CPS 230

DESIGN AND ANALYSIS OF ALGORITHMS

Fall 2008

Instructor: **Herbert Edelsbrunner**
Teaching Assistant: **Zhiqiang Gu**

Table of Contents

1 Introduction

Meetings. We meet twice a week, on Tuesdays and Thursdays, from 1:15 to 2:30pm, in room D106 LSRC.

Communication. The course material will be delivered in the two weekly lectures. A written record of the lectures will be available on the web, usually a day after the lecture. The web also contains other information, such as homework assignments, solutions, useful links, etc. The main supporting text is

TARJAN. *Data Structures and Network Algorithms.* SIAM, 1983.

The book focuses on fundamental data structures and graph algorithms, and additional topics covered in the course can be found in the lecture notes or other texts in algorithms such as

KLEINBERG AND TARDOS. *Algorithm Design.* Pearson Education, 2006.

Examinations. There will be a final exam (covering the material of the entire semester) and a midterm (at the beginning of October), You may want to freshen up your math skills before going into this course. The weighting of exams and homework used to determine your grades is

homework	35%,
midterm	25%,
final	40%.

Homework. We have seven homeworks scheduled throughout this semester, one per main topic covered in the course. The solutions to each homework are due one and a half weeks after the assignment. More precisely, they are due at the beginning of the third lecture after the assignment. The seventh homework may help you prepare for the final exam and solutions will not be collected.

Rule 1. The solution to any one homework question must fit on a single page (together with the statement of the problem).

Rule 2. The discussion of questions and solutions before the due date is not discouraged, but you must formulate your own solution.

Rule 3. The deadline for turning in solutions is 10 minutes after the beginning of the lecture on the due date.

Overview. The main topics to be covered in this course are

I Design Techniques;
II Searching;
III Prioritizing;
IV Graph Algorithms;
V Topological Algorithms;
VI Geometric Algorithms;
VII NP-completeness.

The emphasis will be on algorithm **design** and on algorithm **analysis**. For the analysis, we frequently need basic mathematical tools. Think of analysis as the measurement of the quality of your design. Just like you use your sense of taste to check your cooking, you should get into the habit of using algorithm analysis to justify design decisions when you write an algorithm or a computer program. This is a necessary step to reach the next level in mastering the art of programming. I encourage you to implement new algorithms and to compare the experimental performance of your program with the theoretical prediction gained through analysis.

I Design Techniques

2 Divide-and-Conquer

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

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

```
void QUICKSORT(int $\ell, r$)
  if $\ell < r$ then $m = $ SPLIT($\ell, r$);
                QUICKSORT($\ell, m-1$);
                QUICKSORT($m+1, r$)
  endif.
```

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

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

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

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

```
int SPLIT(int $\ell, r$)
  $x = A[\ell]$; $i = \ell$; $j = r+1$;
  repeat repeat $i$++ until $x \le A[i]$;
         repeat $j$-- until $x \ge A[j]$;
         if $i < j$ then SWAP($i, j$) endif
  until $i \ge j$;
  SWAP($\ell, j$); return $j$.
```

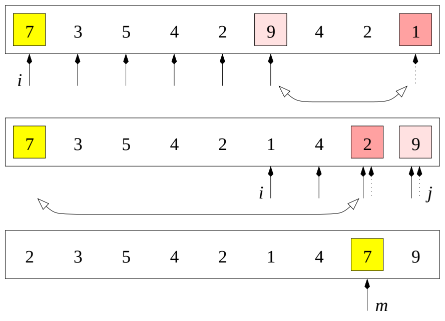


Figure 1: First, i and j stop at items 9 and 1, which are then swapped. Second, i and j cross and the pivot, 7, is swapped with item 2.

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

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

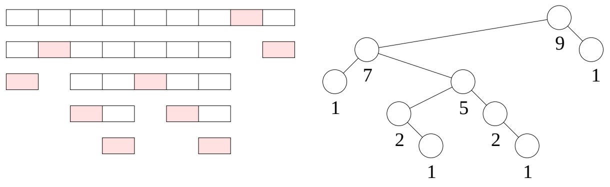


Figure 2: The total amount of time is proportional to the sum of lengths, which are the numbers of nodes in the corresponding subtrees. In the displayed case this sum is 29.

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

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

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

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

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

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

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

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

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

$$\text{Prob}[\text{RANDOM}(\ell, r) = p] = \frac{1}{r - \ell + 1}$$

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

```
int rSPLIT(int $\ell, r$)
  $p = $ RANDOM$(\ell, r)$; SWAP$(\ell, p)$;
  return SPLIT$(\ell, r)$.
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

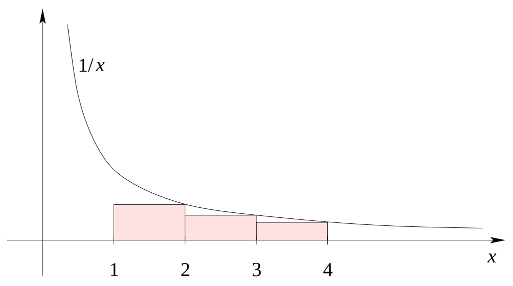


Figure 3: The areas of the rectangles are the terms in the sum, and the total rectangle area is bounded by the integral from 1 through $n+1$.

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

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

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

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

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

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

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

3 Prune-and-Search

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

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

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

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

```
int RSELECT(int $\ell, r, i$)
  $q = $ RSPLIT$(\ell, r)$; $m = q - \ell + 1$;
  if $i < m$ then return RSELECT$(\ell, q-1, i)$
    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.

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

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

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

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

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

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

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

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

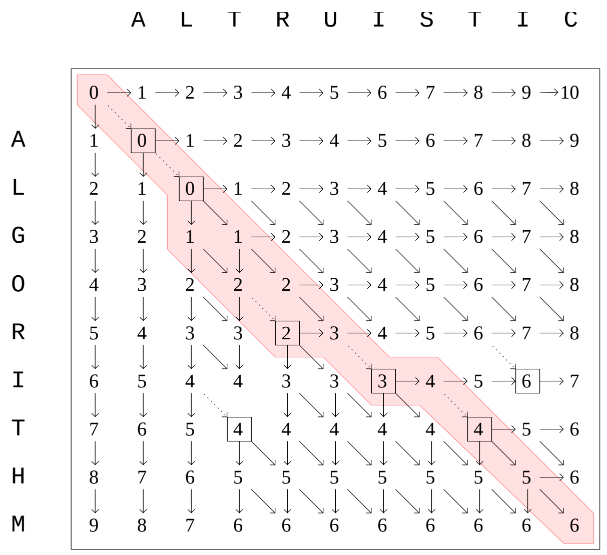


Figure 5: The table of edit distances between all prefixes of ALGORITHM and of ALTRUISTIC. The shaded area highlights the optimal edit sequences, which are paths from the upper left to the lower right corner.

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

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

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

```
void R(int $i, j$)
  if $i > 0$ or $j > 0$ then
    switch incoming arrow:
      case $\searrow$: R$(i-1, j-1)$; print$(A[i], B[j])$
      case $\downarrow$: R$(i-1, j)$; print$(A[i], \_)$
      case $\rightarrow$: R$(i, j-1)$; print$(\_, B[j])$.
    endswitch
  endif.
```

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

II SEARCHING

6 Binary Search Trees

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

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

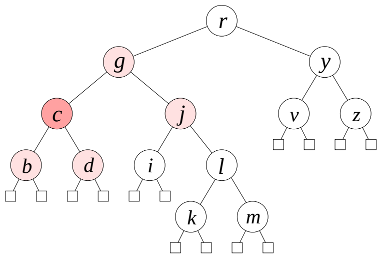


Figure 11: The parent, sibling and two children of the dark node are shaded. The internal nodes are drawn as circles while the leaves are drawn as squares.

