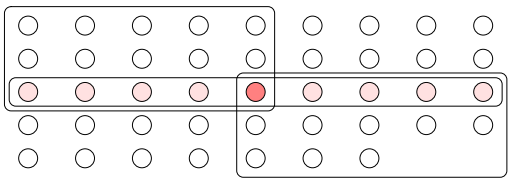

Figure 4: The 43 items are partitioned into seven groups of 5 and two groups of 4, all drawn vertically. The shaded items are the medians and the dark shaded item is the median of medians.

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:

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)?

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.

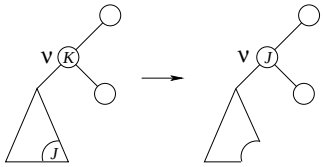

Figure 13: Store J in ν and delete the node that used to store J.

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

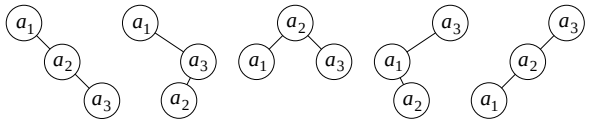

Figure 14: There are five different binary trees of three nodes. From left to right their costs are $\frac{2}{3}, \frac{5}{6}, \frac{2}{3}, \frac{7}{6}, \frac{4}{3}$. The first tree and the third tree are both optimal.

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.

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

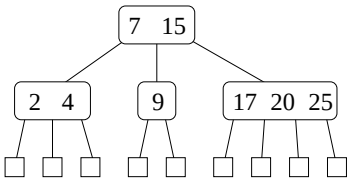

Figure 15: A 2-3-4 tree of height two. All items are stored in internal nodes.

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.

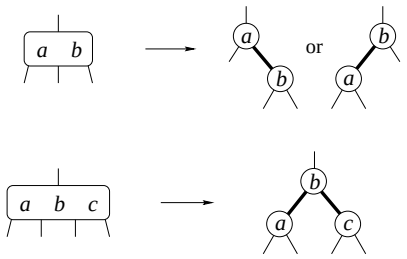

Figure 16: Transforming a 2-3-4 tree into a binary tree. Bold edges are called red and the others are called black.

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

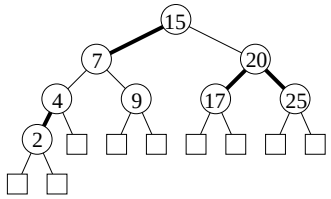

Figure 17: A red-black tree obtained from the 2-3-4 tree in Figure 15.

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:

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.

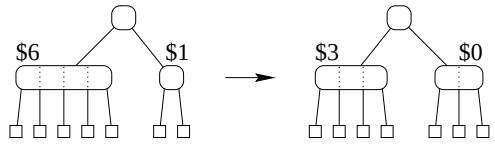

Figure 29: The overflowing node gives one child to a non-saturated sibling.

`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.

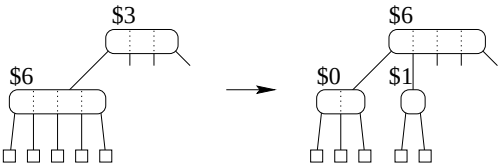

Figure 30: The overflowing node is split into two and the parent is treated recursively.

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.

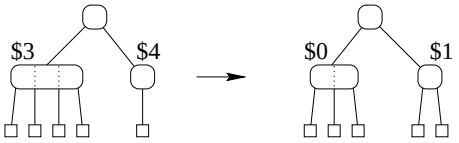

Figure 31: The underflowing node receives one child from a sibling.

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

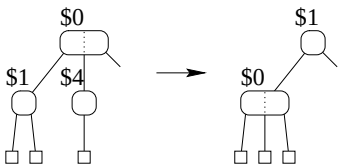

Figure 32: The underflowing node is merged with a sibling and the parent is treated recursively.

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.

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.

