Chapter 5

Search Algorithms

From a general point of view, a *search* of a graph, or a digraph, is an algorithm which makes it possible to search the arcs of the graph and to visit its vertices with a special purpose in mind. This chapter presents one of the most classic of these searches, called a *depth-first search* (often abbreviated *dfs*). This type of tree-search will be completed in Chapter 6 by another classic tree-search, the *breadth*-first search.

In applications which are modeled by an arborescence, this search technique is called *backtracking*, and can be used to solve a wide variety of problems in operations research and artificial intelligence.

5.1 Depth-first search of an arborescence

From an algorithmic viewpoint, the *recursive* form is the most natural and most efficient to express this search. The following procedure expresses the depth-first search of the subarborescence of arborescence T, of root v. We designate as `children(v)` the set of the children of vertex parameter `v`.

```
procedure dfs_arbo_recu(T,v);
begin
  for u in children(v) loop
    dfs_arbo_recu(T,u);  -- recursive call
  end loop;
end dfs_arbo_recu;
```

The complete arborescence is searched by a call of this recursive procedure on the root r of the arborescence T, that is, the call: `dfs_arbo_recu(T,r)`.

In practice, as in the applications we will see later, an arborescence is usually given by what may be called "navigation primitives", meaning subprograms which allow a child or a sibling of a given vertex to be reached. In order to be more specific, the children of the vertex are usually given in a particular order defined by a list. Remember that it is then an *ordered* arborescence, and that we can refer to the *first child* of a vertex when this vertex is not a leaf, and to the *following sibling* of a vertex if it has one. We express the preceding search with the following primitives in which the names indicate what they are for: `exists_sibling(v)` is a Boolean function which returns *true* or *false* depending on whether the vertex parameter `v` has or has not a child in the arborescence; `first_child(v)` returns the first sibling, when it exists. We similarly define the primitive `exists_following_sibling(v)`, which returns *true* or *false* depending on whether or not there is a following sibling of the vertex parameter `v`, and `following_sibling(v)` which returns the first following sibling (the one just following), when it exists.

```
procedure dfs_arbo_recu(T,v);
begin
  if exists_child(v) then
    u:= first_child(v);
    dfs_arbo_recu(T,u);  -- recursive call
    while exists_following_sibling(t) loop
      u:= following_sibling(u);
      dfs_arbo_recu(T,u);  -- recursive call
    end loop;
  end if;
end dfs_arbo_recu;
```

5.1.1 Iterative form

It is interesting to eliminate the recursion in the preceding algorithm to follow the search strategy step by step. One of the possible iterative forms is given below. We could have used a "parent" primitive of the arborescence, which would have made it possible to avoid having to use a stack. In fact, in practice it is easier to do without this primitive by using a stack.

In addition, this stack recalls the stack used by the computer system for the management of recursive calls and recursive returns during execution. The stack primitives used here are classic. Let us specify that `pop(S)` removes the element which is at the top of stack `S`, *without returning it*, and `top_stack(S)` returns the element which is at the top of stack `S`, without removing it from `S`. Variable vertex `v` is local. It represents the current vertex of the search. Parameter `r` represents the root of arborescence `T`, also passed as a parameter of the procedure. Remember that the instruction **exit** causes exit from the current loop.

```
procedure dfs_arbo_ite(T,r);
begin
  push(S,r);
  v:= r;
  loop
    while exits_child(v) loop
      v:= first_child(v);
      push(S,v);
    end loop;
    -- exits_child(v) false
    while v ≠ r and then not exists_following_sibling(v) loop
      pop(S);
      v:= top_stack(S);
    end loop;
    -- v = r or exists_following_sibling(v) true
    exit when v = r;
    pop(S);
    v:= following_sibling(v);
    push(S,v);
  end loop;
  pop(S);
end dfs_arbo_ite;
```

NOTES. 1) As formulated, with the **and then** (the second condition is tested only if the first is verified), the **exit** condition of the second **while** loop ensures that primitive `exists_following_sibling` will not be called on root r, which makes it possible to avoid having to plan this particular case for this primitive. This case is in fact useless since the root of an arborescence never has a sibling.

2) Stack `S` is initially assumed to be empty. It is then also empty at the end of the execution. Indeed, the last pop, after the main **loop**, exits vertex `r`, which is the first and last vertex in the stack.

The strategy of this search may be described in natural language by following the moves of the current vertex v (moves defined by the successive assignment of the variable `v` in the algorithm). Initially, v is in r. Then at each step of the search the current vertex v goes: to the first child of v if v has a child, to the following sibling of v if v has no child or v no longer has any child left unconsidered but has a following sibling, and finally, v goes to the parent of v if v no longer has any child or following sibling left unconsidered but has a parent, that is if $v \neq r$. The search ends when v is back to r. This description reveals the priority for vertex v to move first to a child, what can be called the "depth-first descent", and explains the source of the terminology "depth-first search". The given algorithmic expression shows this strategy through the layout of the loops. The first interior loop (**while**) corresponds to the onward search, the second interior loop (**while**) corresponds to returning to the parent. The exterior loop (**loop**) corresponds to a move toward the following sibling, between the onward and upward searches expressed by the two preceding loops.

5.1.2 Visits to the vertices

As we will see, the use of the search is made through some timely appropriate actions while visiting the various vertices of the arborescence. It is possible to specify these visits in terms of *previsits* or *postvisits*, and equally to spot the visits through the leaves of the arborescence, which are often important. This is done as commentaries of the following version of procedure `dfs_arbo_ite`, which completes the iterative version given earlier. These visits are easy to spot in the recursive version (procedure `dfs_arbo_recu`), because *each previsit corresponds to a push* and *each postvisit corresponds to a pop*. So, there is a *previsit* of the current vertex before a recursive call on this vertex and there is a *postvisit* at the time of the recursive return. This corresponds to the iterative version since, in the management of the recursion, pushes correspond to recursive calls and pops to recursive returns (except for the root).

```
procedure dfs_arbo_ite(T,r);
begin
  -- previsit of r
  push(P,r);
  v:= r;
  loop
    while exists_child(v) loop
      v:= first_child(v);
      -- previsit of v
      push(S,v);
    end loop;
    -- v is a leaf
    while v ≠ r and then not exists_following_sibling(v) loop
      pop(S);
      -- postvisit of v
      v:= top_stack(S);
    end loop;
    -- v = r or exists_following_sibling(v) true
    exit when v = r;
    pop(S);
    -- postvisit of v
    s:= following_sibling(v);
    -- previsit of v
    push(S,v);
  end loop;
  pop(S);
  -- postvisit of v
end dfs_arbo_ite;
```

The numbering defined by the previsits is classically called the *preorder numbering* of the vertices of the arborescence, and the numbering defined by the postvisits the *postorder numbering*. Figure 5.1 gives an example of a depth-first search of an arborescence, with preorder and postorder numbering of the vertices.

