It is not entirely obvious that Dijkstra's algorithm indeed finds the shortest paths to s. To show that it does, we inductively prove that it maintains the following two invariants.

(A) For every unmarked vertex j, $V[j].d$ is the length of the shortest path from j to s that uses only marked vertices other than j.

(B) For every marked vertex i, $V[i].d$ is the length of the shortest path from i to s.

PROOF. Invariant (A) is true at the beginning of Dijkstra's algorithm. To show that it is maintained throughout the process, we need to make sure that shortest paths are computed correctly. Specifically, if we assume Invariant (B) for vertex i then the algorithm correctly updates the priorities $V[j].d$ of all neighbors j of i, and no other priorities change.

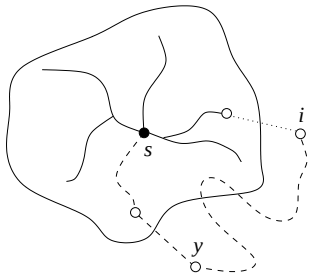

Figure 60: The vertex y is the last unmarked vertex on the hypothetically shortest, dashed path that connects i to s.

At the moment vertex i is marked, it minimizes $V[j].d$ over all unmarked vertices j. Suppose that, at this moment, $V[i].d$ is not the length of the shortest path from i to s. Because of Invariant (A), there is at least one other unmarked vertex on the shortest path. Let the last such vertex be y, as shown in Figure 60. But then $V[y].d < V[i].d$, which is a contradiction to the choice of i. □

We used (B) to prove (A) and (A) to prove (B). To make sure we did not create a circular argument, we parametrize the two invariants with the number k of vertices that are

marked and thus belong to the currently constructed portion of the shortest path tree. To prove (A_k) we need (B_k) and to prove (B_k) we need (A_{k-1}). Think of the two invariants as two recursive functions, and for each pair of calls, the parameter decreases by one and thus eventually becomes zero, which is when the argument arrives at the base case.

All-pairs shortest paths. We can run Dijkstra's algorithm n times, once for each vertex as the source, and thus get the distance between every pair of vertices. The running time is $O(n^2 \log n + nm)$ which, for dense graphs, is the same as $O(n^3)$. Cubic running time can be achieved with a much simpler algorithm using the adjacency matrix to store distances. The idea is to iterate n times, and after the k-th iteration, the computed distance between vertices i and j is the length of the shortest path from i to j that, other than i and j, contains only vertices of index k or less.

```
for k = 1 to n do
  for i = 1 to n do
    for j = 1 to n do
      A[i,j] = min{A[i,j], A[i,k] + A[k,j]}
    endfor
  endfor
endfor.
```

The only information needed to update $A[i, j]$ during the k-th iteration of the outer for-loop are its old value and values in the k-th row and the k-th column of the prior adjacency matrix. This row remains unchanged in this iteration and so does this column. We therefore do not have to use two arrays, writing the new values right into the old matrix. We illustrate the algorithm by showing the adjacency, or distance matrix before the algorithm in Figure 61 and after one iteration in Figure 62.

	s	*a*	*b*	*c*	*d*	*e*	*f*	*g*
s	0	5	10	4	5			
a	5	0	4			5	10	
b	10	4	0					
c	4			0	4			
d	5			4	0			10
e		5				0	6	
f		10				6	0	
g					10			0

Figure 61: Adjacency, or distance matrix of the graph in Figure 57. All blank entries store ∞.

