9 Decision trees

The logical framework of a lot of algorithms can be described by a tree: we start from the root and in every branching point, the result of a certain "test" determines which way we continue. E.g., most sorting algorithms make sometimes comparisons between certain pairs of elements and continue the work according to the result of the comparison. We assume that the tests performed in such computations contain all necessary information about the input, i.e., when we arrive at an endpoint all that is left is to read the output from the endpoint. The complexity of the tree gives some information about the complexity of the algorithm; the depth of the tree (the number of edges in the longest path leaving the root) tells, e.g., how many tests must be performed in the worst case during the computation. We can describe, of course, every algorithm bya trivial tree of depth 1 (the test performed in the root is the computation of the end result); this algorithm scheme makes sense therefore only if we restrict the kind of tests allowed in the nodes.

We will see that decision trees not only give a graphical representation of the structure of some algorithms but are also suitable for proving lower bounds on their depth. Such a lower bound can be interpreted as saying that the problem cannot be solved (for the worst input) in fewer steps if we assume that information on the input is available only by the permissible tests (e.g. in sorting, we can only compare the given numbers with each other and cannot perform e.g. arithmetic operations on them).

9.1 Algorithms using decision trees

Consider some simple examples.

9.1.1 Finding a false coin with a one-armed scale

Given are n coins looking outwardly identical. We know that each must weigh 1g; but we also know that there is a false one among them that is lighter than the rest. We have a one-armed scale; we can measure with it the weight of an arbitrary subset of the coins. How many measurements are enough to decide which coin is false?

The solution is simple: with one measurement, we can decide about an arbitrary set of coins whether the false one is among them. If we put $\lceil n/2 \rceil$ coins on the scale then after one measurement, we have to find the false coin already only among at most $\lceil n/2 \rceil$ ones. This recursion ends in $\lceil \log_2 n \rceil$ steps.

We can characterize the algorithm by a rooted binary tree. Every vertex v corresponds to a set X_v of coins; arriving into this vertex we already know that the false coin is to be found into this set. (The root corresponds to the original set, and the endpoints to the 1-element sets.) For every branching point v, we divide the set X_v into two parts, with numbers of elements $\lceil |X_v|/2 \rceil$ and $\lfloor |X_v|/2 \rfloor$. These correspond to the children of v. Measuring the first one we learn which one contains the false coin.

9.1.2 Finding a false coin with a two-armed scale

Again, we are given n outwardly identical coins. We know that there is a false one among them that is lighter than the rest. This time we have a two-armed scale but without weights.

On this, we can find out which one of two (disjoint) sets of coins is lighter, or whether they are equal. How many measurements suffice to decide which coin is false?

Here is a solution. One measurement consists of putting the same number of coins into each pan. If one side is lighter then the false coin is in that pan. If the two sides have equal weight then the false coin is among the ones left out. It is most practical to put $\lceil n/3 \rceil$ coins into both pans; then after one measurement, the fals coin must be found only among at most $\lceil n/3 \rceil$ coins. This recursion terminates in $\lceil \log_3 n \rceil$ steps.

Since one measurement has 3 possible outcomes the algorithm can be characterized by a rooted tree in which each branching point has 3 children. Every node v corresponds to a set X_v of coins; arriving into this node we already know that the false coin is to be found in this set. (As above, the root corresponds to the original set and the endpoints to the one-element sets.) For each branching point v, we divide the set X_v into three parts, with $\lceil |X_v|/3 \rceil$, $\lceil |X_v|/3 \rceil$ and $|X_v| - 2\lceil |X_v|/3 \rceil$ elements. These correspond to the children of v. Comparing the two first ones we can find out which one of the three contains the false coin.

9.1 **Exercise** Prove that fewer measurements do not suffice in either problem 9.1.1 or problem 9.1.2. $\diamondsuit$

9.1.3 Sorting

Given are n elements that are ordered in some way (unknown for us). We know a procedure to decide the order of two elements; this is called a **comparison** and considered an elementary step. We would like to determine the complete ordering using as few comparisons as possible. Many algorithms are know for this basic problem of data processing; we treat this question only to the depth necessary for the illustration of decision trees.