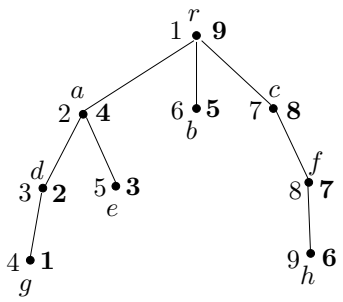

Figure 5.1. *To the left of each vertex is given its preorder number and to the right, in bold type, its postorder number*

5.1.3 Justification

It is easy to be convinced that this algorithm, under any of the versions presented, really performs a search of the arborescence in the sense given above. Indeed, through systematic consideration for each vertex of all its children, each arc is considered and each vertex is visited.

5.1.4 Complexity

Time complexity for this search, as a function of the number of vertices of the arborescence, is linear. Effectively, for each vertex its children are considered only once as children of this vertex. The total number of elementary operations is thus of the order of the sum of the outdegrees of the vertices, that is of the order of the number, m, of arcs of the arborescence. Since $m = n - 1$, where n is the number of vertices, the complexity is $O(n)$. However, if we measure the size of the arborescence by its depth d (the greatest length of a path from the root to a leaf), rather than by the number of vertices, which is much more relevant in applications, the search complexity becomes exponential. For example, the complexity becomes $O(k^d)$ for an arborescence for which any non-leaf vertex has k children and any leaf is of depth d. This complexity is in fact proportional to the number of vertices visited. We will see later the concrete consequences of this exponential complexity.

5.2 Optimization of a sequence of decisions

Let us consider the problem of having to choose one decision among several possibilities at each step of a process. Some states of the process, called *terminal*, no longer require a decision and can be assessed with a *gain* that is an integer or real number, which may be positive, negative or zero. The problem, then, is to find a decision sequence which leads from a given *initial* state to a terminal state with the greatest possible gain.

Modeling this problem by an arborescence is easy: each vertex represents a state, the root represents the initial state, each arc corresponds to a possible decision leading from one state to another. The leaves are the final states and they are assigned to values corresponding to the gains. We have to determine in this arborescence a path from the root to a leaf which has the greatest possible gain value.

As formulated, and with what has been developed above, there is a direct solution to this problem: a depth-first search of the arborescence, recording the values of leaves as they are visited, should bring about a solution. In practice, things are less simple. On the one hand, searching a whole arborescence may be costly in time and frequently even impossible to complete within a reasonable human time scale. On the other hand, the arborescence associated with the problem is not fully known from the start and has to be built, which is not crippling but requires some technical work. Let us illustrate this with a classic algorithmic problem.

5.2.1 The eight queens problem

This is an old puzzle, already known in the 19th century. The aim is to put eight queens on a chessboard so that none of them is able to take another, according to chess rules. Recall that the queen attacks any piece that is in the same row, column or diagonal. It is clear that it is impossible to place more than eight queens. The question therefore is how to find out if it is possible with eight queens.

It is necessary to define the arborescence associated with the problem because several possibilities are conceivable. Since there must not be more than one queen per row, one natural way to proceed is to put one queen per row, row after row, starting from row one (supposing that the rows are numbered from 1 to 8, for example from bottom to top). The root of the

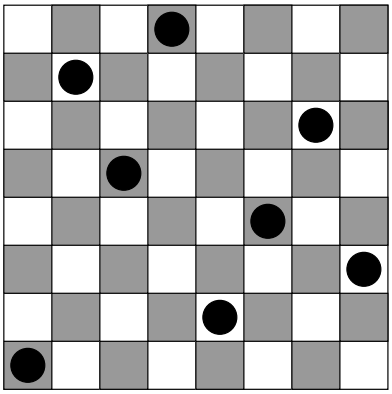

Figure 5.2. *A solution to the eight queens problem*

arborescence is the empty chessboard. The children of the root correspond to the eight ways to place a queen on the first row. The children of the children correspond to the ways of placing a queen on the second row out of reach of the preceding queen, and so on for all the following rows. A leaf of the arborescence corresponds to a state of the chessboard where some queens are already placed on a number of first rows in such a way that it is impossible to add another one on the following row, either because there is in fact no following row or because all the squares of the following row are under the threat of the previously placed queens. The gain associated with an arborescence leaf is the number of queens placed on the board, a number which corresponds to the depth of the leaf in the arborescence. A solution is reached when eight queens are placed, that is for a leaf of depth 8, which is the greatest possible gain.

For the search application to this arborescence, the real work is in defining the arborescence primitives, `exits_child`, `first_child`, `exists_following_sibling`, `following_sibling`. For example, the function `exits_child` must return *true* or *false*, depending on whether there is or is not in the row following the current one a free square where a queen can be placed, taking into account the ones already placed. The function `first_child`, in the case where `exits_child` has returned *true* for the vertex under consideration, must return the first following free square, deciding, for example, to go from left to right on the squares of each row. The complete algorithmic resolution of this problem is proposed as an exercise at the end of this chapter.

5.2.2 Application to game theory: finding a winning strategy

The games under consideration here are *two-player games*, with *complete information* (each player has complete knowledge of the entire game), *hazardless* (no throwing dice or drawing cards for example) and *with zero sum* (the sum of the gains of the two players is zero). Chess is a typical example. The player who starts is denoted by A and the other one by B. Let us consider only simple gains, that is: 1 if A wins (then B loses), -1 if A loses (and B wins), 0 if the game is tied. For this type of game, there is always what is called a *winning strategy*, that is a way of playing for one of the players which, regardless of the moves of the opponent, ensures that he or she does not lose. This means A has a gain ≥ 0 (if A has a winning strategy), B has a gain ≤ 0 (if B has a winning strategy). It is possible to prove that there is always a winning strategy, but it is not possible to say *a priori* which of the players has it. It depends on the game, and both cases really happen.

5.2.3 Associated arborescence

We are going to show how a winning strategy can be found algorithmically, which will constitute a constructive proof of its existence. To do this, let us associate an arborescence with the game in order to apply a search to it. Its root is of course the initial state of the game. Its children are all the states of games obtained after the first move of player A, the children of the children all the states obtained after the move then made by B, and so on, alternating moves by A and B. The leaves are states of the game obtained by a sequence of moves, which then represent a match, and after which there are no more moves, the game being over. For each leaf, the gain is assessed as defined above: 1 if A wins, -1 if B wins and 0 for a tie.

Note. A given state of the game may appear several times as a vertex of the arborescence. This is inherent to this model of the game, since a given situation in a game can generally be obtained by different sequences of moves.