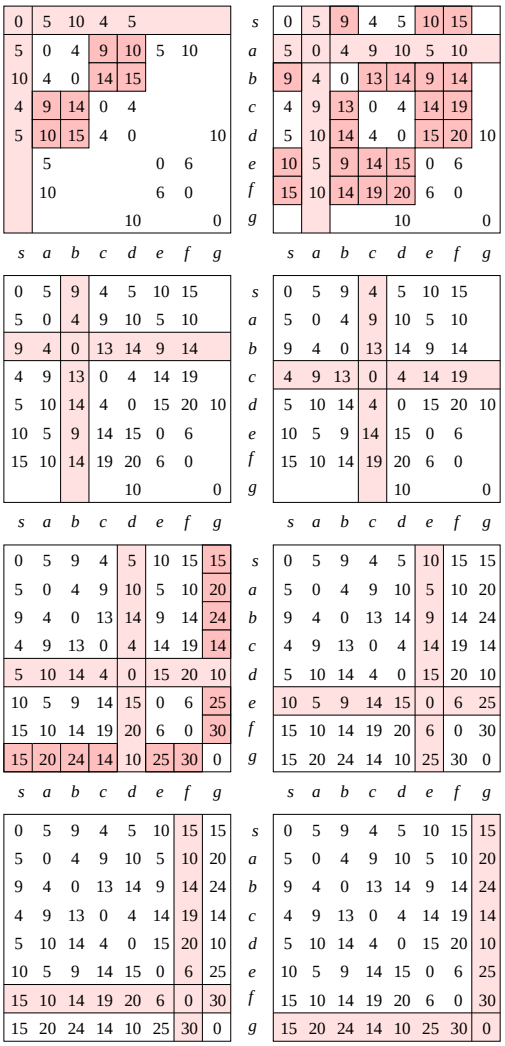

Figure 62: Matrix after each iteration. The k-th row and colum are shaded and the new, improved distances are high-lighted.

The algorithm works for weighted undirected as well as for weighted directed graphs. Its correctness is easily verified inductively. The running time is $O(n^3)$.

15 Minimum Spanning Trees

When a graph is connected, we may ask how many edges we can delete before it stops being connected. Depending on the edges we remove, this may happen sooner or later. The slowest strategy is to remove edges until the graph becomes a tree. Here we study the somewhat more difficult problem of removing edges with a maximum total weight. The remaining graph is then a tree with minimum total weight. Applications that motivate this question can be found in life support systems modeled as graphs or networks, such as telephone, power supply, and sewer systems.

Free trees. An undirected graph (U, T) is a *free tree* if it is connected and contains no cycle. We could impose a hierarchy by declaring any one vertex as the root and thus obtain a *rooted tree*. Here, we have no use for a hierarchical organization and exclusively deal with free trees. The

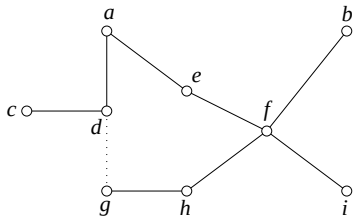

Figure 63: Adding the edge dg to the tree creates a single cycle with vertices d, g, h, f, e, a.

number of edges of a free tree is always one less than the number of vertices. Whenever we add a new edge (connecting two old vertices) we create exactly one cycle. This cycle can be destroyed by deleting any one of its edges, and we get a new free tree, as in Figure 63. Let (V, E) be a connected and undirected graph. A *subgraph* is another graph (U, T) with $U \subseteq V$ and $T \subseteq E$. It is a *spanning tree* if it is a free tree with $U = V$.

Minimum spanning trees. For the remainder of this section, we assume that we also have a weighting function, $w : E \to \mathbb{R}$. The *weight* of subgraph is then the total weight of its edges, $w(T) = \sum_{e \in T} w(e)$. A *minimum spanning tree*, or *MST* of G is a spanning tree that minimizes the weight. The definitions are illustrated in Figure 64 which shows a graph of solid edges with a minimum spanning tree of bold edges. A generic algorithm for constructing an MST grows a tree by adding more and

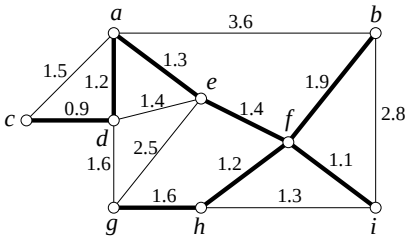

Figure 64: The bold edges form a spanning tree of weight $0.9 + 1.2 + 1.3 + 1.4 + 1.1 + 1.2 + 1.6 + 1.9 = 10.6$.

more edges. Let $A \subseteq E$ be a subset of some MST of a connected graph (V, E). An edge $uv \in E - A$ is *safe for* A if $A \cup \{uv\}$ is also subset of some MST. The generic algorithm adds safe edges until it arrives at an MST.

$A = \emptyset$;
`while` (V, A) is not a spanning tree `do`
  find a safe edge uv; $A = A \cup \{uv\}$
`endwhile`.

As long as A is a proper subset of an MST there are safe edges. Specifically, if (V, T) is an MST and $A \subseteq T$ then all edges in $T - A$ are safe for A. The algorithm will therefore succeed in constructing an MST. The only thing that is not yet clear is how to find safe edges quickly.