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

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

```
    struct Node{item $info$; Node $*\ell$, $*r$};
    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 $\varrho$, item $x$)
  case $\varrho$ = NULL: return NULL;
       $x < \varrho \to info$: return SEARCH($\varrho \to \ell, x$);
       $x = \varrho \to info$: return $\varrho$;
       $x > \varrho \to info$: return SEARCH($\varrho \to r, x$)
  endcase.
```

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

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

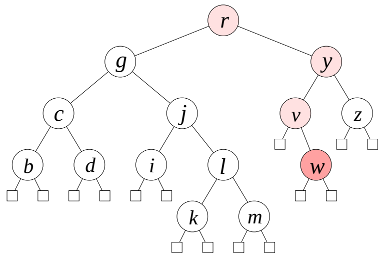


Figure 12: The shaded nodes indicate the path from the root we traverse when we insert w into the sorted tree.

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

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

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

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

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

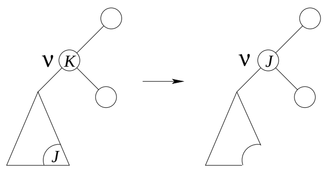


Figure 13: Store J in ν and delete the node that used to store J.

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

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

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

where δ_i is the depth of the node that stores a_i. $C(T)$ is the *weighted path length* or the *cost* of T. We study the problem of constructing a tree that minimizes the cost. To develop an example, let $n = 3$ and $p_1 = \frac{1}{2}$, $p_2 = \frac{1}{3}$, $p_3 = \frac{1}{6}$. Figure 14 shows the five binary trees with three nodes and states their costs. It can be shown that the number of different binary trees with n nodes is $\frac{1}{n+1}\binom{2n}{n}$, which is exponential in n. This is far too large to try all possibilities, so we need to look for a more efficient way to construct an optimum tree.

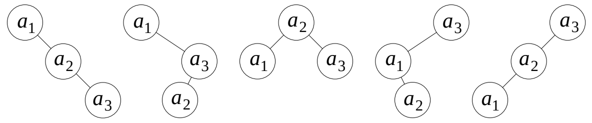


Figure 14: There are five different binary trees of three nodes. From left to right their costs are $\frac{2}{3}, \frac{5}{6}, \frac{2}{3}, \frac{7}{6}, \frac{4}{3}$. The first tree and the third tree are both optimal.

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

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

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

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

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

```
for $\ell = 2$ to $n$ do
  for $i = 1$ to $n - \ell + 1$ do
    $j = i + \ell - 1$;  $C[i, j] = \infty$;
    for $k = i$ to $j$ do
      $cost = C[i, k-1] + C[k+1, j]$
            $+ p[i, j] - p[k, k]$;
      if $cost < C[i, j]$ then
        $C[i, j] = cost$;  $R[i, j] = k$
      endif
    endfor
  endfor
endfor.
```

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

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

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

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

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

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

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

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

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

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

```
for $k = R[i, j-1]$ to $R[i+1, j]$ do
```

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

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

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

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

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

7 Red-Black Trees

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

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

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

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

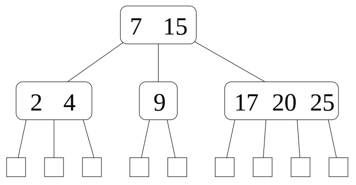


Figure 15: A 2-3-4 tree of height two. All items are stored in internal nodes.

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

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

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

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

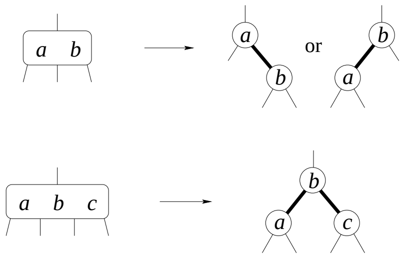


Figure 16: Transforming a 2-3-4 tree into a binary tree. Bold edges are called red and the others are called black.

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

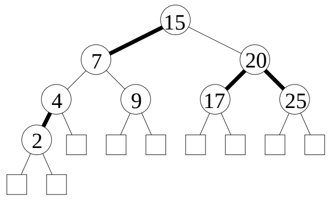


Figure 17: A red-black tree obtained from the 2-3-4 tree in Figure 15.

is shown in Figure 17.

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

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

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

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

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

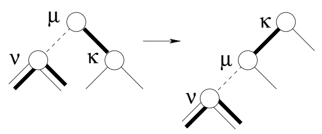


Figure 27: Left rotation of μ.

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

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

operation. The sum of amortized costs of n operations is

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

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

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

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

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

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

(2) all leaves have the same depth.

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

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

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

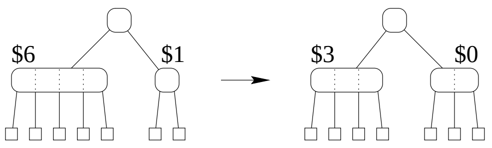


Figure 29: The overflowing node gives one child to a non-saturated sibling.

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

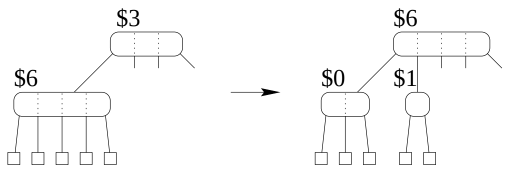


Figure 30: The overflowing node is split into two and the parent is treated recursively.

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

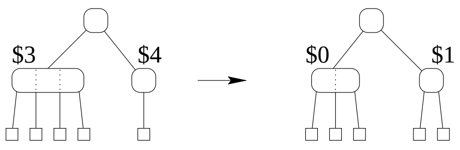


Figure 31: The underflowing node receives one child from a sibling.

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

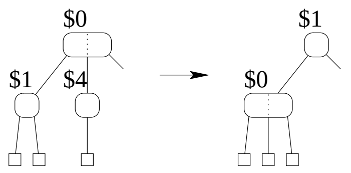


Figure 32: The underflowing node is merged with a sibling and the parent is treated recursively.

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

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

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

10 Heaps and Heapsort

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

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

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

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

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

Function EXTRACTMIN removes and returns the item with smallest rank.

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

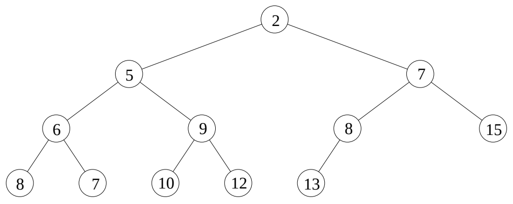
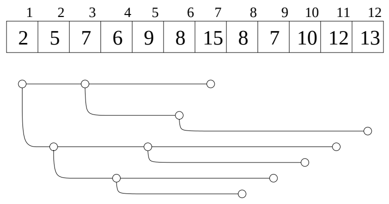


Figure 39: Ranks increase or, more precisely, do not decrease from top to bottom.

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

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



1	2	3	4	5	6	7	8	9	10	11	12
2	5	7	6	9	8	15	8	7	10	12	13

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

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

```
item FINDMIN(int $n$)
  assert $n \geq 1$; return $A[1]$.
```

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

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

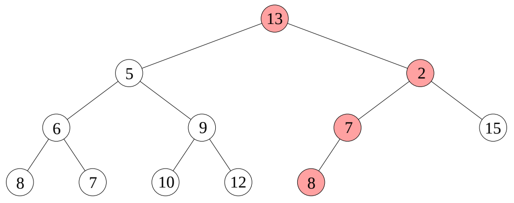


Figure 41: The root is exchanged with the smaller of its two children. The operation is repeated along a single path until the heap-order is repaired.