Obviously, $\binom{n}{2}$ comparisons are enough: with these, we can learn about every pair of elements, wich which one in the pair is greater, and this determines the order. These comparisons are not, however, independent: often, we can infer the order of certain pairs using transitivity. Indeed, it is enough to make $\sum_{k=1}^{n} \lceil \log_2 k \rceil \sim n \log_2 n$ comparisons. Here is the simplest way to see this: suppose that we already determined the ordering of the first $n-1$ elements. Then already only the n-th element must be "inserted", which can obviously be done with $\lceil \log_2 n \rceil$ comparisons.

This algorithm, as well as any other sorting algorithm working with comparisons, can be represented by a binary tree. The root corresponds to the first comparison; depending on its result, the algorithm branches into one of the children of the root. Here, we make another comparison, etc. Every endpoint corresponds to a complete ordering.

(9.1) **Remark** In the above sorting algorithm, we only counted the comparisons. With a real program, one should also take into account the other operations, e.g. the movement of data, etc. From this point of view, the above algorithm is not good since every insertion may require the movement of all elements placed earlier and this may cause $\Omega(n^2)$ extra steps. There exist, however, sorting algorithm requiring altogether only $O(n \log n)$ steps. $\diamondsuit$

9.1.4 Convex hull

The determination of the convex hull of n planar points is as basic among the geometrical algorithms as sorting for data processing. The points are given by their coordinates:

$p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n)$. We assume, for simplicity, that the points are in **general position**, i.e. no 3 of them is on one straight line. We want to determine those indices $i_0, \ldots, i_{k-1}, i_k = i_0$ for which $p_{i_0}, \ldots, p_{i_{k-1}}, p_{i_k}$ are the vertices of the convex hull of the given point set, in this order along the convex hull (starting anticlockwise, say, from the point with the smallest abscissa).

The idea of "insertion" gives a simple algorithm here, too. Sort the elements by their x_i coordinates; this can be done in time $O(n \log n)$. Suppose that $p_1, \ldots, p_n$ are already indexed in this order. Delete the point p_n and determine the convex hull of the points $p_1, \ldots, p_{n-1}$: let this be the sequence of points $p_{j_0}, \ldots, p_{j_{m-1}}, p_{j_m}$ where $j_0 = j_m = 1$.

Now, the addition of p_n consists of deleting the arc of the polygon $p_{j_0}, \ldots, p_{j_m}$ "visible" from p_n and replacing it with the point p_n. Let us determine the first and last elements of the sequence $p_{j_0}, \ldots, p_{j_m}$ visible from p_n, let these be p_{j_a} and p_{j_b}. Then the convex hull sought for is $p_{j_0}, \ldots, p_{j_a}, p_n, p_{j_b}, p_{j_m}$.

How to determine whether some vertex p_{j_s} is visible from p_n? The point p_{n-1} is evidently among the vertices of the polygon and is visible from p_n; let $j_t = n - 1$. If $s < t$ then, obviously, p_{j_s} is visible from p_n if and only if p_n is below the line $p_{j_s} p_{j_{s+1}}$. Similarly, if $s > t$ then p_{j_s} is visible from p_n if and only if p_n is above the line $p_{j_s} p_{j_{s-1}}$. In this way, it can be decided about every p_{j_s} in $O(1)$ steps whether it is visible from p_n.

Using this, we can determine a and b in $O(\log n)$ steps and we can perform the "insertion" of the point p_n. This recursion gives an algorithm with $O(n \log n)$ steps.

It is worth separating here the steps in which we do computations with the coordinates of the points, from the other steps (of combinatorial character). We do not know namely, how large are the coordinates of the points, whether multiple-precision computation is needed, etc. Analysing the described algorithm, we can see that the coordinates needed to be taken into account only in two ways: at the sorting, when we had to make comparisons among the abscissas, and at deciding whether point p_n was above or below the straight line determined by the points p_i and p_j. The last one can be also formulated by saying that we must determine the orientation of the triangle $p_i p_j p_k$. This can be done in several ways using the tools of analytic geometry.

The above algorithm can again be described by a binary decision tree: each of its nodes corresponds either to the comparison of the abscissas of two given points or to the determination of the orientation of a triangle given by three points. The algorithm gives a tree of depth $O(n \log n)$. (Many other algorithms looking for the convex hull lead to a decision tree of similar depth.)