5.2.4 Example

The arborescence of a game is soon enormous (the preceding remarks contribute to that effect)! To present a case which remains accessible, we are going to consider the game of Nim. In its classic form (popularized by a French film in the 1960s), there are four piles of matches containing respectively one, three, five, and seven matches. In turn, each player removes as many matches (but at least one) as he or she wishes, from only one pile. The player removing the last match has lost. We will limit ourselves to a reduced version of this game with only two piles of one and three matches. Its arborescence is fully represented in Figure 5.3. Each state of the game is coded by the data of the number of matches in each pile separated by a hyphen, 1-3 for the initial state for example. Note that despite the great simplicity of this game, its arborescence is already a bit complicated.

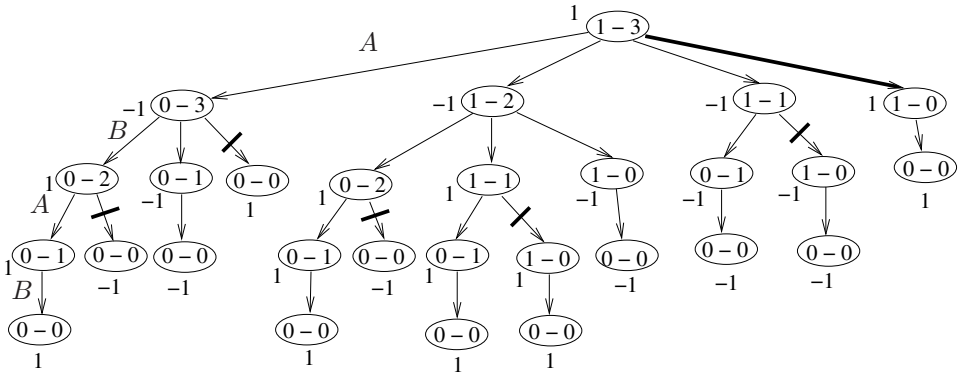

Figure 5.3. *The arborescence of the 1-3 Nim game, gains brought back by application of the minimax algorithm: winning strategy (bold arcs), prunings (bold lines)*

5.2.5 The minimax algorithm

Finding a winning strategy is harder than the simple optimization of a sequence of decisions as described above. Indeed, there is an antagonistic pursuit between the players: the one who starts is looking for a maximum gain, since it is directly his gain. The other is trying to minimize the final gain since his/her gain is the opposite. The essential principle of the algorithm is that of the *mounting of the gains of the leaves*, values which are known, towards the root, where the returned value will indicate which of the players

is benefiting from a winning strategy. In general, the gain returned for a vertex of the arborescence will equal 1 if it is a *winning position* for player A, -1 if it is a *losing position* for A (and therefore winning for B), 0 if it is not a winning position for either of the players. This last situation is that of a tied game, possible for each of the two players if no "mistake" is made, that is the case when each player avoids, as is possible, making a move leading to a winning situation for his opponent. In this particular case, we can say that each player has a winning strategy in the sense defined above (a strategy which would be better called a "non-losing" strategy).

The principle described indicates by itself how to return the gains. Let us take for example the case of a state of the game where it is A's turn, and let us suppose that the gains of all the children of this vertex in the arborescence have already been determined. Then the rule to apply is that the gain returned to the vertex under consideration is the maximum of the gains of its children. That will also define what is the "best move" for A to make at this moment. A similar formula applies in the case where it is B's turn to play, with *minimum* instead of *maximum*. Thus, from bottom to top, that is, from leaves to the root, it is possible to find the values of the gains returned for each vertex of the arborescence and finally for the root. This technique of alternately returning a minimum and a maximum explains the name *minimax* given to this algorithm.

5.2.6 Implementation

A depth-first search of the arborescence of a game is perfectly adapted to the implementation of this technique of gain return. At each postvisit of a vertex the value of its parent is updated by *max* or *min* depending on whether it is a move by A or B which is returned. Indeed, at this point all the children of the vertex under consideration have known gain values.

In order to avoid singling out the postvisit case of the first child of a vertex, a point at which the parent does not yet have a returned value, from the case of the following children, that is to be able to apply one single formula, we initialize in previsit the value of each vertex in the following manner: -1 if it is A's turn, +1 if it is B's turn. These values are chosen in such a way that the value returned the first time will be taken automatically. For example, at the first return of the child of a vertex representing a game situation where A is about to play, the maximum will be taken between -1 and a gain g returned equal to -1, +1 or 0. Therefore, the taken gain will

necessarily be g. The same result can be obtained for B with a minimum between +1 and a gain equal to -1, +1 or 0.

5.2.7 In concrete terms

To effectively find a winning strategy, it is necessary to keep a record of the arc which gave the best return gain so far for each vertex throughout the search. Figure 5.3 illustrates this method. In concrete terms, the application of this minimax algorithm to a realistic game is impossible to do within a reasonable time period, so enormous is the arborescence to be searched. For example, it is impossible to search the entire chess game arborescence (the number of leaves of its arborescence can be evaluated to be 20^{50}). Even if it was possible, the storage of the information for a winning strategy would create problems. Such a strategy must give the right answers for all the possible moves of the opponent. We are facing the "combinatorial explosion" phenomenon, in this case the exponential growth of cases to contemplate.

5.2.8 Pruning

In order to calculate the gain returned to the root, it is possible to reduce the search by *pruning*, that is to "prune" the arborescence. This technique does not modify the exponential nature of the search. Figure 5.3 shows pruning cases which can be understood on their own. For example, when value 1 is returned to vertex 0-2, which is at depth 2 at the bottom left, from vertex 0-1, it is useless to explore the other branch leading to vertex 0-0 since the returned value 1 is the best possible for player A, whose turn it is in this situation. Indeed, whatever the value returned from vertex 0-0, it will not modify the value previously returned in vertex 0-1.

Even though the minimax algorithm does not allow a global exhaustive search, it is nevertheless useful for a *local* search, that is a search which does not necessarily continue until the end of the game but limits itself, for example, to a 10-move exploration depth from the situation analyzed. Finding the best move possible is then done on the basis of evaluation functions which quantitatively assess game situations at the limit of the exploration, situations which are not yet end game and thus without known gains *a priori*. This technique is used by chess game software; their high level of performance is well known and is due essentially to the quality of these evaluation functions which contain in fact all the human "expertise" in this matter.

5.3 Depth-first search of a digraph

The digraph G searched here is assumed to be strict (there are no parallel arcs, no loops, and any arc is identified by the ordered pair of its ends). The digraph is also supposed to be given by *linked* lists of successors, and in the following algorithmic expressions `suc(v)` designates a pointer to an element of the list of vertex `v`, initially to the first element of this list. Each of these elements is a record[1] which contains: `suc(v).vertex` the next successor of `v` to be examined, `suc(v).next` a pointer to the following element of the list, equal to **null** if there are no more elements in it. When a successor has been read, `suc(v)` must be incremented, that is moved to point to the following element of the list of successors. This is what is produced by the assignment `suc(s):= suc(s).next`. The array `visited`, indexed on the vertices of the digraph, is supposed initialized to the value *false* for each vertex.