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

```
void SIFT-DN(int $i,\ n$)
  if $2i \leq n$ 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 $i \geq 2$ then $k = \lfloor i/2 \rfloor$;
    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 $\sum_{i=1}^{n} \log_2 i \leq n \log_2 n$. 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-

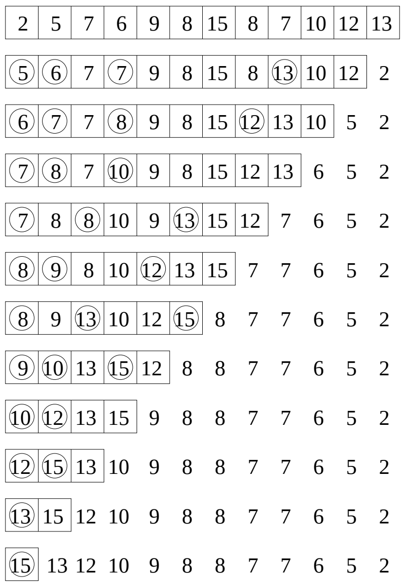


Figure 42: Each step moves the last heap element to the root and thus shrinks the heap. The circles mark the items involved in the sift-down operation.

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

11 Fibonacci Heaps

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

Binomial trees. The *binomial tree* of height h is a tree obtained from two binomial trees of height $h-1$, by linking the root of one to the other. The binomial tree of height 0 consists of a single node. Binomial trees of heights up to 4 are shown in Figure 44. Each step in the construc-

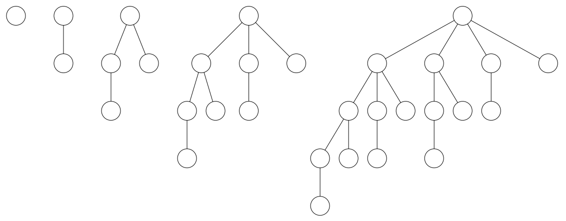


Figure 44: Binomial trees of heights 0, 1, 2, 3, 4. Each tree is obtained by linking two copies of the previous tree.

tion increases the height by one, increases the *degree* (the number of children) of the root by one, and doubles the size of the tree. It follows that a binomial tree of height h has root degree h and size 2^h. The root has the largest degree of any node in the binomial tree, which implies that every node in a binomial tree with n nodes has degree at most $\log_2 n$.

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

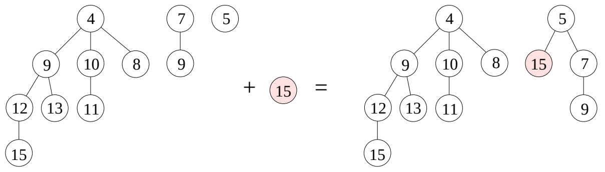


Figure 45: Adding the shaded node to a binomial heap consisting of three binomial trees.

binary notation of n. In the example, we get $1011_2 + 1_2 = 1100_2$. The new collection thus consists of two binomial trees with sizes 8 and 4. The size 8 tree is the old one, and the size 4 tree is obtained by first linking the two size 1 trees and then linking the resulting size 2 tree to the old size 2 tree. All this is illustrated in Figure 45.

Fibonacci heaps. A *Fibonacci heap* is a collection of heap-ordered trees. Ideally, we would like it to be a collection of binomial trees, but we need more flexibility. It will be important to understand how exactly the nodes of a Fibonacci heap are connected by pointers. Siblings are organized in doubly-linked cyclic lists, and each node has a pointer to its parent and a pointer to one of its children, as shown in Figure 46. Besides the pointers, each node stores

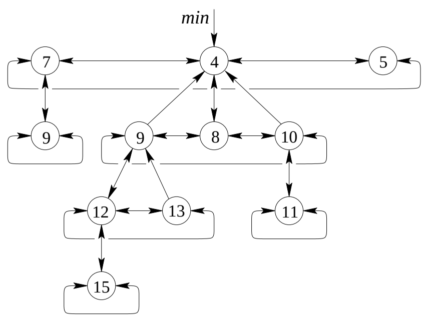


Figure 46: The Fibonacci heap representation of the first collection of heap-ordered trees in Figure 45.

a key, its degree, and a bit that can be used to mark or unmark the node. The roots of the heap-ordered trees are doubly-linked in a cycle, and there is an explicit pointer to the root that stores the item with the minimum key. Figure 47 illustrates a few basic operations we perform on a Fibonacci heap. Given two heap-ordered trees, we *link* them by making the root with the bigger key the child of the other root. To *unlink* a heap-ordered tree or subtree, we remove its root from the doubly-linked cycle. Finally, to *merge* two cycles, we cut both open and connect them at

Third Homework Assignment

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

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

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

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

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

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

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

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

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

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

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

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

IV Graph Algorithms

13 Graph Search

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

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

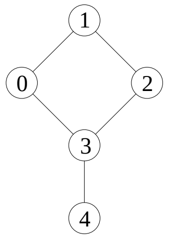


Figure 50: A simple undirected graph with vertices $0, 1, 2, 3, 4$ and edges $\{0, 1\}, \{1, 2\}, \{2, 3\}, \{3, 0\}, \{3, 4\}$.

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

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

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

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

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

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

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

which is symmetric. Irrespective of the number of edges,

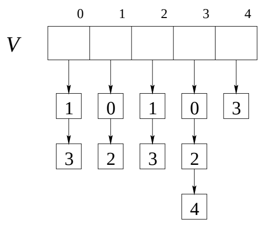


Figure 51: The adjacency list representation of the graph in Figure 50. Each edge is represented twice, once for each endpoint.

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

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

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

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

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

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

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

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

DFS forest. Figure 52 illustrates depth-first search by showing the time-stamps d and f and the pointers π indicating the predecessors in the traversal. We call an edge $\{i, j\} \in E$ a *tree edge* if $i = V[j].\pi$ or $j = V[i].\pi$ and a *back edge*, otherwise. The tree edges form the *DFS forest*

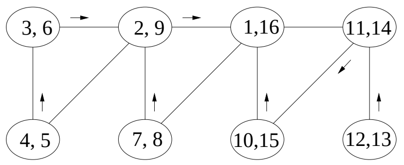


Figure 52: The traversal starts at the vertex with time-stamp 1. Each node is stamped twice, once when it is first encountered and another time when its visit is complete.

of the graph. The forest is a tree if the graph is connected and a collection of two or more trees if it is not connected. Figure 53 shows the DFS forest of the graph in Figure 52 which, in this case, consists of a single tree. The time-

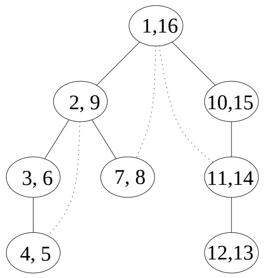


Figure 53: Tree edges are solid and back edges are dotted.

stamps d are consistent with the preorder traversal of the DFS forest. The time-stamps f are consistent with the postorder traversal. The two stamps can be used to decide, in constant time, whether two nodes in the forest live in different subtrees or one is a descendent of the other.

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

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

Directed graphs and relations. As mentioned earlier, we have a *directed graph* if all edges are directed. A directed graph is a way to think and talk about a mathematical relation. A typical problem where relations arise is scheduling. Some tasks are in a definite order while others are unrelated. An example is the scheduling of undergraduate computer science courses, as illustrated in Figure 54. Abstractly, a *relation* is a pair (V, E), where

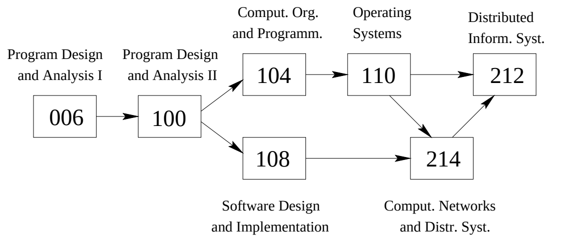

Program Design and Analysis I
Program Design and Analysis II
Comput. Org. and Programm.
Operating Systems
Distributed Inform. Syst.
006
100
104
110
212
108
214
Software Design and Implementation
Comput. Networks and Distr. Syst.

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

$V = \{0, 1, \ldots, n-1\}$ is a finite set of elements and $E \subseteq V \times V$ is a finite set of ordered pairs. Instead of

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

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

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

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

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

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

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

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

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

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

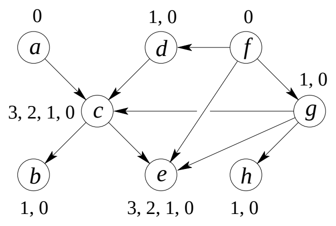


Figure 55: The numbers next to each vertex count the predecessors, which decreases during the algorithm.

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

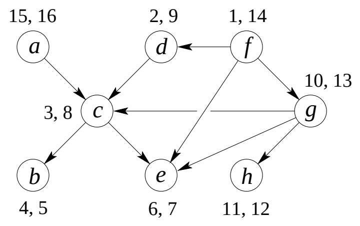


Figure 56: The numbers next to each vertex are the two time stamps applied by the depth-first search algorithm. The first number gives the time the vertex is encountered, and the second when the visit has been completed.

same digraph as Figure 55 and labels vertices with time

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.

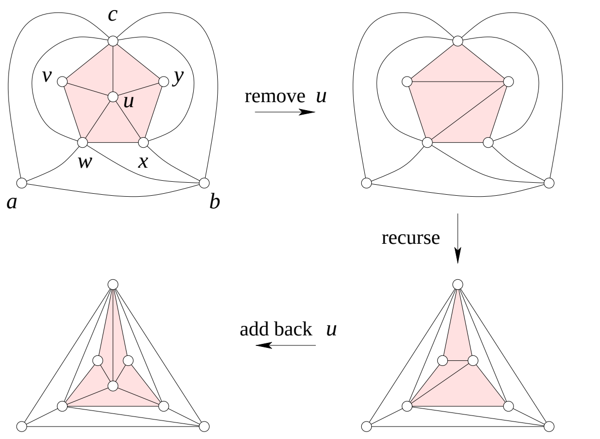


Figure 78: We fix the outer triangle, remove the degree-5 vertex, recursively construct a straight-line embedding of the rest, and finally add the vertex back.

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

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

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,

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

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

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

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.

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

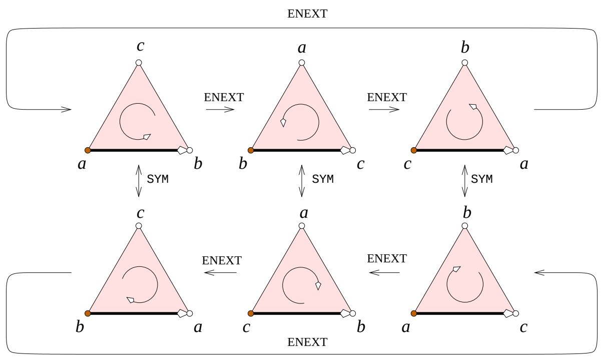


Figure 82: The symmetry group of the triangle consists of six ordered versions. Each ordered triangle has a lead vertex and a lead directed edge.

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