9.2 **Exercise** Show that the problem of sorting n real numbers can be reduced in a linear number of steps to the problem of determining the convex hull of n planar points. $\diamondsuit$

*(9.3) **Exercise** Show that the second phase of the above algorithm, i.e. the determination of the convex hull of the points $p_1, \ldots, p_i$ for $i = 2, \ldots, n$, can be performed in $O(n)$ steps provided that the points are already sorted by their x coordinates. $\diamondsuit$

To formalize the notion of a decision tree let us be given the set A of possible inputs, the set B of possible outputs and a set Φ of functions defined on A with values in $\{1, \ldots, d\}$, the **allowed test-functions**. A **decision tree** is a rooted tree whose internal points (including the root) have d children (the tree is d-regular), its endpoints are labelled with the elements of B, the other points with the functions of Φ. We assume that for every vertex, the edges leaving it are numbered in some order.

Every decision tree determines a function $f : A \to B$. Let namely $a \in A$. Starting from the root, we walk down to an endpoint as follows. If we are in an internal point v then we compute the test function assigned to v at the place a; if its value is i then we step further to the i-th child of node v. In this way, we arrive at an endpoint w; the value of $f(a)$ is the label of w.

The question is that for a given function f, what is the shallowest decision tree computing it.

In the simplest case, we want to compute a Boolean function $f(x_1, \dots, x_n)$ and every test that can be made in the vertices of the decision tree is the reading in of the value of one of the variables. In this case, we call the decision tree **simple**. Every simple decision tree is binary (2-regular), the branching points are indexed with the variables, the endpoints with 0 and 1. Such is the yes-no question of whether there is an absolute winner, in a competition by elimination. Notice that the decision tree concerning sorting is not such: there, the tests (comparisons) are not independent since the ordering is transitive. We denote by $D(f)$ the minimal depth of a simple decision tree computing a Boolean function f.

(9.2) **Example** Consider the Boolean function $f(x_1, x_2, x_3, x_4) = (x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (x_3 \vee x_4)$. This is computed by the following simple decision tree. (Here, the root is on the left, the leaves on the right and the levels of the tree are indicated by indentation.)

- x_2
    - x_3
        - 1
        - x_4
            - 1
            - 0
    - x_3
        - x_1
            - 1
            - 0
        - 0

Therefore $D(f) \le 3$. It is easy to see that $D(f) = 3$. $\diamondsuit$

Every decision tree can also be considered a two-person "twenty questions"-like game. One player (Xavier) thinks of an element $a \in A$, and it is the task of the other player (Yvette) to determine the value of $f(a)$. For this, she can pose questions to Xavier. Her questions cannot be, however, arbitrary, she can only ask the value of some test function in Φ. How many questions do suffice for her to compute the answer? Yvette's strategy corresponds to a decision tree, and Xavier plays optimally if with his answers, he drives Yvette to the endpoint farthest away from the root. (Xavier can "cheat, as long as he is not caught"—i.e., he can change his mind about the element $a \in A$ as long as the new one still makes all his previous answers correct. In case of a simple decision tree, Xavier has no such worry at all.)

9.2 Nondeterministic decision trees

The idea learned in Section 5, **nondeterminism**, helps in other complexity-theoretic investigations, too. In the decision-tree model, this can be formulated as follows (we will only consider the case of simple decision trees). Let $f : \{0,1\}^n \to \{0,1\}$ be the function to be computed. Two numbers characterize the nondeterministic decision-tree complexity (similarly to having two non-deterministic polynomial classes, P and NP). For every input x, let $D(f,x)$ denote the minimum number of those variables whose value already determines the value of $f(x)$. Let

$$D_0(f) = \max\{\, D(f,x) : f(x) = 0 \,\}, \qquad D_1(f) = \max\{\, D(f,x) : f(x) = 1 \,\}.$$

In other words, $D_0(f)$ is the smallest number with the property that for all inputs x with $f(x) = 0$, we can test $D_0(f)$ variables in such a way that knowing these, the value of the function can already determined (it may depend on x which variables we will test). The number $D_1(f)$ can be characterized similarly. Obviously,

$$D(f) \geq \max\{D_0(f), D_1(f)\}.$$

It can be seen from the examples below that equality does not necessarily hold here.