Cuts. To develop a mechanism for identifying safe edges, we define a *cut*, which is a partition of the vertex set into two complementary sets, $V = W \dot\cup (V - W)$. It is *crossed* by an edge $uv \in E$ if $u \in W$ and $v \in V - W$, and it *respects* an edge set A if A contains no crossing edge. The definitions are illustrated in Figure 65.

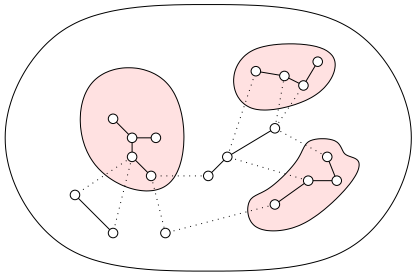

Figure 65: The vertices inside and outside the shaded regions form a cut that respects the collection of solid edges. The dotted edges cross the cut.

CUT LEMMA. Let A be subset of an MST and consider a cut $W \dot\cup (V - W)$ that respects A. If uv is a crossing edge with minimum weight then uv is safe for A.

PROOF. Consider a minimum spanning tree (V, T) with $A \subseteq T$. If $uv \in T$ then we are done. Otherwise, let $T' = T \cup \{uv\}$. Because T is a tree, there is a unique path from u to v in T. We have $u \in W$ and $v \in V - W$, so the path switches at least once between the two sets. Suppose it switches along xy, as in Figure 66. Edge xy

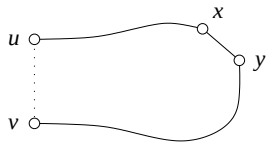

Figure 66: Adding uv creates a cycle and deleting xy destroys the cycle.

crosses the cut, and since A contains no crossing edges we have $xy \notin A$. Because uv has minimum weight among crossing edges we have $w(uv) \leq w(xy)$. Define $T'' = T' - \{xy\}$. Then (V, T'') is a spanning tree and because

$$w(T'') \;=\; w(T) - w(xy) + w(uv) \;\leq\; w(T)$$

it is a minimum spanning tree. The claim follows because $A \cup \{uv\} \subseteq T''$. □

A typical application of the Cut Lemma takes a component of (V, A) and defines W as the set of vertices of that component. The complementary set $V - W$ contains all other vertices, and crossing edges connect the component with its complement.

Prim's algorithm. Prim's algorithm chooses safe edges to grow the tree as a single component from an arbitrary first vertex s. Similar to Dijkstra's algorithm, the vertices that do not yet belong to the tree are stored in a priority queue. For each vertex i outside the tree, we define its priority $V[i].d$ equal to the minimum weight of any edge that connects i to a vertex in the tree. If there is no such edge then $V[i].d = \infty$. In addition to the priority, we store the index of the other endpoint of the minimum weight edge. We first initialize this information.

```
V[s].d = 0; V[s].π = −1; INSERT(s);
forall vertices i ≠ s do
  V[i].d = ∞; INSERT(i)
endfor.
```

The main algorithm expands the tree by one edge at a time. It uses marks to distinguish vertices in the tree from vertices outside the tree.

```
while priority queue is non-empty do
  i = EXTRACTMIN; mark i;
  forall neighbors j of i do
    if j is unmarked and w(ij) < V[j].d then
      V[j].d = w(ij); V[j].π = i
    endif
  endfor
endwhile.
```

After running the algorithm, the MST can be recovered from the π-fields of the vertices. The algorithm together with its initialization phase performs $n = |V|$ insertions into the priority queue, n extractmin operations, and at most $m = |E|$ decreasekey operations. Using the Fibonacci heap implementation, we get a running time of $O(n \log n + m)$, which is the same as for constructing the shortest-path tree with Dijkstra's algorithm.

Kruskal's algorithm. Kruskal's algorithm is another implementation of the generic algorithm. It adds edges in a sequence of non-decreasing weight. At any moment, the chosen edges form a collection of trees. These trees merge to form larger and fewer trees, until they eventually combine into a single tree. The algorithm uses a priority queue for the edges and a set system for the vertices. In this context, the term 'system' is just another word for 'set', but we will use it exclusively for sets whose elements are themselves sets. Implementations of the set system will be discussed in the next lecture. Initially, $A = \emptyset$, the priority queue contains all edges, and the system contains a singleton set for each vertex, $C = \{\{u\} \mid u \in V\}$. The algorithm finds an edge with minimum weight that connects two components defined by A. We set W equal to the vertex set of one component and use the Cut Lemma to show that this edge is safe for A. The edge is added to A and the process is repeated. The algorithm halts when only one tree is left, which is the case when A contains $n - 1 = |V| - 1$ edges.

```
A = ∅;
while |A| < n − 1 do
  uv = EXTRACTMIN;
  find P, Q ∈ C with u ∈ P and v ∈ Q;
  if P ≠ Q then
    A = A ∪ {uv}; merge P and Q
  endif
endwhile.
```

The running time is $O(m \log m)$ for the priority queue operations plus some time for maintaining C. There are two operations for the set system, namely finding the set that contains a given element, and merging two sets into one.

