Seventh Homework Assignment

The purpose of this assignment is to help you prepare for the final exam. Solutions will neither be graded nor even collected.

Problem 1. ($20 = 5 + 15$ points). Consider the class of satisfiable boolean formulas in conjunctive normal form in which each clause contains two literals, 2-SAT $= \{\varphi \in \text{SAT} \mid \varphi \text{ is 2-CNF}\}$.

(a) Is 2-SAT $\in$ NP?

(b) Is there a polynomial-time algorithm for deciding whether or not a boolean formula in 2-CNF is satisfiable? If your answer is yes, then describe and analyze your algorithm. If your answer is no, then show that 2-SAT $\in$ NPC.

Problem 2. (20 points). Let A be a finite set and f a function that maps every $a \in A$ to a positive integer $f(a)$. The PARTITION problem asks whether or not there is a subset $B \subseteq A$ such that

$$\sum_{b \in B} f(b) \quad = \quad \sum_{a \in A - B} f(a).$$

We have learned that the PARTITION problem is NP-complete. Given positive integers j and k, the SUM OF SQUARES problem asks whether or not A can be partitioned into j disjoint subsets, $A = B_1 \dot\cup B_2 \dot\cup \ldots \dot\cup B_j$, such that

$$\sum_{i=1}^{j} \left(\sum_{a \in B_i} f(a) \right)^2 \quad \leq \quad k.$$

Prove that the SUM OF SQUARES problem is NP-complete.

Problem 3. ($20 = 10+10$ points). Let G be an undirected graph. A path in G is *simple* if it contains each vertex at most once. Specifying two vertices u, v and a positive integer k, the LONGEST PATH problem asks whether or not there is a simple path connecting u and v whose length is k or longer.

(a) Give a polynomial-time algorithm for the LONGEST PATH problem or show that it is NP-hard.

(b) Revisit (a) under the assumption that G is directed and acyclic.

Problem 4. ($20 = 10 + 10$ points). Let $A \subseteq 2^V$ be an abstract simplicial complex over the finite set V and let k be a positive integer.

(a) Is it NP-hard to decide whether A has k or more disjoint simplices?

(b) Is it NP-hard to decide whether A has k or fewer simplices whose union is V?

Problem 5. (20 points). Let $G = (V, E)$ be an undirected, bipartite graph and recall that there is a polynomial-time algorithm for constructing a maximum matching. We are interested in computing a minimum set of matchings such that every edge of the graph is a member of at least one of the selected matchings. Give a polynomial-time algorithm constructing an $\mathrm{O}(\log n)$ approximation for this problem.

I DESIGN TECHNIQUES

3 Prune-and-Search

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

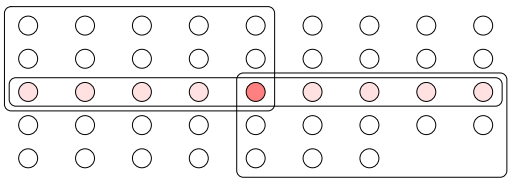

Figure 4: The 43 items are partitioned into seven groups of 5 and two groups of 4, all drawn vertically. The shaded items are the medians and the dark shaded item is the median of medians.

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

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

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

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

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

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

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

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

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

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

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

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

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

4 Dynamic Programming

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

□

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

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

First Homework Assignment

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

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

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

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

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

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

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

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

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

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

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

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

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

(b) Analyze the running time of your algorithm.

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

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

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

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

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

6 Binary Search Trees

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

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

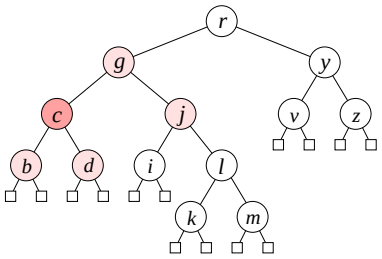

Figure 11: The parent, sibling and two children of the dark node are shaded. The internal nodes are drawn as circles while the leaves are drawn as squares.

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

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

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

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

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

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

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

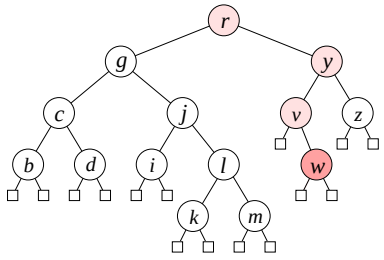

Figure 12: The shaded nodes indicate the path from the root we traverse when we insert w into the sorted tree.

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

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

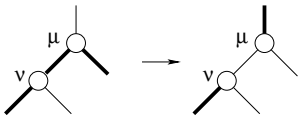

Figure 22: Promotion of μ. (The colors of the outgoing edges of ν may be the other way round).