(9.1) **Example** Assign a Boolean variable x_e to each edge e of the complete graph K_n. Then every assignment corresponds to an n-point graph (we connect with edges those pairs whose assigned value is 1). Let f be the Boolean function with $\binom{n}{2}$ variables whose value is 1 if in the graph corresponding to the input, the degree of every node is at least one and 0 if not (i.e. if there is an isolated point). Then $D_0(f) \leq n-1$ since if there is an isolated point in the graph it is enough to know about the $n-1$ edges leaving it that they are not in the graph. It is also easy to see that we cannot infer an isolated point from the connectedness or unconnectedness of $n-2$ pairs, and thus

$$D_0(f) = n - 1.$$

Similarly, if there are no isolated points in a graph then this can be proved by the existence of $n-1$ edges (it is enough to know one edge leaving each node and one of the edges even covers 2 nodes). If the input graph is an $n-1$-arm star then fewer than $n-1$ edges are not enough. Therefore

$$D_1(f) = n - 1.$$

Thus, whichever is the case, we can know the answer after $n-1$ lucky questions. On the other hand, if we want to decide which one is the case then we cannot know in advance which edges to ask; it can be shown that the situation is as bad as it can be, namely

$$D(f) = \binom{n}{2}.$$

We return to the proof of this in the next subsection (exercise 9.8). $\diamondsuit$

(9.2) **Example** Let now G be an arbitrary but fixed n-point graph an let us assign a variable to each of its vertices. An assignment of the variables corresponds to a subset of the vertices. Let the value of the function f be 0 if this set is independent in the graph and 1 otherwise. This property can also be simply expressed by a Boolean formula:

$$f(x_1, \dots, x_n) = \bigvee_{ij \in E(G)} (x_i \wedge x_j).$$

If the value of this Boolean function is 1 then this will be found out already from testing 2 vertices, but of course not from testing a single point, i.e.

$$D_1(f) = 2.$$

On the other hand, if after testing certain points we are sure that the set is independent then the vertices that we did not ask must form an independent set. Thus

$$D_0(f) \ge n - \alpha$$

where α is the maximum number of independent points in the graph. It can also be proved (see Theorem 9.7) that if n is a prime and a cyclic permutation of the points of the graph maps the graph onto itself, and the graph has some edges but is not complete, then

$$D(f) = n.$$

$\diamondsuit$

We see therefore that $D(f)$ can be substantially larger than the maximum of $D_0(f)$ and $D_1(f)$, moreover, it can be that $D_1(f) = 2$ and $D(f) = n$. However, the following beautiful relation holds:

(9.3) **Theorem**
$$D(f) \le D_0(f) D_1(f).$$

Proof We use mathematical induction over the number n of variables. If $n = 1$ then the inequality is trivial.

Let (say) $f(0, \dots, 0) = 0$; then $k \le D_0(f)$ variables can be chosen such that fixing their values to 0, the function is 0 independently of the other variables. We can assume that the first k variables have this property.

Next, consider the following decision tree: we ask the value of the first k variables, let the obtained answers be $a_1, \dots, a_k$. Fixing these, we obtain a Boolean function

$$g(x_{k+1}, \dots, x_n) = f(a_1, \dots, a_k, x_{k+1}, \dots, x_n).$$

Obviously, $D_0(g) \le D_0(f)$ and $D_1(g) \le D_1(f)$. We claim that the latter inequality can be strenghened:

$$D_1(g) \le D_1(f) - 1.$$

Consider an input $(a_{k+1}, \dots, a_n)$ of g with $g(a_{k+1}, \dots, a_n) = 1$. Together with the bits $a_1, \dots, a_k$, this gives an input of the Boolean function f for which $f(a_1, \dots, a_n) = 1$. According to the definition of the quantity $D_1(f)$, one can choose $m \le D_1(f)$ variables, say,

$x_{i_1}, \ldots, x_{i_m}$ of f such that fixing them at the value a_i, the value of f becomes 1 independently of the other variables. One of the first k variables must occur among these m variables; otherwise, $f(0, \ldots, 0, a_{k+1}, \ldots, a_n)$ would have to be 0 (due to the fixing of the first k variables) but would also have to be 1 (due to the fixing of $x_{i_1}, \ldots, x_{i_m}$), which is a contradiction. Thus, in the function g, at the position $a_{k+1}, \ldots, a_k$, only $m-1$ variables must be fixed to obtain the identically 1 function. From this, the claim follows.