In a recursive form, we first have the following procedure, with parameter `G` as the digraph to be searched and `v` as the initial vertex of the search:

```
procedure dfs_recu(G,v);
begin
  visited(v):= true;
  while suc(v) ≠ null loop
    u:= suc(v).vertex; suc(v):= suc(v).next;
    if not visited(u) then
      -- previsit of u
      dfs_recu(G,u); -- recursive call
      -- postvisit of u
    else
      -- revisit of u
      null;
    end if;
  end loop;
end dfs_recu;
```

The main procedure for a search starting at a given vertex `r` of `G`, which is the vertex origin of the search and which is passed as a parameter, is written as follows:

[1] *record* in Ada, *struct* in C.

```
procedure dfs(G,r);
begin
  -- previsit of r
  dfs_recu(G,r);
  -- postvisit of r
end dfs;
```

5.3.1 Comments

We have first given the recursive form, which is more concise. It is useful to locate all the different types of visits at the vertices in this algorithm, because it will be of the greatest use in the application of this search. The *previsits* and the *postvisits*, which correspond, as for the above arborescence, respectively to *recursive calls* and *recursive returns*, are mentioned as comments. The case of the *revisit* of a vertex is new, compared with the case of the arborescence. It corresponds to the case of a vertex which has already been visited and is encountered again as successor to the current vertex (a case which may not happen for an arborescence since there is only one path from one vertex to another). This case appears in the **else** of the **if** and since there is then nothing to do, this is made explicit by the instruction **null**.[2]

The following second form of the depth search is iterative and corresponds to recursion elimination of the preceding one. It is therefore the same search strategy. It is instructive to follow the evolution of the stack in this search, in particular for *previsits* and *postvisits* which, as in the arborescence case, correspond respectively to *pushes* and *pops*. The array `visited` is still assumed to be initialized to the value *false* for each vertex of the digraph.

```
procedure dfs_ite(G,r);
begin
  -- previsit of r
  visited(r):= true;
  push(S,r); v:= r;
  loop
    while suc(v) ≠ null loop
```

[2]Be sure to distinguish the *statement* **null**, as here, and the *value* **null** of a pointer which points to nothing, as seen above.

```
      u:= suc(v).vertex; suc(v):= suc(v).next;
      if not visited(u) then
        -- previsit of u
        visited(u):= true;
        push(S,u); v:= u;
      else
        -- revisit of u
        null;
      end if;
    end loop;
    pop(S);
    exit when empty_stack(S);
    -- postvisit of v
    v:= top(S);
  end loop;
  -- postvisit of r
end dfs_ite;
```

Figure 5.4 gives an example of an application.

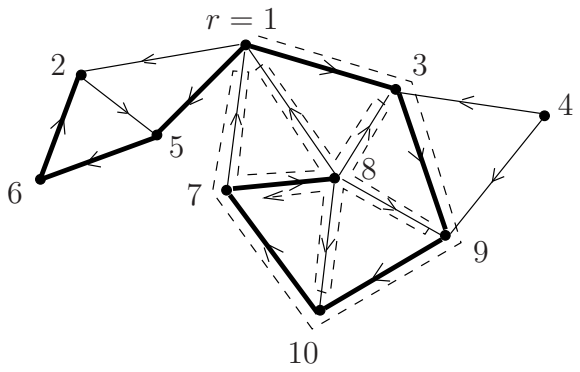

Figure 5.4. *Beginning of a depth-first search of a digraph starting at vertex* $r = 1$. *The visits are: previsits of 1, 3, 9, 10, 7, revisit of 1, previsit of 8, revisits of 1, 3, 9, 10, postvisit of 8, etc. The arcs in bold are those of previsits (for the complete search). They define in the digraph an arborescence of root* $r = 1$, *called Trémaux's arborescence. Note that vertex 4 will not be visited by the search*

The sequence of vertices which are in a stack at a given time defines a path in the digraph, called the *current (directed) path* of the search, the last vertex being the *current vertex*, the one where the search is at that time.

5.3.2 Justification

PROPOSITION 5.1. *During a depth-first search of a strict digraph, any vertex accessible by a path from the initial vertex origin r of the search is visited, with a previsit, then a postvisit, and eventually also a revisit.*

Proof. This is easy to justify by reasoning step by step along the vertices from a path of r to the vertex under consideration. □

Note that the vertices which are unreachable by a path from r are not visited at all. We will remedy this later with an *extended* version of the search.

There is an amusing illustration of this algorithm in the study of mazes, the different ways to go through them and, above all, to come out of them! It is in this context that this algorithm has been known for a long time under the name of *Trémaux's algorithm*, named after the author of studies on this subject in the 19th century, long before the theory of the algorithmic graph. Let us imagine that the digraph represents a maze. The vertices represent the crossroads and the arcs the corridors (assumed to be one way). The preceding search defines a systematic exploration strategy: after the entrance, we take a new corridor as long as there is one to take and it leads to a crossroads not yet visited. If no such corridor is available, and if we are back at the entrance to the maze, we stop; if not, we go back to the crossroad where we were before we arrived for the first time where we are now. Of course, to apply this strategy we have to mark one by one every corridor and crossroads followed. There is a famous precedent in Greek mythology with Theseus, who had to find the Minotaur in the labyrinth, to kill him, and then rediscover the entrance to the labyrinth! Ariane was waiting at this entrance with a ball of thread which Theseus unwound during his search. This "Ariane's thread" allowed him to identify the locations previously visited, and, more importantly, to recover the entrance to the labyrinth at the end of his mission. It is interesting to note that the thread corresponds to the state of the stack of the preceding algorithm. More specifically, at each time, the sequence of the crossroads crossed by the thread corresponds, in the digraph, to the sequence of the vertices in the stack (supposing that Theseus was rewinding the thread when he was retracing his steps in a corridor he had already searched).

5.3.3 Complexity

Let us refer to the second iterative version of the algorithm. The main loop is executed, for each vertex v considered, a number of times equal to the outdegree of v, that is $d_G^+(v)$. Each vertex of the digraph is thus considered once. The total number of elementary operations is thus proportional to the sum of the outdegrees of the vertices visited, a sum which is less than or equal to the number m of arcs of the digraph. The complexity of the search itself is thus $O(m)$. To this must be added the complexity required by the initialization of the array `visited`, that is $O(n)$. In total, the complexity is thus $O(\max(n, m))$. The depth-first search algorithm is linear.

We can be even more specific by saying that the search requires a time proportional to the size of what it is visiting (which is not necessarily all of the digraph).