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

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

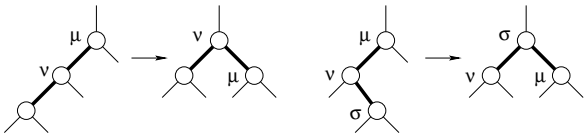

Figure 23: Right rotation of μ to the left and double right rotation of μ to the right.

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

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

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

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

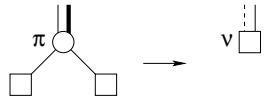

Figure 24: Deletion of node π. The dashed edge counts as two black edges when we compute the black depth.

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

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

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

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

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

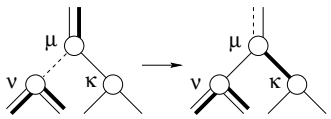

Figure 25: Demotion of μ.

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

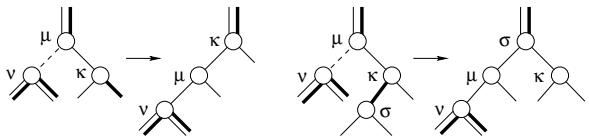

Figure 26: Left rotation of μ to the left and double left rotation of μ to the right.

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

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

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

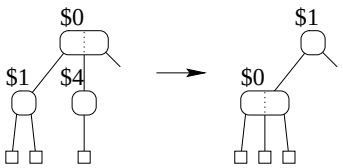

Figure 32: The underflowing node is merged with a sibling and the parent is treated recursively.

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.

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

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

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

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

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

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

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

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

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

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

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

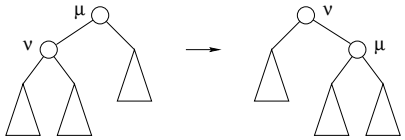

Figure 34: The size of μ decreases and that of ν increases from before to after the rotation.

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

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

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

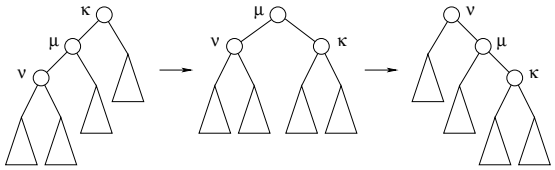

Figure 35: If in the middle tree the balance of ν is the same as the balance of μ then by the Balance Lemma the balance of κ is less than that common balance.

potential pays for the two single rotations.

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

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

III PRIORITIZING

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

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

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

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

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

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

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

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

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

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

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

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

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

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

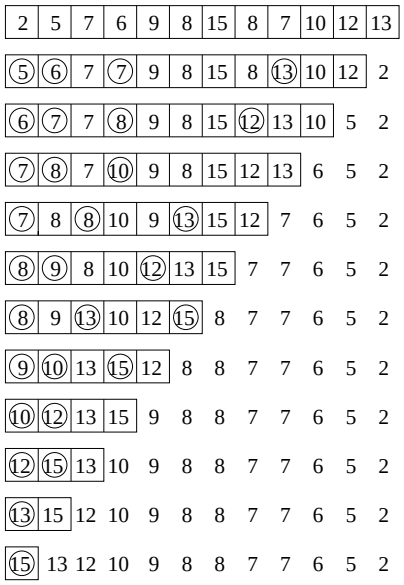

Figure 42: Each step moves the last heap element to the root and thus shrinks the heap. The circles mark the items involved in the sift-down operation.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

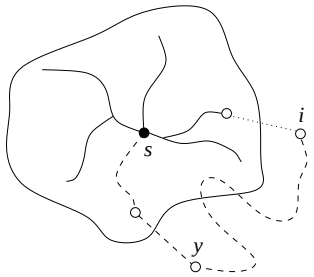

Figure 60: The vertex y is the last unmarked vertex on the hypothetically shortest, dashed path that connects i to s.

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

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

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

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

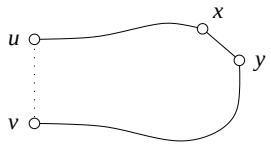

Figure 66: Adding uv creates a cycle and deleting xy destroys the cycle.

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

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

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

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

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

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

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

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

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

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

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

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

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

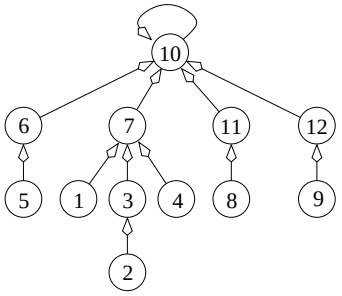

Figure 69: The UNION operations create a tree by linking the root of the first set to the root of the second set.

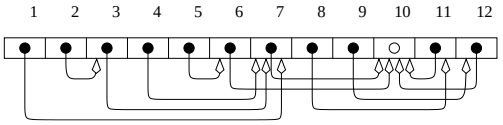

Figure 70: The table stores indices which function as pointers as well as names of elements and of sets. The white dot represents a pointer to itself.