An example. We illustrate Kruskal's algorithm by applying it to the weighted graph in Figure 64. The sequence of edges sorted by weight is cd, fi, fh, ad, ae, hi, de, ef, ac, gh, dg, bf, eg, bi, ab. The evolution of the set system

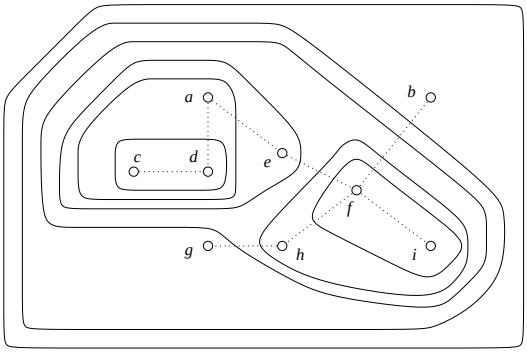

Figure 67: Eight union operations merge the nine singleton sets into one set.

is illustrated in Figure 67, and the MST computed with Kruskal's algorithm and indicated with dotted edges is the same as in Figure 64. The edges cd, fi, fh, ad, ae are all added to the tree. The next two edge, hi and de, are not added because they each have both endpoints in the same component, and adding either edge would create a cycle. Edge ef is added to the tree giving rise to a set in the system that contains all vertices other than g and b. Edge ac is not added, gh is added, dg is not, and finally bf is added to the tree. At this moment the system consists of a single set that contains all vertices of the graph.

As suggested by Figure 67, the evolution of the construction can be interpreted as a hierarchical clustering of the vertices. The specific method that corresponds to the evolution created by Kruskal's algorithm is referred to as single-linkage clustering.

16 Union-Find

In this lecture, we present two data structures for the disjoint set system problem we encountered in the implementation of Kruskal's algorithm for minimum spanning trees. An interesting feature of the problem is that m operations can be executed in a time that is only ever so slightly more than linear in m.

Abstract data type. A *disjoint set system* is an abstract data type that represents a partition C of a set $[n] = \{1, 2, \ldots, n\}$. In other words, C is a set of pairwise disjoint subsets of $[n]$ such that the union of all sets in C is $[n]$. The data type supports

`set` FIND(i): `return` $P \in C$ with $i \in P$;
`void` UNION(P, Q) : $C = C - \{P, Q\} \cup \{P \cup Q\}$.

In most applications, the sets themselves are irrelevant, and it is only important to know when two elements belong to the same set and when they belong to different sets in the system. For example, Kruskal's algorithm executes the operations only in the following sequence:

$P =$ FIND(i); $Q =$ FIND(j);
`if` $P \neq Q$ `then` UNION(P, Q) `endif`.

This is similar to many everyday situations where it is usually not important to know what it is as long as we recognize when two are the same and when they are different.

Linked lists. We construct a fairly simple and reasonably efficient first solution using linked lists for the sets. We use a table of length n, and for each $i \in [n]$, we store the name of the set that contains i. Furthermore, we link the elements of the same set and use the name of the first element as the name of the set. Figure 68 shows a sample set system and its representation. It is convenient to also store the size of the set with the first element.

To perform a UNION operation, we need to change the name for all elements in one of the two sets. To save time, we do this only for the smaller set. To merge the two lists without traversing the longer one, we insert the shorter list between the first two elements of the longer list.