5.3.4 Extended depth-first search

As we have seen, the preceding search only visits the vertices which can be reached by a path from the initial vertex origin r. When we want to visit all the vertices of the digraph, we have to start a new search from a vertex not yet visited, as long as one exists. This search, called an *extended search*, ends when all vertices of the digraph have been visited. This is what is done by the following algorithm, in which it is important to note that the array visited, still supposed initialized to *false* for each vertex, is global with respect to the different search procedures started. This means that it is not reinitialized between successive searches. A vertex is marked visited only once, by one of the searches. In the following expression of the extended depth-first search, the vertices are supposed numbered from 1 to n.

```
procedure dfs_ext(G);
begin
  r:= 1;
  loop
    dfs_ite(G,r);
    -- looking for a non visited vertex
    while r < n and visited(r) loop
      r:= r + 1;
    end loop
    -- r = n or visited(r) false
```

```
      exit when visited(r);
    end loop;
  end dfs_ext;
```

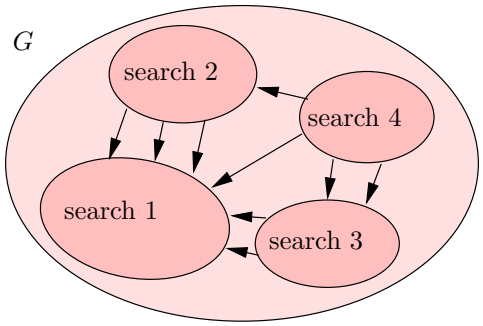

Figure 5.5. *Schema of the successive searches of an extended search*

NOTE. This procedure calls upon the procedure `dfs_ite` but the recursive procedure `dfs_recu` would work just as well.

5.3.5 Justification

PROPOSITION 5.2. *During an extended depth-first search of a strict digraph, each vertex is visited, in a previsit, then a postvisit, eventually also in a revisit.*

Proof. This can be easily justified by observing that any vertex ends up being visited because of the conception of the algorithm itself. □

We can complete the preceding proposition by saying that if the digraph being searched is strict, any arc (u, v) is considered during the search: either during the previsit of v from u, (u, v) is then called a *previsit* arc, or during a revisit of v from u, (v, u) is then called a *revisit* arc. Any arc of the digraph is either a previsit arc or a revisit arc. Note that if (u, v) is a revisit arc, v may have already been visited in a search previous to the one during which u is visited.

5.3.6 Complexity

The various simple searches which constitute the extended one do not overlap, in the sense that a vertex previsited in one of them will not be previsited again in another (let us remember that the array `visited` is global), it can only possibly be revisited. Each search is finite and there is a finite number of searches (at the most equal to the number of vertices, in the case of a digraph without arcs). In addition, as above, each successor of a vertex is considered only once, even when it is a vertex visited in one of the searches with a successor visited in another search. In fact, as we already noted, each search only requires a time proportional to the size of what it is searching, and the total is proportional to the size of the digraph, $\max(n, m)$, thus yielding a linear complexity.

An essential point of the depth-first search, extended or not, is that *when a vertex is postvisited all its successors have been visited, that is previsited from that vertex or revisited (and so previously previsited from another vertex)*. This can be clearly seen in particular with the recursive form of the simple (not extended) search. We can also specify that when a vertex is revisited, it has necessarily been previsited (since it is a *re*visit), but it may or may not be postvisited, that is popped or not from the stack of the iterative expression. This point will be useful for recognizing digraphs without circuits.

5.3.7 Application to acyclic numbering

PROPOSITION 5.3. *(1) A strict digraph G is without a circuit if and only if during an extended depth-first search of this digraph, when a vertex is revisited, it has already been postvisited.*

(2) If digraph G is without a circuit, the postorder of the vertices in an extended depth-first search of this digraph is the reverse of that of an acyclic numbering.

Proof. Let us suppose that during an extended depth-first search of digraph G, there is a vertex u which is the successor of current vertex v and which is revisited but not yet postvisited. Vertex u is thus in the stack at that moment, with v, which is at the top of it, and the sequence of the vertices from u to v in the stack is a directed path (directed subpath of the current path of the search). With the arc from v to u, this directed path defines a circuit of G. This proves, by contradiction, the necessary condition of part (1).

Let us now suppose that the preceding circumstances do not happen during a depth-first search of the digraph. So, when a vertex is revisited it is then postvisited. Let us suppose that the vertices are numbered in the reverse postorder, that is during the postvisits, from n to 1. At the time when a vertex v is postvisited, all its successors have been visited, previsited or revisited, according to a general property of the search noted earlier on. Let us show that they are also all postvisited. Those of the successors of v which have been previsited from v are then necessarily postvisited, because they were in the stack above v and thus necessarily popped before v. The other successors of v have been revisited from v. They were then postvisited at the time of their revisit (by hypothesis). Thus, the successors of v were all numbered before v, and since the numbering is done decreasingly from n to 1, they have received a higher number than the one received by v during its postvisit. That is the property which defines an acyclic numbering. It should also be noted that this property is compatible with the extended nature of the search (the numbering is global). The existence of an acyclic numbering under the hypothesis that when a vertex is revisited it is then postvisited, which results itself from the hypothesis without circuits, proves part (2) of the proposition. It also proves the sufficient condition of part (1), because a digraph which admits an acyclic numbering has no circuit, according to proposition 4.1 of Chapter 4. □

5.3.8 Acyclic numbering algorithms

An acyclic numbering algorithm will consist of launching a depth-first search of the given digraph with the following operation while going through the vertices:

- on *postvisit* of v: mark v postvisited and number it decreasing from n to 1.
- on *revisit* of u: if u is not postvisited, stop because there is a circuit in the digraph.

On *previsit* we do nothing.

† As an application, we can again take the digraph in Figure 4.5 in Chapter 4, which is without a circuit, and apply this algorithm to it. (The acyclic numbering obtained may not be identical to the one given on this figure. It can depend on the search followed.)

This acyclic numbering algorithm is, as for the depth-first search, of linear complexity.

5.3.9 Practical implementation

In practice (as we will see later on, in scheduling with the potential task graph), it is not sufficient to detect the presence of a circuit and to stop, because then the digraph has no acyclic numbering. In fact this circumstance corresponds to an error in the datum of the digraph, one arc too many somewhere, creating a circuit. To correct such an error, it is necessary to have a circuit in order to verify its arcs and to detect one that should not be present. The justification of proposition 5.3 makes it possible to exhibit a detected circuit and to do this work, provided that we have access to the search stack. This is direct with the iterative form but is not directly possible with the recursive form, since the stack is managed by the system and therefore hidden. A solution is then to manage an auxiliary stack provided for that effect, pushing and popping respectively on previsits and postvisits as with the stack of the iterative version of the search.

5.4 Exercises

+5.1. Write two programs (in your preferred language) solving the eight queens problem, in the iterative form as well as the recursive, inspiring yourself from the arborescence searches given. Test them (you should find 92 solutions) and compare their time performances. What conclusion can you draw?

5.2. Give an example of a game in which it is the second player who has a winning strategy.

5.3. Consider the following game:[3] two players called A and B choose alternately the digit 1 or 2. A starts. When four digits have been chosen, the game is over and the greatest prime divisor of the number formed by the four chosen digits, written in the order of their choice from left to right, is then what B wins (and what A loses). Explain what may be an optimal strategy for a player and determine this strategy for B, specifying the minimum gain it ensures.