`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

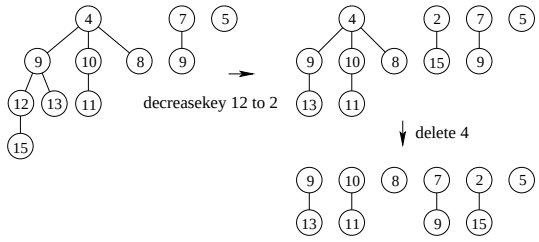

Figure 48: A Fibonacci heap initially consisting of three binomial trees modified by a decreasekey and a delete operation.

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.

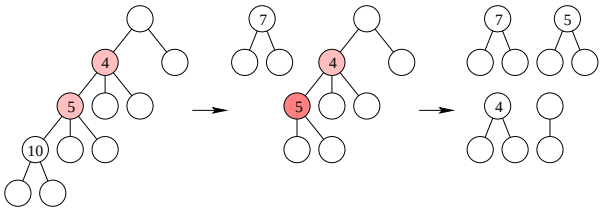

Figure 49: The effect of cascading after decreasing 10 to 7. Marked nodes are shaded.

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

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

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

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

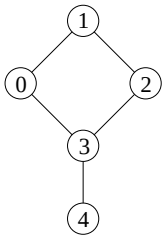

Figure 50: A simple undirected graph with vertices $0, 1, 2, 3, 4$ and edges $\{0,1\}, \{1,2\}, \{2,3\}, \{3,0\}, \{3,4\}$.

$\binom{V}{2}$ 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,

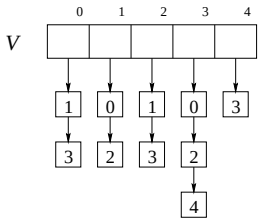

Figure 51: The adjacency list representation of the graph in Figure 50. Each edge is represented twice, once for each endpoint.

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.

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 $\lambda(\mu) < \lambda(\nu)$.

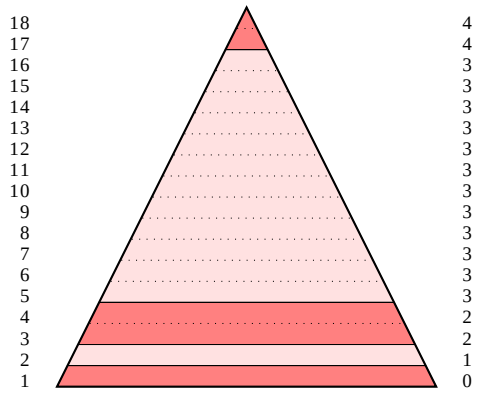

Figure 72: A schematic drawing of the tree T between the column of level numbers on the left and the column of group numbers on the right. The tree is decomposed into five groups, each a sequences of contiguous levels.

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 $\alpha(n)$.

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.

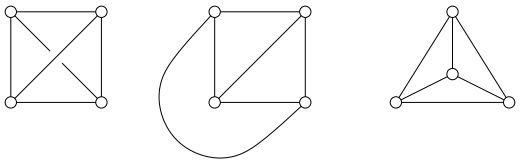

Figure 73: Three drawings of K_4, the complete graph with four vertices. From left to right: a drawing that is not an embedding, an embedding with one curved edge, a straight-line embedding.

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

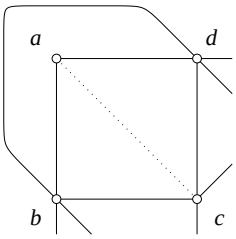

Figure 74: Drawing the edge from a to c decomposes the quadrangle into two triangles. Note that we cannot draw the edge from b to d since it already exists outside the quadrangle.

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

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 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.