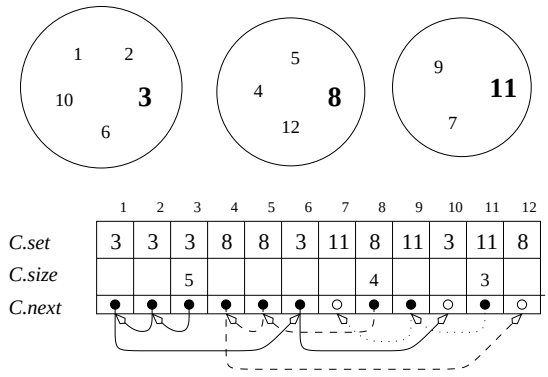

Figure 68: The system consists of three sets, each named by the bold element. Each element stores the name of its set, possibly the size of its set, and possibly a pointer to the next element in the same set.

```
void UNION(int P, Q)
  if C[P].size < C[Q].size then P ↔ Q endif;
  C[P].size = C[P].size + C[Q].size;
  second = C[P].next; C[P].next = Q; t = Q;
  while t ≠ 0 do
    C[t].set = P; u = t; t = C[t].next
  endwhile; C[u].next = second.
```

In the worst case, a single UNION operation takes time $\Theta(n)$. The amortized performance is much better because we spend time only on the elements of the smaller set.

WEIGHTED UNION LEMMA. $n - 1$ UNION operations applied to a system of n singleton sets take time $O(n \log n)$.

PROOF. For an element, i, we consider the cardinality of the set that contains it, $\sigma(i) = C[\text{FIND}(i)].size$. Each time the name of the set that contains i changes, $\sigma(i)$ at least doubles. After changing the name k times, we have $\sigma(i) \geq 2^k$ and therefore $k \leq \log_2 n$. In other words, i can be in the smaller set of a UNION operation at most $\log_2 n$ times. The claim follows because a UNION operation takes time proportional to the cardinality of the smaller set. □

Up-trees. Thinking of names as pointers, the above data structure stores each set in a tree of height one. We can use more general trees and get more efficient UNION operations at the expense of slower FIND operations. We consider a class of algorithms with the following commonalities:

- each set is a tree and the name of the set is the index of the root;
- FIND traverses a path from a node to the root;
- UNION links two trees.

It suffices to store only one pointer per node, namely the pointer to the parent. This is why these trees are called *up-trees*. It is convenient to let the root point to itself.

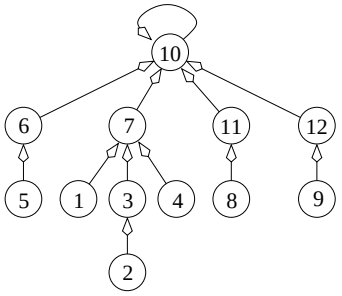

Figure 69: The UNION operations create a tree by linking the root of the first set to the root of the second set.

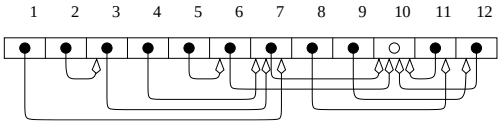

Figure 70: The table stores indices which function as pointers as well as names of elements and of sets. The white dot represents a pointer to itself.

Figure 69 shows the up-tree generated by executing the following eleven UNION operations on a system of twelve singleton sets: $2 \cup 3$, $4 \cup 7$, $2 \cup 4$, $1 \cup 2$, $4 \cup 10$, $9 \cup 12$, $12 \cup 2$, $8 \cup 11$, $8 \cup 2$, $5 \cup 6$, $6 \cup 1$. Figure 70 shows the embedding of the tree in a table. UNION takes constant time and FIND takes time proportional to the length of the path, which can be as large as $n - 1$.

Weighted union. The running time of FIND can be improved by linking smaller to larger trees. This is the ide of *weighted union* again. Assume a field $C[i].p$ for the index of the parent ($C[i].p = i$ if i is a root), and a field $C[i].$*size* for the number of elements in the tree rooted at i. We need the size field only for the roots and we need the index to the parent field everywhere except for the roots. The FIND and UNION operations can now be implemented as follows:

```
int FIND(int i)
  if C[i].p ≠ i then return FIND(C[i].p) endif;
  return i.

void UNION(int i, j)
  if C[i].size < C[j].size then i ↔ j endif;
  C[i].size = C[i].size + C[j].size; C[j].p = i.
```

The size of a subtree increases by at least a factor of 2 from a node to its parent. The depth of a node can therefore not exceed $\log_2 n$. It follows that FIND takes at most time $O(\log n)$. We formulate the result on the height for later reference.

HEIGHT LEMMA. An up-tree created from n singleton nodes by $n - 1$ weighted union operations has height at most $\log_2 n$.