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

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

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

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

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

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

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

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

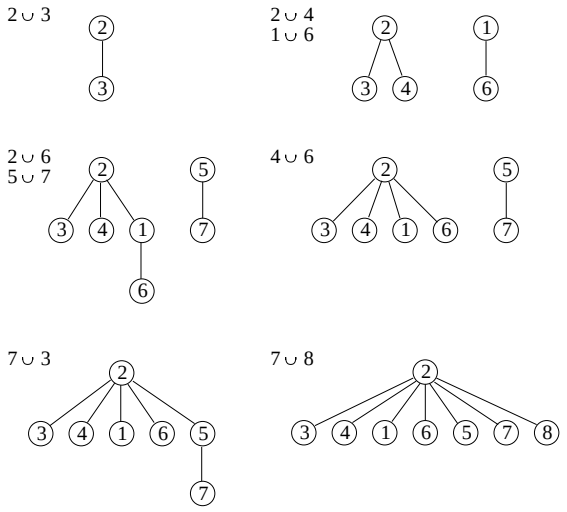

Figure 71: The operations and up-trees develop from top to bottom and within each row from left to right.

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

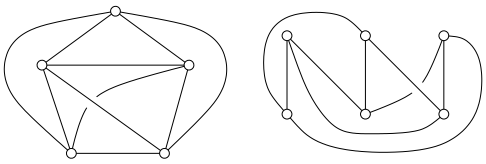

Figure 75: A drawing of K_5 on the left and of $K_{3,3}$ on the right.

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

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

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

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

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

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

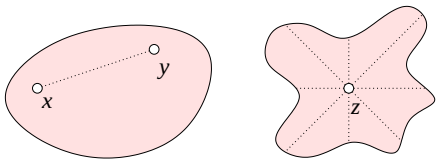

Figure 76: A convex region on the left and a non-convex star-convex region on the right.

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

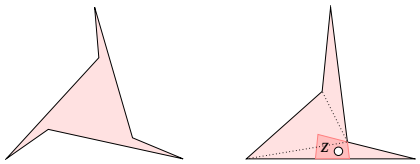

Figure 77: A non-star-convex hexagon on the left and a star-convex pentagon on the right. The dark region inside the pentagon is its kernel.

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

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

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

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

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

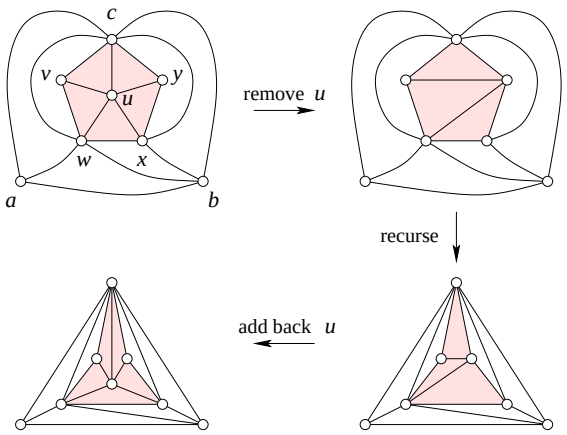

Figure 78: We fix the outer triangle, remove the degree-5 vertex, recursively construct a straight-line embedding of the rest, and finally add the vertex back.

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

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

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

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

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

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

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

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

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

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

18 Surfaces

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

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

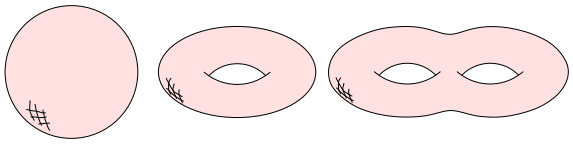

Figure 79: Three compact 2-manifolds, the sphere, the torus, and the double torus.

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

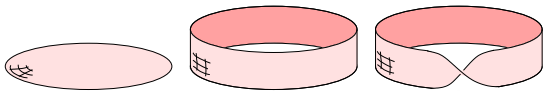

Figure 80: Three 2-manifolds with boundary, the closed disk, the cylinder, and the Möbius strip.

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

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

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

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

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

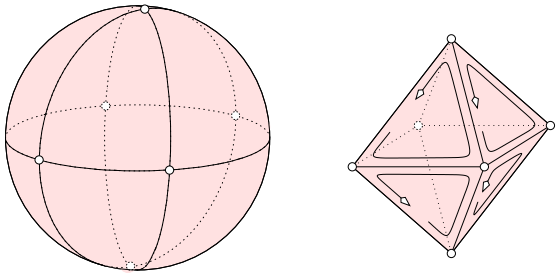

Figure 81: A triangulation of the sphere. The eight triangles are glued to form the boundary of an octahedron which is homeomorphic to the sphere.