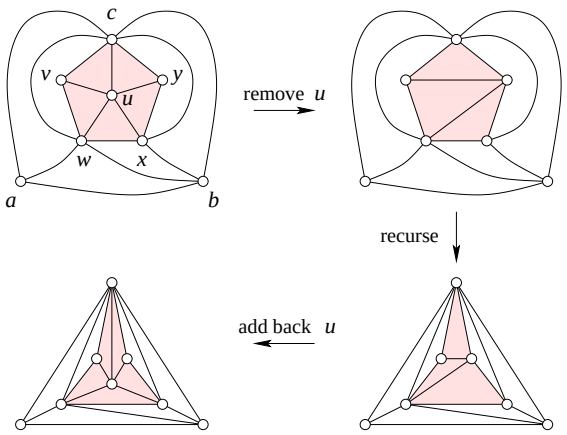

Figure 78: We fix the outer triangle, remove the degree-5 vertex, recursively construct a straight-line embedding of the rest, and finally add the vertex back.

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.

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.

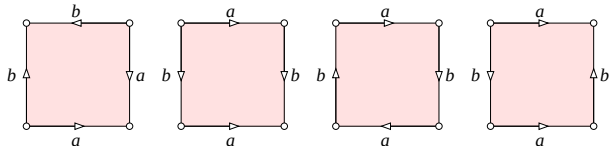

Figure 84: From left to right: the sphere, the torus, the projective plane, and the Klein bottle.

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

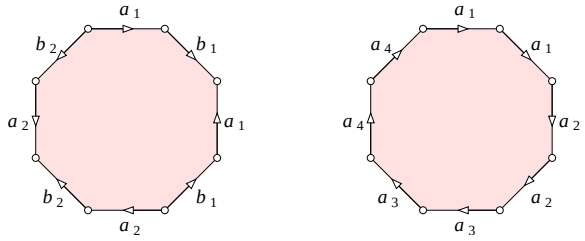

Figure 85: The polygonal schema in standard form for the double torus and the double Klein bottle.

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.

$z_1 = 3$ 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.

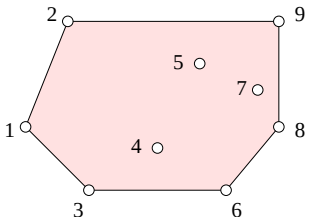

Figure 91: The convex hull of nine points, which we represent by the counterclockwise sequence of boundary vertices: 1, 3, 6, 8, 9, 2.

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.

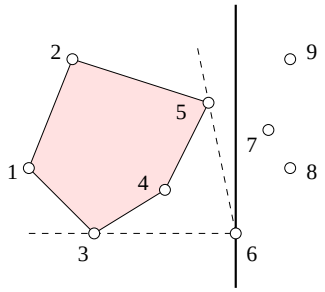

Figure 92: The vertical sweep-line passes through point 6. To add 6, we substitute 6 for the sequence of vertices on the boundary between 3 and 5.

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

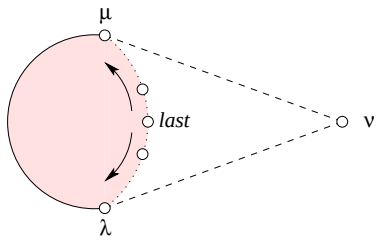

Figure 93: The upper support line passes through the first point $\mu \to pt$ that forms a left-turn from $\nu \to pt$ to $\mu \to next \to pt$.

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

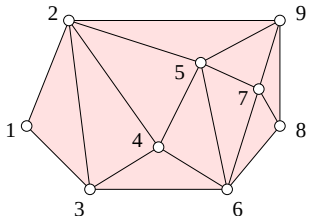

Figure 94: Triangulation constructed with the plane-sweep algorithm.

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).

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

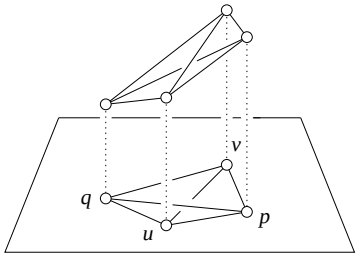

Figure 104: A flip in the plane lifts to a tetrahedron in space in which the lD edge passes below the non-lD edge.

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).