From the inductive assumption,

$$D(g) \le D_0(g)D_1(g) \le D_0(f)(D_1(f)-1)$$

and hence

$$D(f) \le k + D(g) \le D_0(f) + D(g) \le D_0(f)D_1(f).$$

■

In Example 9.2, we could define the function by a disjunctive 2-normal form and $D_1(f) = 2$ was true. This is not an accidental coincidence:

(9.4) **Proposition** *If f is expressible by a disjunctive k-normal form then $D_1(f) \le k$. If f is expressible by a conjunctive k-normal form then $D_0(f) \le k$.*

Proof It is enough to prove the first assertion. Let $(a_1, \ldots, a_n)$ be an input for which the value of the function is 1. Then there is an elementary conjunction in the disjunctive normal form whose value is 1. If we fix the variables occurring in this conjunction then the value of the function will be 1 independently of the values of the other variables. ■

For monotonic functions, the connection expressed in the previous proposition is even tighter:

(9.5) **Proposition** *A monotonic Boolean function is expressible by a disjunctive [conjunctive] k-normal form if and only if $D_1(f) \le k$ [$D_0(f) \le k$].*

Proof According to Proposition 9.4, it is sufficient to see that if $D_1(f) = k$ then f is expressible by a disjunctive k-normalform. Let $\{x_{i_1}, \ldots, x_{i_m}\}$ be a subset of the variables minimal with respect to containment, that can be fixed in such a way as to make the obtained function is identically 1. (Such a function is called a **mintag**.) Notice that then we had to fix every variable x_{i_j} necessarily to 1: due to the monotonicity, this fixing gives the identically 1 function, and if a variable could also be fixed to 0 then it would not have to be fixed to begin with.

We will show that $m \le k$. Let us namely assign the value 1 to the variables $x_{i_1}, \ldots, x_{i_m}$ and 0 to the others. According to the foregoing, the value of the function is 1. By the definition of the quantity $D_1(f)$, we can fix in this assignment k values in such a way as to make the obtained function identically 1. By the above remarks, we can assume that we only fix 1's, i.e. we only fix some of the variables $x_{i_1}, \ldots, x_{i_m}$. But then due to the minimality of the set $\{x_{i_1}, \ldots, x_{i_m}\}$, we had to fix all of them, and hence $m \le k$.

Let us prepare for every mintag S the elementary conjunction $E_S = \bigwedge_{x_i \in S} x_i$ and take the disjunction of these. By what was said above, we obtain a disjunctive k-normal form this way. It can be verified trivially that this defines the function f. ■

9.4 **Exercise** Give an example showing that in Proposition 9.5, the condition of monotonicity cannot be omitted. $\diamondsuit$

9.3 Lower bounds on the depth of decision trees

We mentioned that decision trees as computation models have the merit that non-trivial lower bounds can be given for their depth. First we mention, however, a non-trivial lower bound also called **information-theoretic estimate**.

(9.1) **Lemma** *If the range of f has t elements then the depth of every decision tree of degree d computing f is at least $\log_d t$.*

Proof A d-regular rooted tree of depth h has at most d^h endpoints. Since every element of the range of f must occur as a label of an endpoint it follows that $t \geq d^h$. $\blacksquare$

For application, let us take an arbitrary sorting algorithm. The input of this is a permutation $a_1, \dots, a_n$ of the elements $1, 2, \dots, n$, its output is the same, while the test functions compare two elements:

$$\varphi_{ij}(a_1, \dots, a_n) = \begin{cases} 1 & \text{if } a_i < a_j \\ 0 & \text{otherwise.} \end{cases}$$

Since there are $n!$ possible outputs, the depth of any binary decision tree computing the complete order is at least $\log n! \sim n \log n$. The sorting algorithm mentioned in the introduction makes at most $\lceil \log n \rceil + \lceil \log(n-1) \rceil + \cdots + \lceil \log 1 \rceil \sim n \log n$ comparisons.

This bound is often very weak; if e.g. only a single bit must be computed then it says nothing. Another simple trick for proving lower bounds is the following observation.

(9.2) **Lemma** *Assume that there is an input $a \in A$ such that no matter how we choose k test functions, say, $\varphi_1, \dots, \varphi_k$, there is an $a' \in A$ for which $f(a') \neq f(a)$ but $\varphi_i(a') = \varphi_i(a)$ holds for all $1 \leq i \leq k$. Then the depth of every decision tree computing f is greater than k.*