[3]From Boussard-Mahl, *Programmation avancée*, Eyrolles (1983).

N.B. Prime divisor (between parentheses): 1111 (101), 1112 (139), 1121 (59), 1122 (17), 1211 (173), 1212 (101), 1221 (37), 1222 (47), 2111 (2111), 2112 (11), 2121 (101), 2122 (1061), 2211 (67), 2212 (79), 2221 (2221), 2222 (101).

*5.4. *(Digraph kernel and winning strategy)*

Let us go back to the game of Nim 1-3 dealt with as an example, but with a different representation of the game from that of the arborescence associated with it and described in this chapter (Figure 5.3).

a) Find all possible states of the game (there are eight). Build the digraph of which the vertices are these states, with an arc from one vertex to another when it is possible to go from one to the other with one move allowed by the game.

b) Find in this digraph a set N of vertices, called the *kernel*, which has the following properties: for any vertex $x \notin N$, there is an arc joining x to a vertex $y \in N$, arc entering into N, and any arc exiting from a vertex of N has its head outside N. Hint: you will take in N the vertex representing the final state (0-0) of the game.

c) Use the kernel to define a winning strategy (always play by going *into* the kernel).

d) Try to generalize the preceding idea for a winning strategy of any game represented by a digraph which admits a kernel. You will first define the way to play on the given digraph by inspiring yourself from the preceding particular case.

+5.5. a) Show that the previsit arcs of an extended depth-first search of a strict digraph G define a directed forest in G.

b) Try to list the different cases of arcs of revisit which can be found, in particular for the preceding forest (there are three different cases).

Appendix B

Bases of Complexity Theory

B.1 The concept of complexity

In concrete terms, the concept of time complexity corresponds to the time required to run a program. It is an essential practical parameter in numerous applications. This running time depends on many different factors, first of all on the size of the data to be processed. It is obvious that it will not take the same time to process a graph with 100 vertices as one with 1000. The running time has to be considered in relation to the size of the case dealt with. We talk of the *complexity function*.

Another equally essential factor is the power of the computer used for processing. Again, differences may be great, therefore a reference machine has to be specified as we will see. There are other factors, less obvious but as important, such as the manner in which the data are represented. This representation may be more or less efficient with regard to processing.

To speak of complexity means to speak of concepts which at first may seem clear intuitively but which in fact need to be specified and formalized. Let us start with the concept of an algorithm. A complete formalization of this concept requires the definition of a machine in the sense of a model capable of an automatic process. The model which is the principal reference is the *Turing machine*, which bears the name of its inventor in the 1930s. This theoretical "machine" is, as a machine, reduced to its simplest expression, and that is precisely what makes it useful from a theoretical point of view. Also, despite its extreme simplicity, it seems to contain all the calculation possibilities of the most evolved computers. We will not develop

this theoretical model here; it is enough to know that it exists and makes it possible to formalize the concept of algorithms.

A machine, no matter which one, processes data which have to be presented to it in a certain manner. For example, graphs may be modeled in different ways. We speak of data *encoding*, which is also a concept which has to be specified because it has a direct influence on the complexity of the processing. Let us consider a simple case with the classic algorithm used to decide if a given integer n is a prime number, meaning that it has no other divisor than 1 and itself. A classic method is to try as divisors all the integers less than or equal to $\sqrt{n}$. The number k of divisions to be done, equal to the integral part of $\sqrt{n}$, is a measure of the complexity of this test since it is clear that the time necessary will be proportional to this number, while considering, nevertheless, the division as an elementary operation of constant time. We may consider *a priori* that this complexity is reasonable since it is simply proportional to the square root of the integer. In fact the value of k is exponential in relation to an usual encoding of integer n. Indeed, in any number system, decimal or binary for example, n is represented by a number of digits proportional to $\log n$ (log being in the base considered), and $\sqrt{n}$ is expressed exponentially in function of $\log n$. In base 10, for example, $\sqrt{n} = 10^{\frac{1}{2}t}$, where $t = \log_{10} n$ is the size of the representation of integer n. To take n as the reference size of the data n corresponds to what is called the unary representation, encoding which consists of using only one digit (a "stick", for example, as in primary school), and in which each integer is written by repeating this digit as often as the value of the number. This manner of proceeding is not adequate in relation to complexity. This encoding is therefore considered "unreasonable". We will suppose implicitly that, in the development which will follow, the encodings used are "reasonable", which is necessary for a realistic complexity concept.

From a general algorithmic perspective, running time is measured by the number of operations which are said to be *elementary*. But this concept of elementary operation depends on the operating level from which we operate, and, above all, on the nature of the problem and the calculation. This can be arithmetic operations, for a sorting algorithm it will involve comparisons and exchanges of elements, and for the processing of graphs visits and specific operations on the vertices. We introduce the general concept of elementary operations *pertinent* for the problem under consideration. These are the operations which are directly involved in seeking the result.

Size of problem	Complexity function		
	n	n^2	2^n
10	0.01 μs	0.1 μs	1.024 μs
20	0.02 μs	0.4 μs	1.049 ms
30	0.03 μs	0.9 μs	1.074 s
40	0.04 μs	1.6 μs	18.3 minutes
50	0.05 μs	2.5 μs	13.0 days
60	0.06 μs	3.6 μs	36.6 years
70	0.07 μs	4.9 μs	374 centuries

Table B.1. *Comparison of complexity functions*

However, we may then worry about the possibly arbitrary nature of this concept, and, even more, about the imprecision which results from neglecting other more elementary operations, or even at the lowest level, "machine operations". In fact, this has no impact on the complexity classes which are defined. Indeed, it is always possible to consider that an operation at a higher level is equivalent to a bounded number of lower level machine operations.

B.2 Class P

After all these specifications, we come to what is at the heart of the complexity theory, that is the *polynomiality* criterion. When the size n of the datum increases, there is a great difference theoretically, but also practically in most cases, between the growth of a complexity function which would be of the order of a polynomial function and an exponential growth. Let us consider for example Table B.1, which gives the running time requested for a datum of size $n = 10, 20, \ldots, 70$, with the number of operations expressed by different complexity functions. Here the hypothesis is of a machine which runs 10^9 operations per second, that is, a billion operations per second, which is already a respectable speed (abbreviations used: s for second, ms for 10^{-3} second and μs for 10^{-6} second; the other time units are spelled out). We see in this table that the running time remains reasonable with polynomial functions n and n^2. However, they no longer remain reasonable with an exponential function such an 2^n as soon as the size n becomes a little large (although still very modest in practice; here we will stop at $n = 70$: the time obtained, 374 centuries, is sufficient explanation!). Asymptotically,

it is well known that an exponential function (with a positive base) has a faster growth than any power function.