```
ordTri ENEXT($\mu, \iota$)
  if $\iota \leq 2$ then return $(\mu, (\iota + 1) \bmod 3)$
          else return $(\mu, (\iota + 1) \bmod 3 + 4)$
  endif.
```

```
ordTri SYM($\mu, \iota$)
  return $(\mu, (\iota + 4) \bmod 8)$.
```

To get the index of the lead vertex, we use the integer function ORG(μ, ι) and to get the pointer to the neighboring triangle, we use FNEXT(μ, ι).

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

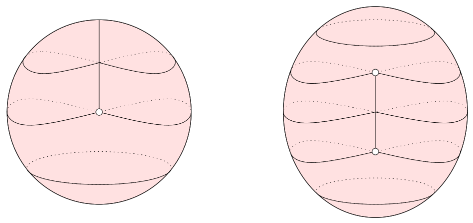


Figure 83: Two non-orientable, compact 2-manifolds, the projective plane on the left and the Klein bottle on the right.

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

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

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

Classification. For the sphere and the torus, it is easy to see how to make them out of a sheet of paper. Twisting the paper gives a non-orientable 2-manifold. Perhaps most difficult to understand is the projective plane. It is obtained by gluing each point of the sphere to its antipodal point. This way, the entire northern hemisphere is glued to the southern hemisphere. This gives the disk except that we still need to glue points of the bounding circle (the equator) in pairs, as shown in the third paper construction in Figure 84. The Klein bottle is easier to imagine as it is obtained by twisting the paper just once, same as in the construction of the Möbius strip.

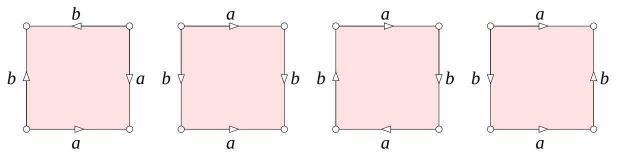


Figure 84: From left to right: the sphere, the torus, the projective plane, and the Klein bottle.

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

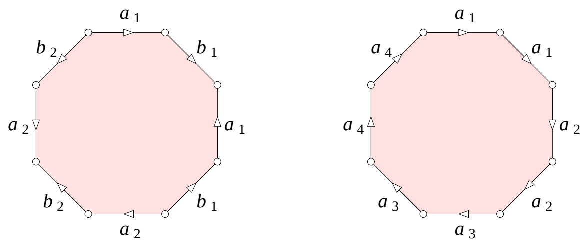


Figure 85: The polygonal schema in standard form for the double torus and the double Klein bottle.

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

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

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

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

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

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

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

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

VI GEOMETRIC ALGORITHMS

20 Plane-Sweep

Plane-sweep is an algorithmic paradigm that emerges in the study of two-dimensional geometric problems. The idea is to sweep the plane with a line and perform the computations in the sequence the data is encountered. In this section, we solve three problems with this paradigm: we construct the convex hull of a set of points, we triangulate the convex hull using the points as vertices, and we test a set of line segments for crossings.

Convex hull. Let S be a finite set of points in the plane, each given by its two coordinates. The *convex hull* of S, denoted by $\operatorname{conv} S$, is the smallest convex set that contains S. Figure 91 illustrates the definition for a set of nine points. Imagine the points as solid nails in a planar board. An intuitive construction stretches a rubber band around the nails. After letting go, the nails prevent the complete relaxation of the rubber band which will then trace the boundary of the convex hull.

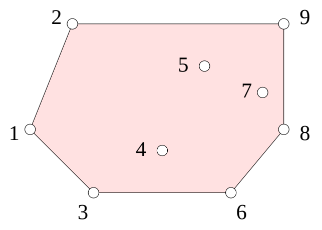


Figure 91: The convex hull of nine points, which we represent by the counterclockwise sequence of boundary vertices: 1, 3, 6, 8, 9, 2.

To construct the counterclockwise cyclic sequence of boundary vertices representing the convex hull, we sweep a vertical line from left to right over the data. At any moment in time, the points in front (to the right) of the line are untouched and the points behind (to the left) of the line have already been processed.

`Step 1.` Sort the points from left to right and relabel them in this sequence as $x_1, x_2, \ldots, x_n$.

`Step 2.` Construct a counterclockwise triangle from the first three points: $x_1x_2x_3$ or $x_1x_3x_2$.

`Step 3.` For i from 4 to n, add the next point x_i to the convex hull of the preceding points by finding the two lines that pass through x_i and support the convex hull.

The algorithm is illustrated in Figure 92, which shows the addition of the sixth point in the data set.

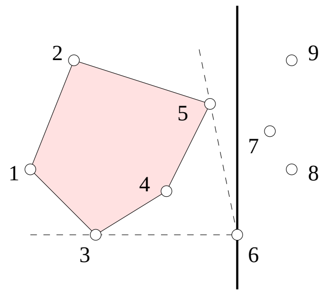


Figure 92: The vertical sweep-line passes through point 6. To add 6, we substitute 6 for the sequence of vertices on the boundary between 3 and 5.

Orientation test. A critical test needed to construct the convex hull is to determine the orientation of a sequence of three points. In other words, we need to be able to distinguish whether we make a left-turn or a right-turn as we go from the first to the middle and then the last point in the sequence. A convenient way to determine the orientation evaluates the determinant of a three-by-three matrix. More precisely, the points $a = (a_1, a_2)$, $b = (b_1, b_2)$, $c = (c_1, c_2)$ form a left-turn iff

$$\det \begin{bmatrix} 1 & a_1 & a_2 \\ 1 & b_1 & b_2 \\ 1 & c_1 & c_2 \end{bmatrix} > 0.$$

The three points form a right-turn iff the determinant is negative and they lie on a common line iff the determinant is zero.