For application, let us see how many comparisons suffice to find the largest one of n elements. We have seen (championship by elimination) that $n-1$ comparisons are enough for this. Lemma 9.1 gives only $\log n$ for lower bound; but we can apply Lemma 9.2 as follows. Let $a = (a_1, \dots, a_n)$ be an arbitrary permutation, and consider $k < n-1$ comparison tests. The pairs (i, j) for which a_i and a_j will be compared form a graph G over the underlying set $\{1, \dots, n\}$. Since it has fewer than $n-1$ edges this graph falls into two disconnected parts, G_1 and G_2. Without loss of generality, let G_1 contain the maximal element and let p denote its number of vertices. Let $a' = (a'_1, \dots a'_n)$ be the permutation containing the numbers $1, \dots, p$ in the positions corresponding to the vertices of G_1 and the numbers $p+1, \dots, n$ in those corresponding to the vertices of G_2; the order of the numbers within both sets must be the same as in the original permutation. Then the maximal element is in different places in a and in a' but the given k tests give the same result for both permutations.

9.5 **Exercise** Show that to pick the middle one by magnitude among $2n+1$ elements,

(a) at least $2n$ comparisons are needed;

*(b) $O(n)$ comparisons suffice.

$\diamondsuit$

In what follows we show estimates for the depth of some more special decision trees, applying, however, some more interesting methods. First we mention a result of Best, Schrijver and Van Emde Boas, then one of Rivest and Vuillemin which gives a lower bound of unusual character for the depth of decision trees.

(9.3) **Theorem** *Let $f : \{0,1\}^n \to \{0,1\}$ be an arbitrary Boolean function. Let N denote the number of those substitutions making the value of the function "1" and let 2^k be the largest power of 2 dividing N. Then the depth of any decision tree computing f is at least $n - k$.*

Proof Consider an arbitrary decision tree of depth d that computes the function f, and a leave of this tree. Here, $m \le d$ variables are fixed, therefore there are at least 2^{n-m} inputs leading to this leaf. All of these correspond to the same function value, therefore the number of inputs leading to this leaf and giving the function value "1" is given by either 0 or or 2^{n-m}. This number is therefore divisible by 2^{n-d}. Since this holds for all leaves, the number of inputs giving the value "1" is divisible by 2^{n-d} and hence $k \ge n - d$. ■

By an appropriate extension of the above proof, the following generalization of Theorem 9.3 can be proved.

(9.4) **Theorem** *For a given n-variable Boolean function f, let us construct the following polynomial: $\Psi_f(t) = \sum f(x_1, \dots, x_n) t^{x_1 + \dots + x_n}$ where the summation extends to all systems of values $(x_1, \dots, x_n) \in \{0,1\}^n$. If f can be computed by a decision tree of depth d then $\Psi_f(t)$ is divisible by $(t+1)^{n-d}$.*

9.6 **Exercise** Prove Theorem 9.4. $\diamondsuit$

We call a Boolean function f of n variables **laconic** if it cannot be computed by a decision tree of length smaller than n. It follows from Theorem 9.3 that *if a Boolean function has an odd number of substitutions making it "1" then the function is laconic.*

We obtain another important class of laconic functions by symmetry-conditions. A Boolean function is called **symmetric** if every permutation of its variables leaves its value unchanged. E.g., the functions $x_1 + \dots + x_n$, $x_1 \vee \dots \vee x_n$ and $x_1 \wedge \dots \wedge x_n$ are symmetric. A Boolean function is symmetric if and only if its value depends only on how many of its variables are 0 resp. 1.

(9.5) **Proposition** *Every non-constant symmetric Boolean function is laconic.*

Proof Let $f : \{0,1\}^n \to \{0,1\}$ be the Boolean function in question. Since f is not constant, there is a j with $1 \le j \le n$ such that if $j-1$ variables have value 1 then the function's value is 0 but if j variables are 1 then the function's value is 1 (or conversely).

Using this, we can propose the following strategy to Xavier. Xavier thinks of a 0-1-sequence of length n and Yvette can ask the value of each of the x_i. Xavier answers 1 on the first $j-1$ questions and 0 on every following question. Thus after $n-1$ questions, Yvette cannot know whether the number of 1's is $j-1$ or j, i.e. she cannot know the value of the function. ∎

Symmetric Boolean functions are very special; the following class is significantly more general. A Boolean function of n variables is called **weakly symmetric** if for all pairs x_i, x_j of variables, there is a permutation of the variables that takes x_i into x_j but does not change the value of the function. E.g. the function

$$(x_1 \wedge x_2) \vee (x_2 \wedge x_3) \vee \cdots \vee (x_{n-1} \wedge x_n) \vee (x_n \wedge x_1)$$

is weakly symmetric but not symmetric. The question below (the so-called generalized Aandera-Rosenberg-Karp conjecture) is open:

(9.6) **Conjecture** *If a non-constant monotonic Boolean function is weakly symmetric then it is laconic.*

We show by an application of Theorem 9.4 that this conjecture is true in an important special case.