Path compression. We can further improve the time for FIND operations by linking traversed nodes directly to the root. This is the idea of *path compression*. The UNION operation is implemented as before and there is only one modification in the implementation of the FIND operation:

```
int FIND(int i)
  if C[i].p ≠ i then C[i].p = FIND(C[i].p) endif;
  return C[i].p.
```

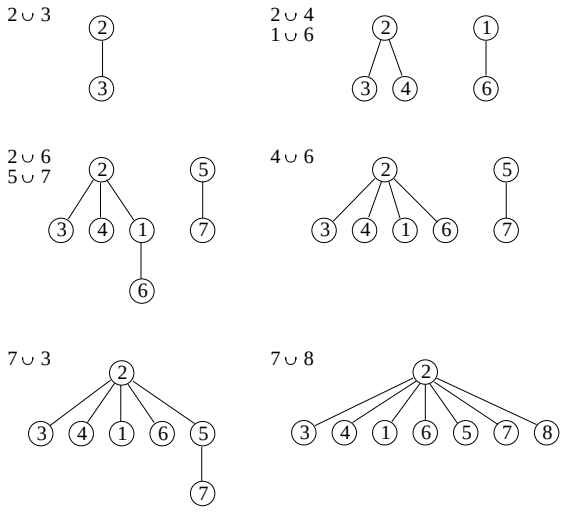

Figure 71: The operations and up-trees develop from top to bottom and within each row from left to right.

If i is not root then the recursion makes it the child of a root, which is then returned. If i is a root, it returns itself

because in this case $C[i].p = i$, by convention. Figure 71 illustrates the algorithm by executing a sequence of eight operations $i \cup j$, which is short for finding the sets that contain i and j, and performing a UNION operation if the sets are different. At the beginning, every element forms its own one-node tree. With path compression, it is difficult to imagine that long paths can develop at all.

Iterated logarithm. We will prove shortly that the iterated logarithm is an upper bound on the amortized time for a FIND operation. We begin by defining the function from its inverse. Let $F(0) = 1$ and $F(i+1) = 2^{F(i)}$. We have $F(1) = 2$, $F(2) = 2^2$, and $F(3) = 2^{2^2}$. In general, $F(i)$ is the tower of i 2s. Table 5 shows the values of F for the first six arguments. For $i \leq 3$, F is very small, but

i	0	1	2	3	4	5
F	1	2	4	16	$65,536$	$2^{65,536}$

Table 5: Values of F.

for $i = 5$ it already exceeds the number of atoms in our universe. Note that the binary logarithm of a tower of i 2s is a tower of $i-1$ 2s. The *iterated logarithm* is the number of times we can take the binary logarithm before we drop down to one or less. In other words, the iterated logarithm is the inverse of F,

$$\begin{aligned} \log^* n &= \min\{i \mid F(i) \geq n\} \\ &= \min\{i \mid \log_2 \log_2 \ldots \log_2 n \leq 1\}, \end{aligned}$$

where the binary logarithm is taken i times. As n goes to infinity, $\log^* n$ goes to infinity, but very slowly.

Levels and groups. The analysis of the path compression algorithm uses two Census Lemmas discussed shortly. Let $A_1, A_2, \ldots, A_m$ be a sequence of UNION and FIND operations, and let T be the collection of up-trees we get by executing the sequence, but *without* path compression. In other words, the FIND operations have no influence on the trees. The *level* $\lambda(\mu)$ of a node μ is its height of its subtree in T plus one.

LEVEL CENSUS LEMMA. There are at most $n/2^{\ell-1}$ nodes at level ℓ.

PROOF. We use induction to show that a node at level ℓ has a subtree of at least $2^{\ell-1}$ nodes. The claim follows because subtrees of nodes on the same level are disjoint. □

Note that if μ is a proper descendent of another node ν at some moment during the execution of the operation sequence then μ is a proper descendent of ν in T. In this case $\lambda(\mu) < \lambda(\nu)$.

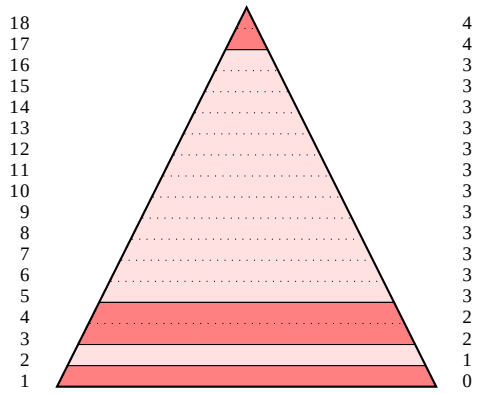

Figure 72: A schematic drawing of the tree T between the column of level numbers on the left and the column of group numbers on the right. The tree is decomposed into five groups, each a sequences of contiguous levels.