```
boolean LEFT(Points $a, b, c$)
  return [$a_1(b_2 - c_2) + b_1(c_2 - a_2)$
              $+ c_1(a_2 - b_2) > 0$].
```

To see that this formula is correct, we may convince ourselves that it is correct for three non-collinear points, e.g. $a = (0,0)$, $b = (1,0)$, and $c = (0,1)$. Remember also that the determinant measures the area of the triangle and is therefore a continuous function that passes through zero only when the three points are collinear. Since we can continuously move every left-turn to every other left-turn without leaving the class of left-turns, it follows that the sign of the determinant is the same for all of them.

Finding support lines. We use a doubly-linked cyclic list of vertices to represent the convex hull boundary. Each node in the list contains pointers to the next and the previous nodes. In addition, we have a pointer $last$ to the last vertex added to the list. This vertex is also the rightmost in the list. We add the i-th point by connecting it to the vertices $\mu \to pt$ and $\lambda \to pt$ identified in a counterclockwise and a clockwise traversal of the cycle starting at $last$, as illustrated in Figure 93. We simplify notation by using

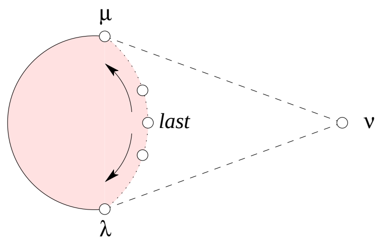


Figure 93: The upper support line passes through the first point $\mu \to pt$ that forms a left-turn from $\nu \to pt$ to $\mu \to next \to pt$.

nodes in the parameter list of the orientation test instead of the points they store.

```
$\mu = \lambda = last$; create new node with $\nu \to pt = i$;
while RIGHT$(\nu, \mu, \mu \to next)$ do
  $\mu = \mu \to next$
endwhile;
while LEFT$(\nu, \lambda, \lambda \to prev)$ do
  $\lambda = \lambda \to prev$
endwhile;
$\nu \to next = \mu$;  $\nu \to prev = \lambda$;
$\mu \to prev = \lambda \to next = \nu$;  $last = \nu$.
```

The effort to add the i-th point can be large, but if it is then we remove many previously added vertices from the list. Indeed, each iteration of the for-loop adds only one vertex to the cyclic list. We charge \$2 for the addition, one dollar for the cost of adding and the other to pay for the future deletion, if any. The extra dollars pay for all iterations of the while-loops, except for the first and the last. This implies that we spend only constant amortized time per point. After sorting the points from left to right, we can therefore construct the convex hull of n points in time O(n).

Triangulation. The same plane-sweep algorithm can be used to decompose the convex hull into triangles. All we need to change is that points and edges are never removed and a new point is connected to every point examined during the two while-loops. We define a *(geometric) triangulation* of a finite set of points S in the plane as a maximally connected straight-line embedding of a planar graph whose vertices are mapped to points in S. Figure 94 shows the triangulation of the nine points in Figure 91 constructed by the plane-sweep algorithm. A triangulation is

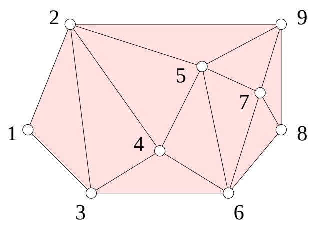


Figure 94: Triangulation constructed with the plane-sweep algorithm.

not necessarily a maximally connected planar graph since the prescribed placement of the points fixes the boundary of the outer face to be the boundary of the convex hull. Letting k be the number of edges of that boundary, we would have to add $k - 3$ more edges to get a maximally connected planar graph. It follows that the triangulation has $m = 3n - (k + 3)$ edges and $\ell = 2n - (k + 2)$ triangles.

Line segment intersection. As a third application of the plane-sweep paradigm, we consider the problem of deciding whether or not n given line segments have pairwise disjoint interiors. We allow line segments to share endpoints but we do not allow them to cross or to overlap. We may interpret this problem as deciding whether or not a straight-line drawing of a graph is an embedding. To simplify the description of the algorithm, we assume no three endpoints are collinear, so we only have to worry about crossings and not about other overlaps.

How can we decide whether or not a line segment with endpoint $u = (u_1, u_2)$ and $v = (v_1, v_2)$ crosses another line segment with endpoints $p = (p_1, p_2)$ and $q = (q_1, q_2)$? Figure 95 illustrates the question by showing the four different cases of how two line segments and the lines they span can intersect. The line segments cross iff uv intersects the line of pq and pq intersects the line of uv. This condition can be checked using the orientation test.

```
boolean CROSS(Points $u, v, p, q$)
  return [(LEFT$(u, v, p)$ xor LEFT$(u, v, q)$) and
          (LEFT$(p, q, u)$ xor LEFT$(p, q, v)$)].
```

We can use the above function to test all $\binom{n}{2}$ pairs of line segments, which takes time O(n^2).

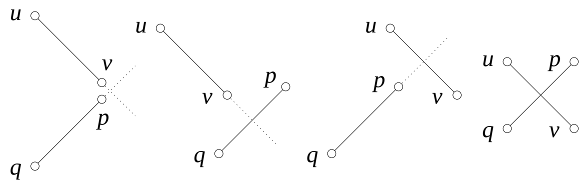


Figure 95: Three pairs of non-crossing and one pair of crossing line segments.

Plane-sweep algorithm. We obtain a faster algorithm by sweeping the plane with a vertical line from left to right, as before. To avoid special cases, we assume that no two endpoints are the same or lie on a common vertical line. During the sweep, we maintain the subset of line segments that intersect the sweep-line in the order they meet the line, as shown in Figure 96. We store this subset

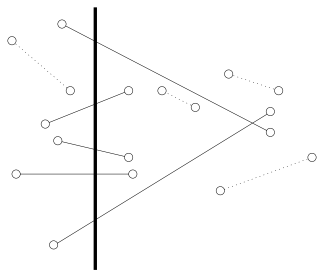


Figure 96: Five of the line segments intersect the sweep-line at its current position and two of them cross.

in a dictionary, which is updated at every endpoint. Only line segments that are adjacent in the ordering along the sweep-line are tested for crossings. Indeed, two line segments that cross are adjacent right before the sweep-line passes through the crossing, if not earlier.