There is another way to look at things, maybe more explicit. Given a computation time available on a first machine, and an equal time on a second machine 10 times faster, how much bigger is the datum we can process with the second machine relative to the first one? Specifically, let n be the size of the problem which can be processed by the first machine in the given time, and let n' be the size of the problem which can be processed on the second machine for the same available time. Starting first with a complexity function equal to n^2, we have:

$$n'^2 = 10n^2$$

and we deduce:

$$n' = \sqrt{10}n \simeq 3,16n$$

Thus, with the machine which is 10 times faster we can in the same given time process problems which are three times larger in size, which is interesting. As we are going to see, the situation is quite different with an exponential complexity, for example 2^n. We then have:

$$2^{n'} = 10 \times 2^n$$

which gives:

$$n' = n + \log_2 10 \simeq n + 3,3$$

A machine which is 10 times more powerful can only process data of a few additional size units. For an initial datum of size 1000 for example, the gain is not significant.

This polynomial growth criterion of the complexity function was therefore introduced, in particular by J. Edmonds in 1965. This criterion quickly turns out to be pertinent, not just because of its asymptotic nature which we mentioned. There is also the stability of this criterion relative to the composition of algorithms, because of the fact that the composition of polynomial functions is itself a polynomial function. In addition, there is the fact, previously mentioned, that reasonable data encodings only differ from one another polynomially, meaning that any encoding may be upper bounded in size by a polynomial in function of another encoding.

We are used to expressing the complexity of an algorithm with the classic *notation of Landau*: an algorithm is of complexity $O(f(n))$ when the number of elementary operations in relation to the size n of the data is upper bounded by $f(n)$ multiplied by a constant, as soon as the integer n is greater than a certain value. If function f is polynomial, the algorithm is called *polynomial* and the problem dealt with by this algorithm is also called *polynomial*. These problems define the complexity class denoted **P**. Since a polynomial function behaves like its terms of higher degree, and taking into account the constant involved in O, we replace $f(n)$ by an expression of the form $O(n^k)$. Thus we commonly write a complexity in the form $O(n^k)$. The particular case $k = 1$ is in principle the best possible *a priori*, since we need to count at least the time to read the data. This is the case of *linear* algorithms.

Concerning other cases, practice shows that we rarely go over small values of exponent k: $2, 3, 4 \ldots$. This fact reinforces once again the interest of the polynomial criterion: indeed we did mention that an algorithm which was polynomial but with a very high exponent k, for example 10^{50}, would be without practical interest, which is obvious. On the other hand, an algorithm of exponential complexity but with a very low coefficient in front of the exponent, for example $2^{10^{-50}n}$, would not be so bad. We must also say that what is under consideration here is what we call complexity *in the worst case*. This means that we upper bound all data cases uniformly. Yet, it may happen *a priori* that most cases only require a reasonable time, contrary to a few cases which are rare or artificial and are sometimes called "pathological". For such a case, and in general, it seems more natural to evaluate what is called the complexity *in the expected case*, which takes into account appearance rates of each data case, that is the probability with which each possible instance is likely to occur as an input. However, such a measure of complexity is often delicate to calculate, if only because of the fact that it means knowing the input distribution.

A finer analysis of complexity leads us to consider mathematical functions with intermediary growth between the integer-power functions. Thus we often consider function n^q, where q is a real number (not necessarily an integer), $\log^k n$, where k is an integer. (We do not specify the base of this logarithm but we can always consider it to be base 2. It will not change the result expressed asymptotically in O because, as we know, all systems of logarithms are proportional.) To give an example, which does not come from graph algorithms but is sometimes useful, the best sorting algorithms are of complexity $O(n \log n)$, which is better, for example, than $O(n^2)$.

B.3 Class NP

When a problem is recognized as class **P**, it is therefore considered as satisfactorily solved algorithmically. This is the case for numerous basic graph problems such as the search for connected components. However, other problems, apparently simple, at least to set, cannot be solved polynomially.

This is the case with the problem of graph isomorphism. Given two simple graphs G and H, is there an isomorphism of G to H, that is a bijection from vertex set of G onto vertex set of H preserving the neighboring relationship defined by the edge? Let us specify that such a problem with a "yes" or "no" answer is called a *decision problem* and that here we are only considering this type of problem. One way, of course non-polynomial in complexity, is to try the $n!$ possible bijections, n being the common number of vertices of the graphs considered, until finding one which respects the condition. If all bijections have been considered and none is appropriate, then it is possible to answer "no". Let us note an important fact here: given a bijection on the vertices, it is possible to verify polynomially that it does (or does not) define an isomorphism, a complexity $O(n)$ algorithm being easy to imagine for this check. If we have such a bijection, we can say that we can verify polynomially the answer "yes", and the bijection plays the role of a "certificate" for this answer, in the sense that we can be assured of a positive response. If, on the other hand, the answer is negative, then such a certificate does not exist.

We have here, therefore, a problem for which we can verify polynomially, thanks to the certificate, a positive answer, even when this certificate itself cannot be found polynomially. This is the idea which presides over the definition of class **NP**: problems for which it is possible to check a positive answer polynomially without necessarily being able to *find polynomially* this answer. The acronym **NP** does not mean "non-polynomial" but "non-*deterministic* polynomial": the non-determinism here represents our incapacity (which may be temporary) to find directly, that is without the help of a certificate, the right answer. This idea, a bit disconcerting for beginners, is clearly formalized by the non-deterministic Turing machine, but that goes beyond this simple introduction to complexity.

The concept of a certificate, more intuitive than the non-deterministic one, gives a good overview of the concept of class **NP** provided it is formalized. In the following case, we will call an *instance* of a problem a data case for that problem, for example a pair (G, H) of graphs for the

isomorphism problem of two graphs. Therefore it is said that a decision problem Π is in class **NP** if there is a polynomial algorithm $\mathcal{A}$ and a polynomial p such that for any instance x of $\mathcal{A}$ the answer is "yes" if and only if there is a datum y such that $|y| \leq p(|x|)$ and algorithm $\mathcal{A}$ applied to x associated with y gives the answer "yes" ($|x|$ designates the size of x, likewise for $|y|$). The algorithm $\mathcal{A}$ is the *checking-algorithm* and y is a *certificate*, for instance x of Π. Let us note the obligation for the certificate to be of polynomial size in relation to the size of the instance; this is an indispensable condition for a polynomial time check. The certificate is said to be *succinct*.

We can immediately verify the inclusion of class **P** into class **NP**: a polynomial algorithm is also a checking algorithm of a class **P** problem, with an empty certificate. On the contrary, one may think that this inclusion is strict, that is that $\mathbf{P} \neq \mathbf{NP}$, taking into account the lower requirement that represents the simple checking of a certificate compared to that of finding a certificate. It is possible, nevertheless, as no one so far has been able to prove or disprove it.[1]

B.4 NP-complete problems

Naturally, from the perspective of the question $\mathbf{P} \neq \mathbf{NP}$, research has been conducted to better understand class **NP**, in particular to spot the most difficult problems of this class in order to attempt to "capture" what creates the intrinsic difficulty of problems of this class.