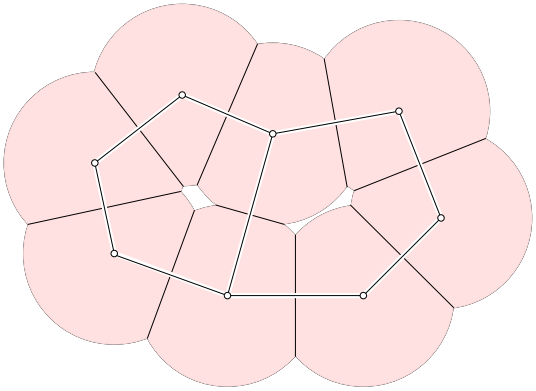

Figure 105: The Voronoi decomposition of a union of eight disks in the plane and superimposed dual alpha complex.

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,

$$S = A_1 \subset A_2 \subset \ldots \subset A_k = D,$$

VII NP-COMPLETENESS

$v \in V$ 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 $L =$

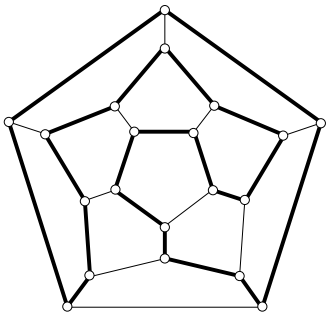

Figure 108: The edge graph of the dodecahedron and one of its hamiltonian cycles.

$\{G \mid G \text{ is hamiltonian}\}$. 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.

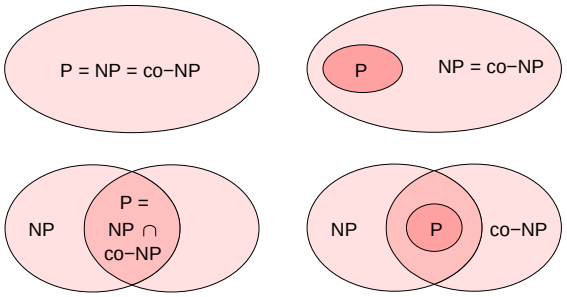

Figure 109: Four possible relations between the complexity classes P, NP, and co-NP.

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.

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.

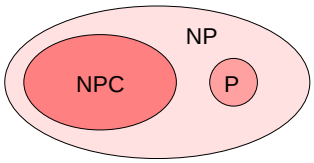

Figure 111: Possible relation between P, NPC, and NP.

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.

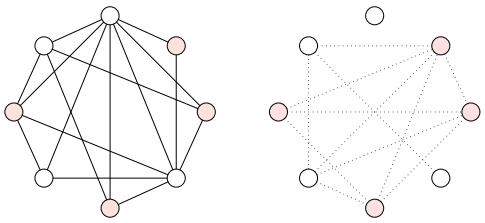

Figure 112: The four shaded vertices form an independent set in the graph on the left and a clique in the complement graph on the right.

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

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

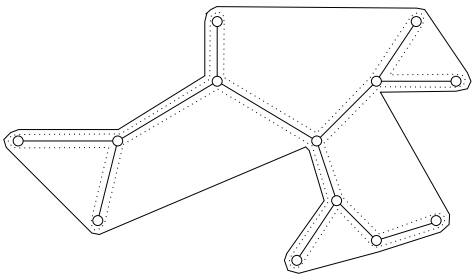

Figure 114: The solid minimum spanning tree, the dotted traversal using each edge of the tree twice, and the solid tour obtained by taking short-cuts.

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

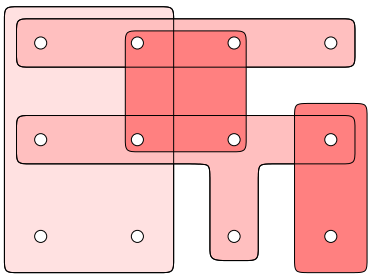

Figure 115: The set X of twelve dots can be covered with four of the five sets in the system.

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.