```
Step 1. Sort the $2n$ endpoints from left to right and relabel them in this sequence as $x_1, x_2, \ldots, x_{2n}$. Each point still remembers the index of the other endpoint of its line segment.
Step 2. For $i$ from 1 to $2n$, process the $i$-th endpoint as follows:
    Case 2.1 $x_i$ is left endpoint of the line segment $x_i x_j$. Therefore, $i < j$. Insert $x_i x_j$ into the dictionary and let $uv$ and $pq$ be its predecessor and successor. If CROSS$(u, v, x_i, x_j)$ or CROSS$(p, q, x_i, x_j)$ then report the crossing and stop.
    Case 2.2 $x_i$ is right endpoint of the line segment $x_i x_j$. Therefore, $i > j$. Let $uv$ and $pq$ be the predecessor and the successor of $x_i x_j$. If CROSS$(u, v, p, q)$ then report the crossing and stop. Delete $x_i x_j$ from the dictionary.
```

We do an insertion into the dictionary for each left endpoint and a deletion from the dictionary for each right endpoint, both in time O($\log n$). In addition, we do at most two crossing tests per endpoint, which takes constant time. In total, the algorithm takes time O($n \log n$) to test whether a set of n line segments contains two that cross.

VII NP-Completeness

23 Easy and Hard Problems

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

What is a problem? An *abstract decision problem* is a function $I \rightarrow \{0,1\}$, 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: $I \rightarrow \{0,1\}^*$. A *concrete decision problem* is then a function $Q : \{0,1\}^* \rightarrow \{0,1\}$. 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 $T(n) = O(n^k)$ 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 $A : \{0,1\}^* \rightarrow \{0,1\}$ *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

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

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

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

$v \in V$ exactly once. The graph G is *hamiltonian* if it has a hamiltonian cycle. Figure 108 shows a hamiltonian cycle of the edge graph of a Platonic solid. How about the edge graphs of the other four Platonic solids? Define $L =$

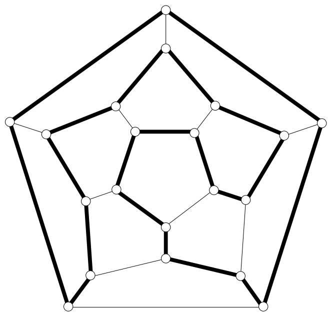


Figure 108: The edge graph of the dodecahedron and one of its hamiltonian cycles.

$\{G \mid G$ is hamiltonian$\}$. We can thus ask whether or not $L \in \mathsf{P}$, that is, whether or not there is a polynomial-time algorithm that decides whether or not a graph is hamiltonian. The answer to this question is currently not known, but there is evidence that the answer might be negative. On the other hand, suppose y is a hamiltonian cycle of G. The language $L' = \{(G, y) \mid y$ is a hamiltonian cycle of $G\}$ is certainly in P because we just need to make sure that y and G have the same number of vertices and every edge of y is also an edge of G.

Non-deterministic polynomial time. More generally, it seems easier to verify a given solution than to come up with one. In a nutshell, this is what NP-completeness is about, namely finding out whether this is indeed the case and whether the difference between accepting and verifying can be used to separate hard from easy problems.

Call $y \in \{0,1\}^*$ a *certificate*. An algorithm A *verifies* a problem instance $x \in \{0,1\}^*$ if there exists a certificate y with $A(x, y) = 1$. The language *verified* by A is the set of strings $x \in \{0,1\}^*$ verified by A. We now define a new class of problems,

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

More formally, L is in NP if for every problem instance $x \in L$ there is a certificate y whose length is bounded from above by a polynomial in the length of x such that $A(x, y) = 1$ and A runs in polynomial time. For example, deciding whether or not G is hamiltonian is in NP. The name NP is an abbreviation for **n**on-deterministic **p**olynomial time, because a non-deterministic computer can guess a certificate and then verify that certificate. In a parallel emulation, the computer would generate all possible certificates and then verify them in parallel. Generating one certificate is easy, because it only has polynomial length, but generating all of them is hard, because there are exponentially many strings of polynomial length.

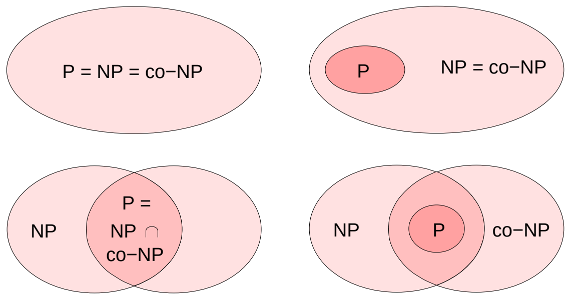


Figure 109: Four possible relations between the complexity classes P, NP, and $\mathsf{co\text{-}NP}$.

Non-deterministic machine are at least as powerful as deterministic machines. It follows that every problem in P is also in NP, $\mathsf{P} \subseteq \mathsf{NP}$. Define

$$\mathsf{co\text{-}NP} \quad = \quad \{L \mid \overline{L} = \{x \notin L\} \in \mathsf{NP}\},$$

which is the class of languages whose complement can be verified in non-deterministic polynomial time. It is not known whether or not $\mathsf{NP} = \mathsf{co\text{-}NP}$. For example, it seems easy to verify that a graph is hamiltonian but it seems hard to verify that a graph is not hamiltonian. We said earlier that if $L \in \mathsf{P}$ then $\overline{L} \in \mathsf{P}$. Therefore, $\mathsf{P} \subseteq \mathsf{co\text{-}NP}$. Hence, only the four relationships between the three complexity classes shown in Figure 109 are possible, but at this time we do not know which one is correct.

Problem reduction. We now develop the concept of reducing one problem to another, which is key in the construction of the class of NP-complete problems. The idea is to map or transform an instance of a first problem to an instance of a second problem and to map the solution to the second problem back to a solution to the first problem. For decision problems, the solutions are the same and need no transformation.

Language L_1 is *polynomial-time reducible* to language L_2, denoted $L_1 \leq_P L_2$, if there is a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that $x \in L_1$ iff $f(x) \in L_2$, for all $x \in \{0,1\}^*$. Now suppose that

L_1 is polynomial-time reducible to L_2 and that L_2 has a polynomial-time algorithm A_2 that decides L_2,

$$x \xrightarrow{f} f(x) \xrightarrow{A_2} \{0,1\}.$$

We can compose the two algorithms and obtain a polynomial-time algorithm $A_1 = A_2 \circ f$ that decides L_1. In other words, we gained an efficient algorithm for L_1 just by reducing it to L_2.

REDUCTION LEMMA. If $L_1 \leq_P L_2$ and $L_2 \in \mathsf{P}$ then $L_1 \in \mathsf{P}$.

In words, if L_1 is polynomial-time reducible to L_2 and L_2 is easy then L_1 is also easy. Conversely, if we know that L_1 is hard then we can conclude that L_2 is also hard. This motivates the following definition. A language $L \subseteq \{0,1\}^*$ is NP-*complete* if

(1) $L \in \mathsf{NP}$;
(2) $L' \leq_P L$, for every $L' \in \mathsf{NP}$.

Since *every* $L' \in \mathsf{NP}$ is polynomial-time reducible to L, *all* L' have to be easy for L to have a chance to be easy. The L' thus only provide evidence that L might indeed be hard. We say L is NP-*hard* if it satisfies (2) but not necessarily (1). The problems that satisfy (1) and (2) form the complexity class

$$\mathsf{NPC} = \{L \mid L \text{ is NP-complete}\}.$$

All these definitions would not mean much if we could not find any problems in NPC. The first step is the most difficult one. Once we have one problem in NPC we can get others using reductions.

Satisfying boolean formulas. Perhaps surprisingly, a first NP-complete problem has been found, namely the problem of satisfiability for logical expressions. A *boolean formula*, φ, consists of variables, $x_1, x_2, \ldots$, operators, $\neg, \wedge, \vee, \Longrightarrow, \ldots$, and parentheses. A *truth assignment* maps each variable to a boolean value, 0 or 1. The truth assignment *satisfies* if the formula evaluates to 1. The formula is *satisfiable* if there exists a satisfying truth assignment. Define SAT $= \{\varphi \mid \varphi \text{ is satisfiable}\}$. As an example consider the formula

$$\psi = (x_1 \Longrightarrow x_2) \Longleftrightarrow (x_2 \vee \neg x_1).$$

If we set $x_1 = x_2 = 1$ we get $(x_1 \Longrightarrow x_2) = 1$, $(x_2 \vee \neg x_1) = 1$ and therefore $\psi = 1$. It follows that $\psi \in$ SAT. In fact, all truth assignments evaluate to 1, which means that ψ is really a tautology. More generally, a boolean formula, φ, is satisfyable iff $\neg\varphi$ is not a tautology.

SATISFIABILITY THEOREM. We have SAT $\in \mathsf{NP}$ and $L' \leq_P$ SAT for every $L' \in \mathsf{NP}$.

That SAT is in the class NP is easy to prove: just guess an assignment and verify that it satisfies. However, to prove that every $L' \in \mathsf{NP}$ can be reduced to SAT in polynomial time is quite technical and we omit the proof. The main idea is to use the polynomial-time algorithm that verifies L' and to construct a boolean formula from this algorithm. To formalize this idea, we would need a formal model of a computer, a Touring machine, which is beyond the scope of this course.

24 NP-Complete Problems

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

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

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

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 $(y_2 \vee x_1 \vee x_2) \wedge (y_2 \vee x_1 \vee \neg x_2) \wedge (y_2 \vee \neg x_1 \vee \neg x_2) \wedge (\neg y_2 \vee \neg x_1 \vee x_2)$.

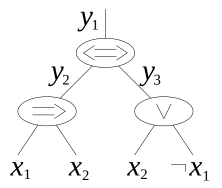


Figure 110: The tree representation of the formula φ. Incidentally, φ is a tautology, which means it is satisfied by every truth assignment. Equivalently, $\neg\varphi$ is not satisfiable.

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. ◻

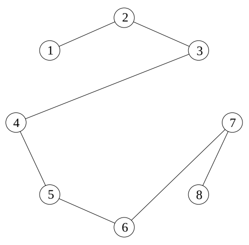
We note that clauses of length three are necessary to make the satisfiability problem hard. Indeed, there is a polynomial-time algorithm that decides the satisfiability of a formula in 2-CNF.

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.

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



$$\begin{pmatrix} 0 & 1 & & & & & & \\ 1 & 0 & 1 & & & \mathbf{0} & & \\ & 1 & 0 & 1 & & & & \\ & & 1 & 0 & 1 & & & \\ & & & 1 & 0 & 1 & & \\ & & & & 1 & 0 & 1 & \\ & \mathbf{0} & & & & 1 & 0 & 1 \\ & & & & & & 1 & 0 \end{pmatrix}$$

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

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

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

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

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

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

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

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

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

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

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

25 Approximation Algorithms

Many important problems are NP-hard and just ignoring them is not an option. There are indeed many things one can do. For problems of small size, even exponential-time algorithms can be effective and special subclasses of hard problems sometimes have polynomial-time algorithms. We consider a third coping strategy appropriate for optimization problems, which is computing almost optimal solutions in polynomial time. In case the aim is to maximize a positive cost, a $\varrho(n)$-*approximation algorithm* is one that guarantees to find a solution with cost $C \geq C^*/\varrho(n)$, where C^* is the maximum cost. For minimization problems, we would require $C \leq C^*\varrho(n)$. Note that $\varrho(n) \geq 1$ and if $\varrho(n) = 1$ then the algorithm produces optimal solutions. Ideally, ϱ is a constant but sometime even this is not achievable in polynomial time.

Vertex cover. The first problem we consider is finding the minimum set of vertices in a graph $G = (V, E)$ that covers all edges. Formally, a subset $V' \subseteq V$ is a *vertex cover* if every edge has at least one endpoint in V'. Observe that V' is a vertex cover iff $V - V'$ is an independent set. Finding a minimum vertex cover is therefore equivalent to finding a maximum independent set. Since the latter problem is NP-complete, we conclude that finding a minimum vertex cover is also NP-complete. Here is a straightforward algorithm that achieves approximation ratio $\varrho(n) = 2$, for all $n = |V|$.