Define the *group number* of a node μ as the iterated logarithm of the level, $g(\mu) = \log^* \lambda(\mu)$. Because the level does not exceed n, we have $g(\mu) \leq \log^* n$, for every node μ in T. The definition of g decomposes an up-tree into at most $1 + \log^* n$ groups, as illustrated in Figure 72. The number of levels in group g is $F(g) - F(g-1)$, which gets large very fast. On the other hand, because levels get smaller at an exponential rate, the number of nodes in a group is not much larger than the number of nodes in the lowest level of that group.

GROUP CENSUS LEMMA. There are at most $2n/F(g)$ nodes with group number g.

PROOF. Each node with group number g has level between $F(g-1)+1$ and $F(g)$. We use the Level Census Lemma to bound their number:

$$\begin{aligned} \sum_{\ell=F(g-1)+1}^{F(g)} \frac{n}{2^{\ell-1}} &\leq \frac{n \cdot (1 + \frac{1}{2} + \frac{1}{4} + \ldots)}{2^{F(g-1)}} \\ &= \frac{2n}{F(g)}, \end{aligned}$$

as claimed. □

Analysis. The analysis is based on the interplay between the up-trees obtained with and without path compression.

The latter are constructed by the weighted union operations and eventually form a single tree, which we denote as T. The former can be obtained from the latter by the application of path compression. Note that in T, the level strictly increases from a node to its parent. Path compression preserves this property, so levels also increase when we climb a path in the actual up-trees.

We now show that any sequence of $m \geq n$ UNION and FIND operations on a ground set $[n]$ takes time at most $O(m \log^* n)$ if weighted union and path compression is used. We can focus on FIND because each UNION operation takes only constant time. For a FIND operation A_i, let X_i be the set of nodes along the traversed path. The total time for executing all FIND operations is proportional to

$$x = \sum_i |X_i|.$$

For $\mu \in X_i$, let $p_i(\mu)$ be the parent during the execution of A_i. We partition X_i into the topmost two nodes, the nodes just below boundaries between groups, and the rest:

$$\begin{aligned} Y_i &= \{\mu \in X_i \mid \mu \text{ is root or child of root}\}, \\ Z_i &= \{\mu \in X_i - Y_i \mid g(\mu) < g(p_i(\mu))\}, \\ W_i &= \{\mu \in X_i - Y_i \mid g(\mu) = g(p_i(\mu))\}. \end{aligned}$$

Clearly, $|Y_i| \leq 2$ and $|Z_i| \leq \log^* n$. It remains to bound the total size of the W_i, $w = \sum_i |W_i|$. Instead of counting, for each A_i, the nodes in W_i, we count, for each node μ, the FIND operations A_j for which $\mu \in W_j$. In other words, we count how often μ can change parent until its parent has a higher group number than μ. Each time μ changes parent, the new parent has higher level than the old parent. If follows that the number of changes is at most $F(g(\mu)) - F(g(\mu) - 1)$. The number of nodes with group number g is at most $2n/F(g)$ by the Group Census Lemma. Hence

$$\begin{aligned} w &\leq \sum_{g=0}^{\log^* n} \frac{2n}{F(g)} \cdot (F(g) - F(g-1)) \\ &\leq 2n \cdot (1 + \log^* n). \end{aligned}$$

This implies that

$$\begin{aligned} x &\leq 2m + m \log^* n + 2n(1 + \log^* n) \\ &= O(m \log^* n), \end{aligned}$$

assuming $m \geq n$. This is an upper bound on the total time it takes to execute m FIND operations. The amortized cost per FIND operation is therefore at most $O(\log^* n)$, which for all practical purposes is a constant.

Summary. We proved an upper bound on the time needed for $m \geq n$ UNION and FIND operations. The bound is more than constant per operation, although for all practical purposes it is constant. The $\log^* n$ bound can be improved to an even smaller function, usually referred to as $\alpha(n)$ or the inverse of the Ackermann function, that goes to infinity even slower than the iterated logarithm. It can also be proved that (under some mild assumptions) there is no algorithm that can execute general sequences of UNION and FIND operations in amortized time that is asymptotically less than $\alpha(n)$.