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

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

19 Homology

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

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

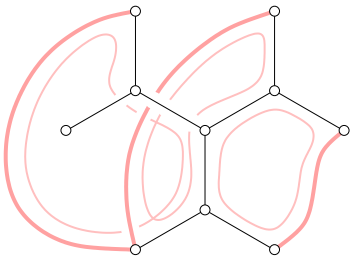

Figure 86: A tree with three additional edges defining the same number of cycles.

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

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

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

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

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

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

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

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

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

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

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

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

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

$z_1 = 3$ 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. $\chi = \sum (-1)^p \beta_p$.

VI GEOMETRIC ALGORITHMS

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

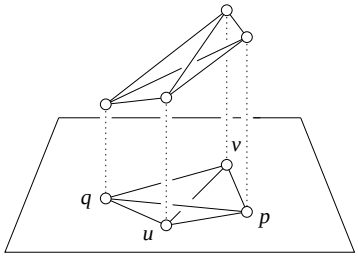

Figure 104: A flip in the plane lifts to a tetrahedron in space in which the lD edge passes below the non-lD edge.

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.

where D denotes the Delaunay triangulation of S. We call such a sequence of complexes a *filtration*. We illustrate this construction in Figure 106. The sequence of al-

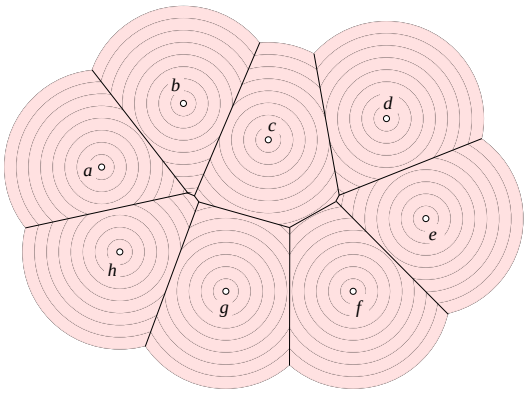

Figure 106: A finite sequence of unions of disks, all decomposed by the same Voronoi diagram.

pha complexes begins with a set of n isolated vertices, the points in S. To go from one complex to the next, we either add an edge, we add a triangle, or we add a pair consisting of a triangle with one of its edges. In Figure 106, we begin with eight vertices and get the following sequence of alpha complexes.

$$\begin{aligned}
A_1 &= \{a, b, c, d, e, f, g, h\};\\
A_2 &= A_1 \cup \{ah\};\\
A_3 &= A_2 \cup \{bc\};\\
A_4 &= A_3 \cup \{ab, ef\};\\
A_5 &= A_4 \cup \{de\};\\
A_6 &= A_5 \cup \{gh\};\\
A_7 &= A_6 \cup \{cd\};\\
A_8 &= A_7 \cup \{fg\};\\
A_9 &= A_8 \cup \{cg\}.
\end{aligned}$$

Going from A_7 to A_8, we get for the first time a 1-cycle, which bounds a hole in the embedding. In A_9, this hole is cut into two. This is the alpha complex depicted in Figure 105. We continue.

$$\begin{aligned}
A_{10} &= A_9 \cup \{cf\};\\
A_{11} &= A_{10} \cup \{abh, bh\};\\
A_{12} &= A_{11} \cup \{cde, ce\};\\
A_{13} &= A_{12} \cup \{cfg\};\\
A_{14} &= A_{13} \cup \{cef\};\\
A_{15} &= A_{14} \cup \{bch, ch\};\\
A_{16} &= A_{15} \cup \{cgh\}.
\end{aligned}$$

At this moment, we have a triangulated disk but not yet the entire Delaunay triangulation since the triangle bcd and the edge bd are still missing. Each step is generic except when we add two equally long edges to A_3.

Compatible ordering of simplices. We can represent the entire filtration of alpha complexes compactly by sorting the simplices in the order they join the growing complex. An ordering $\sigma_1, \sigma_2, \ldots, \sigma_m$ of the Delaunay simplices is *compatible* with the filtration if

1. the simplices in A_i precede the ones not in A_i for each i;
2. the faces of a simplex precede the simplex.

For example, the sequence

$$\begin{aligned}
&a, b, c, d, e, f, g, h; ah; bc; ab, ef;\\
&de; gh; cd; fg; cg; cf; bh, abh; ce,\\
&cde; cfg; cef; ch, bch; cgh; bd; bcd
\end{aligned}$$

is compatible with the filtration in Figure 106. Every alpha complex is a prefix of the compatible sequence but not necessarily the other way round. Condition 2 guarantees that every prefix is a complex, whether an alpha complex or not. We thus get a finer filtration of complexes

$$\emptyset = K_0 \subset K_1 \subset \ldots \subset K_m = D,$$

where K_i is the set of simplices from σ_1 to σ_i. To construct the compatible ordering, we just need to compute for each Delaunay simplex the radius $r_i = r(\sigma_i)$ such that $\sigma_i \in A(r)$ iff $r \geq r_i$. For a vertex, this radius is zero. For a triangle, this is the radius of the circumcircle. For

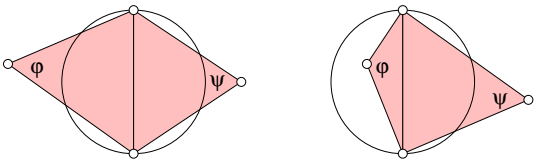

Figure 107: Left: the middle edge belongs to two acute triangles. Right: it belongs to an obtuse and an acute triangle.

an edge, we have two cases. Let φ and ψ be the angles opposite the edge σ_i inside the two incident triangles. We have $\varphi + \psi > 180°$ because of the empty circle property.