(9.7) **Theorem** *If a non-constant monotonic Boolean function is weakly symmetric and the number of its variables is a prime number then it is laconic.*

Proof It is enough to show that $\Psi_f(n-1)$ is not divisible by n. First of all, we use the group-theoretical result that (with a suitable indexing of the variables) the substitution $x_1 \to x_2 \to \cdots \to x_n \to x_1$ does not change the value of the function. It follows that in the definition of $\Psi_f(n-1)$, if in some term, not all the values $x_1, \ldots, x_n$ are identical then n identical terms can be made from it by cyclic substitution. The contribution of such terms is therefore divisible by n. Since the function is not constant and is monotonic, it follows that $f(0,\ldots,0) = 0$ and $f(1,\ldots,1) = 1$, from which it can be seen that $\Psi_f(n-1)$ gives remainder $(-1)^n$ modulo n. ∎

We get important examples of weakly symmetric Boolean functions taking any **graph property**. Consider an arbitrary property of graphs, e.g. planarity; we only assume that if a graph has this property then every graph isomorphic with it also has it. We can specify a graph with n points by fixing its vertices (let these be $1,\ldots,n$), and for all pairs $i,j \subset \{1,\ldots,n\}$, we introduce a Boolean variable x_{ij} with value 1 if i and j are connected and 0 if they are not. In this way, the planarity of n-point graph can be considered a Boolean function with $\binom{n}{2}$ variables. Now, this Boolean function is weakly symmetric: for every two pairs, say, $\{i,j\}$ and $\{u,v\}$, there is a permutation of the vertices taking i into u and j into v. This permutation also induces a permutation on the set of point pairs that takes the first pair into the second one and does not change the planarity property.

A graph property is called **trivial** if either every graph has it or no one has it. A graph property is **monotonic** if whenever a graph has it each of its subgraphs has it. For most graph properties that we investigate (connecivity, the existence of a Hamiltonian circuit, the existence of complete matching, colorability, etc.) either the property itself or its negation is monotonic.

The Aandera-Rosenberg-Karp conjecture applied, in its original form, to graph properties:

(9.8) **Conjecture** *Every non-trivial monotonic graph property is laconic, i.e., every decision tree that decides such a graph property and that can only test whether two nodes are connected, has depth* $\binom{n}{2}$.

This conjecture is proved for a number of graph properties: for a general property, what is known is only that the tree has depth $\Omega(n^2)$ (Rivest and Vuillemin) and that the theorem is true if the number of points is a prime power (Kahn, Saks and Sturtevant). The analogous conjecture is also proved for bipartite graphs (Yao).

9.7 **Exercise** Prove that the connectedness of a graph is a laconic property. ♢

9.8 **Exercise**

(a) Prove that if n is even then on n fixed points, the number of graphs not containing isolated points is odd.

(b) If n is even then the graph property that in an n-point graph there is no isolated point, is laconic.

*(c) This statement holds also for odd n.

♢

9.9 **Exercise** A **tournament** is a complete graph each of whose edges is directed. Each tournament can be described by $\binom{n}{2}$ bits saying how the individual edges of the graph are directed. In this way, every property of tournamentse can be considered an $\binom{n}{2}$-variable Goolean function. Prove that the tournament property that there is a 0-degree vertex is laconic. ♢

Among the more complex decision trees, the **algebraic decision trees** are important. In this case, the input is n real numbers $x_1, \dots, x_n$ and every test function is described by a polynomial; in the branching points, we can go in three directions according to whether the value of the polynomial is negative, 0 or positive (sometime, we distinguish only two of these and the tree branches only in two). An example is provided for the use of such a decision tree by sorting, where the input can be considered n real numbers and the test functions are given by the polynomials $x_i - x_j$.

A less trivial example is the determination of the convex hull of n planar points. Remember that the input here is $2n$ real numbers (the coordinates of the points), and the test functions are represented either by the comparison of two coordinates or by the determination of the orientation of a triangle. The points $(x_1, y_1), (x_2, y_2)$ and (x_3, y_3) form a triangle with positive orientation if and only if

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} > 0.$$

This can be considered therefore the determination of the sign of a second-degree polynomial. The algorithm described in Subsection 9.1 gives thus an algebraic decision tree in which the test functions are given by polynomials of degree at most two and whose depth is $O(n \log n)$.