```
$V' = \emptyset$;  $E' = E$;
while $E' \neq \emptyset$ do
  select an arbitrary edge $uv$ in $E'$;
  add $u$ and $v$ to $V'$;
  remove all edges incident to $u$ or $v$ from $E'$
endwhile.
```

Clearly, V' is a vertex cover. Using adjacency lists with links between the two copies of an edge, the running time is O($n + m$), where m is the number of edges. Furthermore, we have $\varrho = 2$ because every cover must pick at least one vertex of each edge uv selected by the algorithm, hence $C \leq 2C^*$. Observe that this result does not imply a constant approximation ratio for the maximum independent set problem. We have $|V - V'| = n - C \geq n - 2C^*$, which we have to compare with $n - C^*$, the size of the maximum independent set. For $C^* = \frac{n}{2}$, the approximation ratio is unbounded.

Let us contemplate the argument we used to relate C and C^*. The set of edges uv selected by the algorithm is a *matching*, that is, a subset of the edges so that no two share a vertex. The size of the minimum vertex cover is at least the size of the largest possible matching. The algorithm finds a matching and since it picks two vertices per edge, we are guaranteed at most twice as many vertices as needed. This pattern of bounding C^* by the size of another quantity (in this case the size of the largest matching) is common in the analysis of approximation algorithms. Incidentally, for bipartite graphs, the size of the largest matching is equal to the size of the smallest vertex cover. Furthermore, there is a polynomial-time algorithm for computing them.

Traveling salesman. Second, we consider the traveling salesman problem, which is formulated for a complete graph $G = (V, E)$ with a positive integer cost function $c : E \to \mathbb{Z}_+$. A *tour* in this graph is a Hamiltonian cycle and the problem is finding the tour, A, with minimum total cost, $c(A) = \sum_{uv \in A} c(uv)$. Let us first assume that the cost function satisfies the triangle inequality, $c(uw) \leq c(uv) + c(vw)$ for all $u, v, w \in V$. It can be shown that the problem of finding the shortest tour remains NP-complete even if we restrict it to weighted graphs that satisfy this inequality. We formulate an algorithm based on the observation that the cost of every tour is at least the cost of the minimum spanning tree, $C^* \geq c(T)$.