CASE 1. $\varphi < 90°$ and $\psi < 90°$. Then $r_i = r(\sigma_i)$ is half the length of the edge.

Sixth Homework Assignment

Write the solution to each problem on a single page. The deadline for handing in solutions is November 25.

Problem 1. (20 points). Let S be a set of n unit disks in the Euclidean plane, each given by its center and radius, which is one. Give an algorithm that decides whether any two of the disks in S intersect.

Problem 2. ($20 = 10 + 10$ points). Let S be a set of n points in the Euclidean plane. The *Gabriel graph* connects points $u, v \in S$ with a straight edge if

$$\|u - v\|^2 \quad \leq \quad \|u - p\|^2 + \|v - p\|^2$$

for every point p in S.

(a) Show that the Grabriel graph is a subgraph of the edge skeleton of the Delaunay triangulation.

(b) Is the Gabriel graph necessarily connected? Justify your answer.

Problem 3. ($20 = 10 + 10$ points). Consider a set of $n \geq 3$ closed disks in the Euclidean plane. The disks are allowed to touch but no two of them have an interior point in common.

(a) Show that the number of touching pairs of disks is at most $3n - 6$.

(b) Give a construction that achieves the upper bound in (a) for any $n \geq 3$.

Problem 4. ($20 = 10 + 10$ points). Let K be a triangulation of a set of $n \geq 3$ points in the plane. Let L be a line that avoids all the points.

(a) Prove that L intersects at most $2n - 4$ of the edges in K.

(b) Give a construction for which L achieves the upper bound in (a) for any $n \geq 3$.

Problem 5. (20 points). Let S be a set of n points in the Euclidean plane, consider its Delaunay triangulation and the corresponding filtration of alpha complexes,

$$S = A_1 \subset A_2 \subset \ldots \subset A_k.$$

Under what conditions is it true that A_i and A_{i+1} differ by a single simplex for every $1 \leq i \leq m - 1$?

23 Easy and Hard Problems

The theory of NP-completeness is an attempt to draw a line between tractable and intractable problems. The most important question is whether there is indeed a difference between the two, and this question is still unanswered. Typical results are therefore relative statements such as "if problem B has a polynomial-time algorithm then so does problem C" and its equivalent contra-positive "if problem C has no polynomial-time algorithm then neither has problem B". The second formulation suggests we remember hard problems C and for a new problem B we first see whether we can prove the implication. If we can then we may not want to even try to solve problem B efficiently. A good deal of formalism is necessary for a proper description of results of this kind, of which we will introduce only a modest amount.

What is a problem? An *abstract decision problem* is a function $I \to \{0,1\}$, where I is the set of problem instances and 0 and 1 are interpreted to mean FALSE and TRUE, as usual. To completely formalize the notion, we encode the problem instances in strings of zeros and ones: $I \to \{0,1\}^*$. A *concrete decision problem* is then a function $Q : \{0,1\}^* \to \{0,1\}$. Following the usual convention, we map bit-strings that do not correspond to meaningful problem instances to 0.

As an example consider the shortest-path problem. A problem instance is a graph and a pair of vertices, u and v, in the graph. A solution is a shortest path from u and v, or the length of such a path. The decision problem version specifies an integer k and asks whether or not there exists a path from u to v whose length is at most k. The theory of NP-completeness really only deals with decision problems. Although this is a loss of generality, the loss is not dramatic. For example, given an algorithm for the decision version of the shortest-path problem, we can determine the length of the shortest path by repeated decisions for different values of k. Decision problems are always easier (or at least not harder) than the corresponding optimization problems. So in order to prove that an optimization problem is hard it suffices to prove that the corresponding decision problem is hard.

Polynomial time. An algorithm *solves* a concrete decision problem Q *in time* $T(n)$ if for every instance $x \in \{0,1\}^*$ of length n the algorithm produces $Q(x)$ in time at most $T(n)$. Note that this is the worst-case notion of time-complexity. The problem Q is *polynomial-time solvable* if $T(n) = \mathrm{O}(n^k)$ for some constant k independent of n. The first important complexity class of problems is

$$\mathsf{P} = \text{set of concrete decision problems that are polynomial-time solvable.}$$

The problems $Q \in \mathsf{P}$ are called *tractable* or *easy* and the problems $Q \notin \mathsf{P}$ are called *intractable* or *hard*. Algorithms that take only polynomial time are called *efficient* and algorithms that require more than polynomial time are *inefficient*. In other words, until now in this course we only talked about efficient algorithms and about easy problems. This terminology is adapted because the rather fine grained classification of algorithms by complexity we practiced until now is not very useful in gaining insights into the rather coarse distinction between polynomial and non-polynomial.