The comparison tool here is the *polynomial reduction*. A problem π_1 can be polynomially reduced to a problem π_2 if there is a polynomial algorithm, called a *reduction-algorithm* from π_1 to π_2, which calculates for each instance x_1 of π_1, an instance x_2 of π_2 such that the answer for x_1 is "yes" if and only if the answer for x_2 is "yes" (x_1 and x_2 have the same answer). Thus, if we have a polynomial algorithm solving π_2, we can deduce a polynomial algorithm solving π_1, by composing the reduction-algorithm form π_1 to π_2 and the solving-algorithm for π_2 (note the advantage of being able to compose polynomial algorithms). In other words, if π_2 is in class **P**, so is π_1. Again, π_2 is at least as difficult as π_1 and is thus possibly more representative of the difficulty of class **NP**. The final interest of all of this is to put in evidence a problem at least as difficult as all the others in class **NP**, that is

[1] There is a one million dollar reward offered by an American patron for whoever solves the problem. Go for it!

a problem to which all others can be reduced polynomially. A "universal" problem, in a way, for class **NP** is called **NP**-*complete*. Such a problem can be formally put in evidence with the non-deterministic Turing machine mentioned above. However, it is more interesting to know that there are such problems which can be put naturally. The first found, at the beginning of the 1970s, is a problem of logic, called a *satisfiability problem* (denoted SAT) which we will now describe.

The data are: n Boolean variables $x_1, \ldots, x_n$ taking the value *true* or *false*, m Boolean expressions $C_1, \ldots, C_m$ called *clauses* and expressed in a disjunctive form, that is:

$$C_j = y_1 \vee \cdots \vee y_{k_j}$$

where each y_i is equal to x_l or $\neg x_l$, where $l \in \{1, \ldots, n\}$ (remember the classic logic operators: *or* denoted $\vee$, *no* denoted $\neg$). The question then is: is their an assignment of values to variable x_i such that each clause C_j takes the value *true* (that is it has at least one y_i which has the value *true*)? Let us note that this problem is clearly in class **NP**: such an assignment is in fact an appropriate certificate. Cook–Levin's theorem states that this problem is **NP**-complete. Historically, SAT has been the first natural problem found in class **NP**, but there have been numerous others found since, in particular concerning graphs.

B.5 Classification of problems

It is impossible to speak of the bases of complexity theory without mentioning the class **coNP** and the concept of a "well-characterized" property. The definition of this class is based on the dissymmetry which exists between the answers "yes" and "no" in the problems of class **NP**. This dissymmetry does not appear in class **P**: the answer "no" is automatic if it is not the answer "yes". For a problem of class **NP**, the answer "no" may not have an obvious succinct certificate. For example, for the problem of graph isomorphism, what can be a certificate for the answer "no"? Another example: for the problem of the existence of a Hamilton cycle in a graph, if it is easy to see how to "certify" the answer "yes", simply by giving a Hamilton cycle, we do not see directly how to certify the answer "no".

Class **coNP** is therefore defined as the class of decision problems for which the "complement" problem, that is the problem set in order to invert the answers "yes" and "no", is in **NP**. We do not know if **NP** = **coNP** and, in the hypothesis of an inequality, the intersection of these

two classes is interesting to consider: it represents the problems based on a *well-characterized* property in the sense that the answer "yes" as well as the answer "no" can be certified by a succinct certificate. This is a typical case, for example, with the problem of recognition of planar graphs: the answer "yes" can be certified by a planar embedding of the graph, while the answer "no" can be certified by the presence in the graph of an excluded configuration (by application of Kuratowski's theorem, see Chapter 11). However, the problem of the recognition of planar graphs is in fact in **P**; it is therefore not surprising that it belongs to this intersection since we have in general $\mathbf{P} \subseteq \mathbf{NP} \cap \mathbf{coNP}$. Another basic question of complexity theory is to know if we have equality in this inclusion. Very few problems are known in this intersection without also being known in **P**.

On the whole, we have a possible configuration of the previous classes as shown in the diagram in Figure B.1. Some people think this is probable. If it is not like this, then things are very different from what they are thought to be today ....

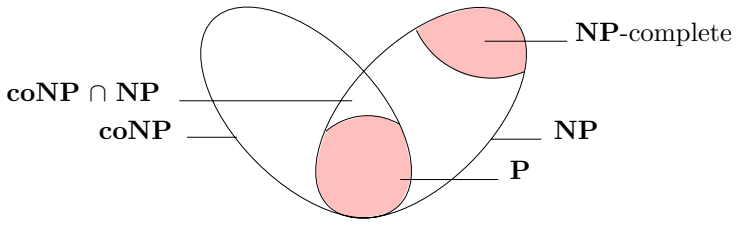

Figure B.1. *Classes of complexity*

Let us specify, finally, that an *optimization* problem is different from a *decision* problem. Let us take for example the traveling salesman problem (Chapter 10), which can be stated as follows: find in a weighted graph a Hamilton cycle minimum for the sum of the values of its edges. We associate it with the following decision problem: given an integer k, is there a Hamilton cycle of length $\leq k$? Obviously a solution to the optimization problem gives a solution to the decision problem, simply by comparing the value of a minimum cycle with k. However, what is more interesting is that the converse is true. It is less obvious; it can be seen by bounding by successive dichotomies the value of a minimum cycle. A problem for which the associated decision problem is **NP**-complete is sometimes called **NP**-*difficult* rather than **NP**-complete.

We again find in an interesting way the idea of a well-characterized property in certain optimization problems, typically the maximum flow problem in a transportation network and that of the minimum cut (Chapter 8). At the optimum, we have an equality which certifies one as maximum and the other as minimum.

B.6 Other approaches to difficult problems

Once it has been admitted that some problems can probably not be practically solved in a reasonable manner, that is by a polynomial algorithm (in the hypothesis $\mathbf{P} \neq \mathbf{NP}$), other approaches to **NP**-complete problems have to be contemplated.

A first, natural, approach is to try to obtain in polynomial time an approximate solution with a precision which has to be given. Problems are not all equal when it comes to this. For some it is possible to find good approximations, others are resistant to any reasonable results, such as, for example, the traveling salesman problem for which one shows that, except if $\mathbf{P} = \mathbf{NP}$, there is no satisfactory approximate polynomial algorithm to solve it (see Chapter 10). There are many more approaches, not studied here, and we refer the reader to the specialized literature on this subject.[2]

[2]For example: C.H. Papadimitriou, *Computational Complexity*, Addison-Wesley (1994); J. Stern, *Fondements mathématiques de l'informatique*, McGraw-Hill (1990); M.R. Garey and D.S. Johnson, *Computers and Intractability*, Freeman (1979), a great classic reference in the field.