```
1  Construct the minimum spanning tree $T$ of $G$.
2  Return the preorder sequence of vertices in $T$.
```

Using Prim's algorithm for the minimum spanning tree, the running time is O(n^2). Figure 114 illustrates the algorithm. The preorder sequence is only defined if we have a root and the neighbors of each vertex are ordered, but

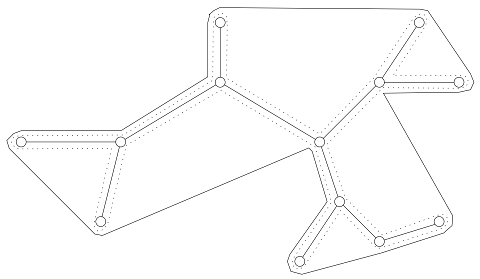


Figure 114: The solid minimum spanning tree, the dotted traversal using each edge of the tree twice, and the solid tour obtained by taking short-cuts.

we may choose both arbitrarily. The cost of the returned tour is at most twice the cost of the minimum spanning tree. To see this, consider traversing each edge of the minimum spanning tree twice, once in each direction. Whenever a vertex is visited more than once, we take the direct edge connecting the two neighbors of the second copy as a short-cut. By the triangle inequality, this substitution can only decrease the overall cost of the traversal. It follows that $C \leq 2c(T) \leq 2C^*$.

The triangle inequality is essential in finding a constant approximation. Indeed, without it we can construct instances of the problem for which finding a constant approximation is NP-hard. To see this, transform an unweighted graph $G' = (V', E')$ to the complete weighted graph $G = (V, E)$ with

$$c(uv) = \begin{cases} 1 & \text{if } uv \in E', \\ \varrho n + 1 & \text{otherwise.} \end{cases}$$

Any ϱ-approximation algorithm must return the Hamiltonian cycle of G', if there is one.

Set cover. Third, we consider the problem of covering a set X with sets chosen from a set system $\mathcal{F}$. We assume the set is the union of sets in the system, $X = \bigcup \mathcal{F}$. More precisely, we are looking for a smallest subsystem $\mathcal{F}' \subseteq \mathcal{F}$ with $X = \bigcup \mathcal{F}'$. The *cost* of this subsystem is the number of sets it contains, $|\mathcal{F}'|$. See Figure 115 for an illustration of the problem. The vertex cover problem

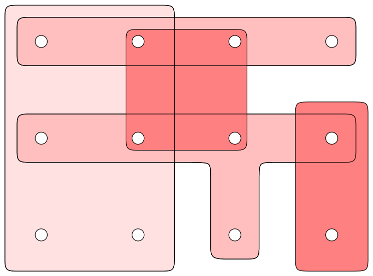


Figure 115: The set X of twelve dots can be covered with four of the five sets in the system.

is a special case: $X = E$ and $\mathcal{F}$ contains all subsets of edges incident to a common vertex. It is special because each element (edge) belongs to exactly two sets. Since we no longer have a bound on the number of sets containing a single element, it is not surprising that the algorithm for vertex covers does not extend to a constant-approximation algorithm for set covers. Instead, we consider the following greedy approach that selects, at each step, the set containing the maximum number of yet uncovered elements.

```
$\mathcal{F}' = \emptyset$; $X' = X$;
while $X' \neq \emptyset$ do
  select $S \in \mathcal{F}$ maximizing $|S \cap X'|$;
  $\mathcal{F}' = \mathcal{F}' \cup \{S\}$; $X' = X' - S$
endwhile.
```

Using a sparse matrix representation of the set system (similar to an adjacency list representation of a graph), we can run the algorithm in time proportional to the total size of the sets in the system, $n = \sum_{S \in \mathcal{F}} |S|$. We omit the details.

Analysis. More interesting than the running time is the analysis of the approximation ratio the greedy algorithm achieves. It is convenient to have short notation for the d-th harmonic number, $H_d = \sum_{i=1}^{d} \frac{1}{i}$ for $d \geq 0$. Recall that $H_d \leq 1 + \ln d$ for $d \geq 1$. Let the size of the largest set in the system be $m = \max\{|S| \mid S \in \mathcal{F}\}$.

CLAIM. The greedy method is an H_m-approximation algorithm for the set cover problem.

PROOF. For each set S selected by the algorithm, we distribute \$1 over the $|S \cap X'|$ elements covered for the first time. Let c_x be the cost allocated this way to $x \in X$. We have $|\mathcal{F}'| = \sum_{x \in X} c_x$. If x is covered the first time by the i-th selected set, S_i, then

$$c_x = \frac{1}{|S_i - (S_1 \cup \ldots \cup S_{i-1})|}.$$

We have $|\mathcal{F}'| \leq \sum_{S \in \mathcal{F}^*} \sum_{x \in S} c_x$ because the optimal cover, $\mathcal{F}^*$, contains each element x at least once. We will prove shortly that $\sum_{x \in S} c_x \leq H_{|S|}$ for every set $S \in \mathcal{F}$. It follows that

$$|\mathcal{F}'| \leq \sum_{S \in \mathcal{F}^*} H_{|S|} \leq H_m |\mathcal{F}^*|,$$

as claimed. $\square$

For $m = 3$, we get $\varrho = H_3 = \frac{11}{6}$. This implies that for graphs with vertex-degrees at most 3, the greedy algorithm guarantees a vertex cover of size at most $\frac{11}{6}$ times the optimum, which is better than the ratio 2 guaranteed by our first algorithm.

We still need to prove that the sum of costs c_x over the elements of a set S in the system is bounded from above by $H_{|S|}$. Let u_i be the number of elements in S that are not covered by the first i selected sets, $u_i = |S - (S_1 \cup \ldots \cup S_i)|$, and observe that the numbers do not increase. Let u_{k-1} be the last non-zero number in the sequence, so $|S| = u_0 \geq \ldots \geq u_{k-1} > u_k = 0$. Since $u_{i-1} - u_i$ is the number of elements in S covered the first time by S_i, we have

$$\sum_{x \in S} c_x \quad = \quad \sum_{i=1}^{k} \frac{u_{i-1} - u_i}{|S_i - (S_1 \cup \ldots \cup S_{i-1})|}.$$

We also have $u_{i-1} \leq |S_i - (S_1 \cup \ldots \cup S_{i-1})|$, for all $i \leq k$, because of the greedy choice of S_i. If this were not the case, the algorithm would have chosen S instead of S_i in the construction of $\mathcal{F}'$. The problem thus reduces to bounding the sum of ratios $\frac{u_{i-1} - u_i}{u_{i-1}}$. It is not difficult to see that this sum can be at least logarithmic in the size of S. Indeed, if we choose u_i about half the size of u_{i-1}, for all $i \geq 1$, then we have logarithmically many terms, each roughly $\frac{1}{2}$. We use a sequence of simple arithmetic manipulations to prove that this lower bound is asymptotically tight:

$$\begin{aligned} \sum_{x \in S} c_x \quad &\leq \quad \sum_{i=1}^{k} \frac{u_{i-1} - u_i}{u_{i-1}} \\ &= \quad \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{u_{i-1}}. \end{aligned}$$

We now replace the denominator by $j \leq u_{i-1}$ to form a telescoping series of harmonic numbers and get

$$\begin{aligned} \sum_{x \in S} c_x \quad &\leq \quad \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j} \\ &= \quad \sum_{i=1}^{k} \left(\sum_{j=1}^{u_{i-1}} \frac{1}{j} - \sum_{j=1}^{u_i} \frac{1}{j} \right) \\ &= \quad \sum_{i=1}^{k} (H_{u_{i-1}} - H_{u_i}). \end{aligned}$$

This is equal to $H_{u_0} - H_{u_k} = H_{|S|}$, which fills the gap left in the analysis of the greedy algorithm.

Seventh Homework Assignment

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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