It is convenient to recast the scenario in a formal language framework. A *language* is a set $L \subseteq \{0,1\}^*$. We can think of it as the set of problem instances, x, that have an affirmative answer, $Q(x) = 1$. An algorithm $A : \{0,1\}^* \to \{0,1\}$ *accepts* $x \in \{0,1\}^*$ if $A(x) = 1$ and it *rejects* x if $A(x) = 0$. The language *accepted* by A is the set of strings $x \in \{0,1\}^*$ with $A(x) = 1$. There is a subtle difference between accepting and *deciding* a language L. The latter means that A accepts every $x \in L$ and rejects every $x \notin L$. For example, there is an algorithm that accepts every program that halts, but there is no algorithm that decides the language of such programs. Within the formal language framework we redefine the class of polynomial-time solvable problems as

$$\begin{aligned}\mathsf{P} &= \{L \subseteq \{0,1\}^* \mid L \text{ is accepted by a polynomial-time algorithm}\} \\ &= \{L \subseteq \{0,1\}^* \mid L \text{ is decided by a polynomial-time algorithm}\}.\end{aligned}$$

Indeed, a language that can be accepted in polynomial time can also be decided in polynomial time: we keep track of the time and if too much goes by without x being accepted, we turn around and reject x. This is a non-constructive argument since we may not know the constants in the polynomial. However, we know such constants exist which suffices to show that a simulation as sketched exists.

Hamiltonian cycles. We use a specific graph problem to introduce the notion of verifying a solution to a problem, as opposed to solving it. Let $G = (V, E)$ be an undirected graph. A *hamiltonian cycle* contains every vertex

24 NP-Complete Problems

In this section, we discuss a number of NP-complete problems, with the goal to develop a feeling for what hard problems look like. Recognizing hard problems is an important aspect of a reliable judgement for the difficulty of a problem and the most promising approach to a solution. Of course, for NP-complete problems, it seems futile to work toward polynomial-time algorithms and instead we would focus on finding approximations or circumventing the problems altogether. We begin with a result on different ways to write boolean formulas.

Reduction to 3-satisfiability. We call a boolean variable or its negation a *literal*. The *conjunctive normal form* is a sequence of clauses connected by $\wedge$s, and each *clause* is a sequence of literals connected by $\vee$s. A formula is in *3-CNF* if it is in conjunctive normal form and each clause consists of three literals. It turns out that deciding the satisfiability of a boolean formula in 3-CNF is no easier than for a general boolean formula. Define 3-SAT $= \{\varphi \in \text{SAT} \mid \varphi \text{ is in 3-CNF}\}$. We prove the above claim by reducing SAT to 3-SAT.

SATISFIABILITY LEMMA. SAT $\leq_P$ 3-SAT.

PROOF. We take a boolean formula φ and transform it into 3-CNF in three steps.

`Step 1.` Think of φ as an expression and represent it as a binary tree. Each node is an operation that gets the input from its two children and forwards the output to its parent. Introduce a new variable for the output and define a new formula φ' for each node, relating the two input edges with the one output edge. Figure 110 shows the tree representation of the formula $\varphi = (x_1 \Longrightarrow x_2) \Longleftrightarrow (x_2 \vee \neg x_1)$. The new formula is

$$\begin{aligned}\varphi' \quad = \quad & (y_2 \Longleftrightarrow (x_1 \Longrightarrow x_2)) \\ & \wedge (y_3 \Longleftrightarrow (x_2 \vee \neg x_1)) \\ & \wedge (y_1 \Longleftrightarrow (y_2 \Longleftrightarrow y_3)) \wedge y_1.\end{aligned}$$

It should be clear that there is a satisfying assignment for φ iff there is one for φ'.

`Step 2.` Convert each clause into disjunctive normal form. The most mechanical way uses the truth table for each clause, as illustrated in Table 6. Each clause has at most three literals. For example, the negation of $y_2 \Longleftrightarrow (x_1 \Longrightarrow x_2)$ is equivalent to the disjunction of the conjunctions in the rightmost column. It follows that $y_2 \Longleftrightarrow (x_1 \Longrightarrow x_2)$ is equivalent to the negation of that disjunction, which by de Morgan's law is $(y_2 \vee x_1 \vee x_2) \wedge (y_2 \vee x_1 \vee \neg x_2) \wedge (y_2 \vee \neg x_1 \vee \neg x_2) \wedge (\neg y_2 \vee \neg x_1 \vee x_2)$.