The following theorem of Ben-Or provides a general lower bound on the depth of algebraic decision trees. Before the formulation of the theorem, we introduce an elementary topological notion. Let $U \subset \mathbf{R}^n$ be a set in the n-dimensional space. Two points x_1, x_2 of the set U are called **equivalent** if there is no decomposition $U = U_1 \cup U_2$ for which $x_i \in U_i$ and the closure of U_1 is disjoint from the clusure of U_2. The equivalence classes of this equivalence relation are called the **components** of U. We call a set **connected** if it has only a single connected component.

(9.9) **Theorem** *Suppose that the set $U \subset \mathbf{R}^n$ has at least N connected components. Then every algebraic decision tree deciding $x \in U$ whose test functions are polynomials of degree at most d, has depth at least $\log N / \log(6d) - n$. If $d = 1$ then the depth of every such decision tree is at least $\log_3 N$.*

Proof We give the proof first for the case $d = 1$. Consider an algebraic decision tree of depth h. This has at most 3^h endpoints. Consider an endpoint reaching the conclusion $x \in U$. Let the results of the tests on the path leading here be, say,

$$f_1(x) = 0, \quad \ldots, \quad f_j(x) = 0, \quad f_{j+1}(x) > 0, \quad \ldots, \quad f_h(x) > 0.$$

Let us denote the set of solutions of this set of equations and inequalities by K. Then every input $x \in K$ leads to the same endpoint and therefore we have $K \subset U$. Since every test function f_i is linear, the set K is convex and is therefore connected. Therefore K is contained in a single connected component of the set U. It follows that the inputs belonging to different components of U lead to different endpoints of the tree. Therefore $N \le 3^h$, which proves the statement referring to the case $f = 1$.

In the general case, the proof must be modified in that K is not necessarily convex and so not necessarily connected either. Instead, we can use an important result from algebraic geometry (a theorem of Milnor and Thom) implying that the number of connected components of K is at most $(2d)^{n+h}$. From this, it follows similarly to the first part that

$$N \ge 3^h (2d)^{n+h} \ge (6d)^{n+h},$$

which implies the statement of the theorem. ∎

For an application, consider the following problem: given n real numbers $x_1, \ldots, x_n$; let us decide whether they are all different. We consider an elementary step the comparison of two given numbers, x_i and x_j. This can have three outcomes: $x_i < x_j$, $x_i = x_j$ and $x_i > x_j$. What is the decision tree with the smallest depth solving this problem?

It is very simple to give a decision tree of depth $n \log n$. Let us namely apply an arbitrary sorting algorithm to the given elements. If anytime during this, two compared elements are found to be equal then we can stop since we know the answer. If not then after $n \log n$ steps, we can order the elements completely, and thus they are all different.

Let us convince ourselves that $\Omega(n \log n)$ comparisons are indeed needed. Consider the following set:

$$U = \{ (x_1, \ldots, x_n) : x_1, \ldots, x_n \text{ are all different } \}.$$

This set has exactly $n!$ connected components (two n-tuples belong to the same component if they are ordered in the same way). So, according to Theorem 9.9, every algebraic decision tree deciding $x \in U$ in which the test functions are linear, has depth at least $\log_3(n!) = \Omega(n \log n)$. The theorem also shows that we cannot gain an order of magnitude with respect to this even if we permitted quadratic or other bounded-degree polynomials as test polynomials.

We have seen that the convex hull of n planar points in general position can be determined by an algebraic decision tree of depth $n \log n$ in which the test polynomials have degree at most two. Since the problem of sorting can be reduced to the problem of determining the convex hull it follows that this is essentially optimal.

9.10 **Exercise** (a) If we allow a polynomial of degree n^2 as test function then a decision tree of depth 1 can be given to decide whether n numbers are different.

(b) If we allow degree n polynomials as test functions then a depth n decision tree can be given to decide whether n numbers are different.

$\diamondsuit$

9.11 **Exercise** Given are $2n$ different real numbers: $x_1, \ldots, x_n, y_1, \ldots, y_n$. We want to decide whether it is true that ordering them by magnitude, there is a x_j between every pair of y_i's. Prove that this needs $\Omega(n \log n)$ comparisons. $\diamondsuit$