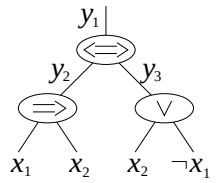

Figure 110: The tree representation of the formula φ. Incidentally, φ is a tautology, which means it is satisfied by every truth assignment. Equivalently, $\neg\varphi$ is not satisfiable.

y_2	x_1	x_2	$y_2 \Leftrightarrow (x_1 \Rightarrow x_2)$	prohibited
0	0	0	0	$\neg y_2 \wedge \neg x_1 \wedge \neg x_2$
0	0	1	0	$\neg y_2 \wedge \neg x_1 \wedge x_2$
0	1	0	1	
0	1	1	0	$\neg y_2 \wedge x_1 \wedge x_2$
1	0	0	1	
1	0	1	1	
1	1	0	0	$y_2 \wedge x_1 \wedge \neg x_2$
1	1	1	1	

Table 6: Conversion of a clause into a disjunction of conjunctions of at most three literals each.

`Step 3.` The clauses with fewer than three literals can be expanded by adding new variables. For example $a \vee b$ is expanded to $(a \vee b \vee p) \wedge (a \vee b \vee \neg p)$ and (a) is expanded to $(a \vee p \vee q) \wedge (a \vee p \vee \neg q) \wedge (a \vee \neg p \vee q) \wedge (a \vee \neg p \vee \neg q)$.

Each step takes only polynomial time. At the end, we get an equivalent formula in 3-conjunctive normal form. □

We note that clauses of length three are necessary to make the satisfiability problem hard. Indeed, there is a polynomial-time algorithm that decides the satisfiability of a formula in 2-CNF.

NP-completeness proofs. Using polynomial-time reductions, we can show fairly mechanically that problems are NP-complete, if they are. A key property is the transitivity of $\leq_P$, that is, if $L' \leq_P L_1$ and $L_1 \leq_P L_2$ then $L' \leq_P L_2$, as can be seen by composing the two polynomial-time computable functions to get a third one.

REDUCTION LEMMA. Let $L_1, L_2 \subseteq \{0,1\}^*$ and assume $L_1 \leq_P L_2$. If L_1 is NP-hard and $L_2 \in$ NP then $L_2 \in$ NPC.

A generic NP-completeness proof thus follows the steps outline below.

`Step 1.` Prove that $L_2 \in$ NP.

`Step 2.` Select a known NP-hard problem, L_1, and find a polynomial-time computable function, f, with $x \in L_1$ iff $f(x) \in L_2$.

This is what we did for $L_2 =$ 3-SAT and $L_1 =$ SAT. Therefore 3-SAT $\in$ NPC. Currently, there are thousands of problems known to be NP-complete. This is often considered evidence that P $\neq$ NP, which can be the case only if P $\cap$ NPC $= \emptyset$, as drawn in Figure 111.

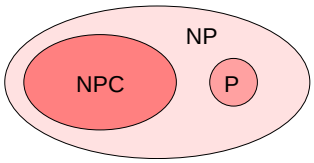

Figure 111: Possible relation between P, NPC, and NP.

Cliques and independent sets. There are many NP-complete problems on graphs. A typical such problem asks for the largest complete subgraph. Define a *clique* in an undirected graph $G = (V, E)$ as a subgraph (W, F) with $F = \binom{W}{2}$. Given G and an integer k, the CLIQUE problem asks whether or not there is a clique of k or more vertices.

CLAIM. CLIQUE $\in$ NPC.

PROOF. Given k vertices in G, we can verify in polynomial time whether or not they form a complete graph. Thus CLIQUE $\in$ NP. To prove property (2), we show that 3-SAT $\leq_P$ CLIQUE. Let φ be a boolean formula in 3-CNF consisting of k clauses. We construct a graph as follows:

(i) each clause is replaced by three vertices;

(ii) two vertices are connected by an edge if they do not belong to the same clause and they are not negations of each other.

In a satisfying truth assignment, there is at least one true literal in each clause. The true literals form a clique. Conversely, a clique of k or more vertices covers all clauses and thus implies a satisfying truth assignment. □

It is easy to decide in time $O(k^2 n^{k+2})$ whether or not a graph of n vertices has a clique of size k. If k is a constant, the running time of this algorithm is polynomial in n. For the CLIQUE problem to be NP-complete it is therefore essential that k be a variable that can be arbitrarily large. We use the NP-completeness of finding large cliques to prove the NP-completeness of large sets of pairwise non-adjacent vertices. Let $G = (V, E)$ be an undirected graph. A subset $W \subseteq V$ is *independent* if none of the vertices in W are adjacent or, equivalently, if $E \cap \binom{W}{2} = \emptyset$. Given G and an integer k, the INDEPENDENT SET problem asks whether or not there is an independent set of k or more vertices.

CLAIM. INDEPENDENT SET $\in$ NPC.

PROOF. It is easy to verify that there is an independent set of size k: just guess a subset of k vertices and verify that no two are adjacent.

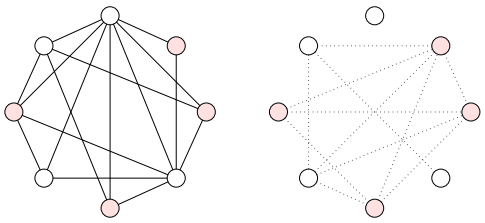

Figure 112: The four shaded vertices form an independent set in the graph on the left and a clique in the complement graph on the right.

We complete the proof by reducing the CLIQUE to the INDEPENDENT SET problem. As illustrated in Figure 112, $W \subseteq V$ is independent iff W defines a clique in the complement graph, $\overline{G} = (V, \binom{V}{2} - E)$. To prove CLIQUE $\leq_P$ INDEPENDENT SET, we transform an instance H, k of the CLIQUE problem to the instance $G = \overline{H}, k$ of the INDEPENDENT SET problem. G has an independent set of size k or larger iff H has a clique of size k or larger. □

Various NP-complete graph problems. We now describe a few NP-complete problems for graphs without proving that they are indeed NP-complete. Let $G = (V, E)$ be an undirected graph with n vertices and k a positive integer, as before. The following problems defined for G and k are NP-complete.

An *ℓ-coloring* of G is a function $\chi : V \to [\ell]$ with $\chi(u) \neq \chi(v)$ whenever u and v are adjacent. The CHROMATIC NUMBER problem asks whether or not G has an ℓ-coloring with $\ell \leq k$. The problem remains NP-complete

for fixed $k \geq 3$. For $k = 2$, the CHROMATIC NUMBER problem asks whether or not G is bipartite, for which there is a polynomial-time algorithm.

The *bandwidth* of G is the minimum ℓ such that there is a bijection $\beta : V \to [n]$ with $|\beta(u) - \beta(v)| \leq \ell$ for all adjacent vertices u and v. The BANDWIDTH problem asks whether or not the bandwidth of G is k or less. The problem arises in linear algebra, where we permute rows and columns of a matrix to move all non-zero elements of a square matrix as close to the diagonal as possible. For example, if the graph is a simple path then the bandwidth is 1, as can be seen in Figure 113. We can transform the

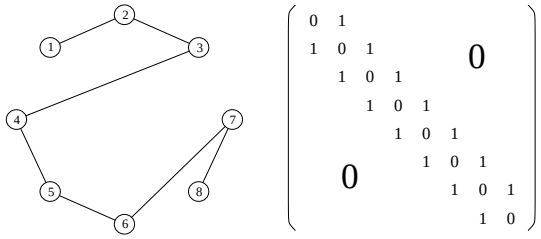

Figure 113: Simple path and adjacency matrix with rows and columns ordered along the path.

adjacency matrix of G such that all non-zero diagonals are at most the bandwidth of G away from the main diagonal.

Assume now that the graph G is complete, $E = \binom{V}{2}$, and that each edge, uv, has a positive integer weight, $w(uv)$. The TRAVELING SALESMAN problem asks whether there is a permutation $u_0, u_1, \ldots, u_{n-1}$ of the vertices such that the sum of edges connecting contiguous vertices (and the last vertex to the first) is k or less,

$$\sum_{i=0}^{n-1} w(u_i u_{i+1}) \quad \leq \quad k,$$

where indices are taken modulo n. The problem remains NP-complete if $w : E \to \{1, 2\}$ (reduction to HAMILTONIAN CYCLE problem), and also if the vertices are points in the plane and the weight of an edge is the Euclidean distance between the two endpoints.

Set systems. Simple graphs are set systems in which the sets contain only two elements. We now list a few NP-complete problems for more general set systems. Letting V be a finite set, $C \subseteq 2^V$ a set system, and k a positive integer, the following problems are NP-complete.

The PACKING problem asks whether or not C has k or more mutually disjoint sets. The problem remains NP-complete if no set in C contains more than three elements, and there is a polynomial-time algorithm if every set contains two elements. In the latter case, the set system is a graph and a maximum packing is a maximum matching.

The COVERING problem asks whether or not C has k or fewer subsets whose union is V. The problem remains NP-complete if no set in C contains more than three elements, and there is a polynomial-time algorithm if every sets contains two elements. In the latter case, the set system is a graph and the minimum cover can be constructed in polynomial time from a maximum matching.

Suppose every element $v \in V$ has a positive integer weight, $w(v)$. The PARTITION problem asks whether there is a subset $U \subseteq V$ with

$$\sum_{u \in U} w(u) \quad = \quad \sum_{v \in V - U} w(v).$$

The problem remains NP-complete if we require that U and $V - U$ have the same number of elements.