Chapter 3

Greedy algorithms

This chapter explains the reasoning in finding optimal greedy algorithms. The main feature of a greedy algorithm is that it builds the solution step by step, and, at each step, it makes a decision that is locally optimal. Throughout Sections 3.1 to 3.3, we illustrate this principle with several examples, and also outline situations where greedy algorithms are not optimal; taking a good local decision may prove a bad choice in the end! In Section 3.4, we also cover matroids, a (mostly theoretical) framework to prove the optimality of greedy algorithms. All of these techniques are then illustrated with a set of exercises in Section 3.5, with solutions found in Section 3.6.

3.1 Motivating example: The sports hall

Problem. Let us consider a sports hall in which several events should be scheduled. The goal is to have as many events as possible, given that two events cannot occur simultaneously (only one hall). Each event i is characterized by its starting time s_i and its ending time e_i. Two events are compatible if their time intervals do not overlap. We would like to solve the problem, i.e., find the maximum number of events that can fit in the sports hall, with a greedy algorithm.

A first greedy algorithm. The first idea consists in sorting events by increasing durations $e_i - d_i$. At each step, we schedule an event into the sports hall if it fits, i.e., if it is compatible with events that have already been scheduled. The idea is that we will be able to accommodate more shorter events than longer ones. However, we make local decisions at each step of the algorithm (this is a greedy algorithm!), and it turns out that we can make decisions that do not lead to the optimal solution. For instance, in the example of Figure 3.1, the greedy algorithm schedules only the shortest event i, while the two compatible events j and k would lead to a better solution.

A second greedy algorithm. In order to avoid the problem encountered in the previous example, we design a new algorithm that sorts events by starting

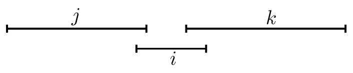


FIGURE 3.1: The first greedy algorithm is not optimal.

times s_i, and then proceeds similarly to the first greedy algorithm. In the example of Figure 3.1, this greedy algorithm returns the optimal solution. However, the local decisions that are made may not be the optimal ones, as shown in the example of Figure 3.2. Indeed, the algorithm schedules event i at the first step, and then no other event can be scheduled, while it would be possible to have eight compatible events. Note that the first greedy algorithm would return the optimal solution for this example.

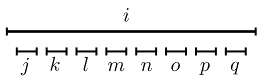


FIGURE 3.2: The second greedy algorithm is not optimal.

A third greedy algorithm. Building upon the first two algorithms, we observe that it is always a good idea to first select events that do not intersect with many other events. In the first example, events j and k intersect with only one other event, while event i intersects with two events and is chosen later; therefore, the new algorithm finds the optimal solution. Similarly in the second example, event i intersects eight other events and it is the only event not to be scheduled. However, this greedy algorithm is still not optimal. We can build an example in which we force the algorithm to make a bad local decision. In the example of Figure 3.3, event i is the first to be chosen because it has the smallest number of intersecting events. However, if we schedule i, we can have only three compatible events, while we could have a solution with four compatible events, j, k, l, and m.

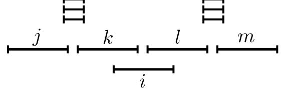


FIGURE 3.3: The third greedy algorithm is not optimal.

An optimal greedy algorithm. Even though many greedy choices do not lead to an optimal solution, as observed with the preceding algorithms, there is a greedy algorithm that solves the sports hall problem in polynomial time. The idea is to sort the events by increasing ending times e_i, and then to greedily schedule the events. This way, at each step we fit the maximum number of events up to a given time, and we never make a bad decision. We now prove the optimality of this algorithm.

Let f_1 be the event with the smallest ending time. We prove first that there exists an optimal solution that schedules this event. Let us consider an optimal solution $O = \{f_{i_1}, f_{i_2}, \ldots, f_{i_k}\}$, where k is the maximum number of events that can be scheduled in the sports hall, and where events are sorted by nondecreasing ending times. There are two possible cases: either (i) $f_{i_1} = f_1$, the optimal solution schedules f_1, and nothing needs to be done, or (ii) $f_{i_1} \neq f_1$. In this second case, we replace f_{i_1} with f_1 in solution O. We have $e_1 \leqslant e_{i_1}$ by definition of event f_1, and $e_{i_1} \leqslant s_{i_2}$ because O is a solution to the problem (f_{i_1} and f_{i_2} are compatible). Therefore, $e_1 \leqslant s_{i_2}$ and, thus, f_{i_2} is compatible with f_1. The new solution is still optimal (the number of events remain unchanged), and event f_1 is scheduled.

The proof works by induction, following the previous reasoning. Once f_1 is scheduled, we consider only events that do not intersect with f_1, and we iterate the reasoning on the remaining events to conclude the proof.

Finally, we emphasize that there can be many optimal solutions, and not all of them will include the first event f_1 selected by the greedy algorithm, namely the event with the smallest end time. However, schedules that select f_1 are *dominant*, meaning that there exists an optimal solution that includes f_1.

3.2 Designing greedy algorithms

The example of the sports hall gives a good introduction to the design principles of greedy algorithms. Actually, the binary method to compute x^n in Section 1.1.2 also is a greedy algorithm, in which we decide at each step which computation to perform. We can formalize the reasoning to find greedy algorithms as follows:

1. Decide on a greedy choice that allows us to locally optimize the problem;

2. Search for a counterexample that shows that the algorithm is not optimal (and go back to step 1 if a counterexample is found), or prove its optimality through steps 3 and 4;

3. Show that there is always an optimal solution that performs the greedy choice of step 1;

4. Show that if we combine the greedy choice with an optimal solution of the subproblem that we still need to solve, then we obtain an optimal solution.

We say that a greedy algorithm is a *top-down* algorithm, because at each step we make a local choice, and we then have a single subproblem to solve, given this choice. On the contrary, we will see in Section 4 that dynamic programming algorithms are *bottom-up*; we will need results of multiple subproblems to make a choice.

3.3 Graph coloring

In this section, we further illustrate the principle of greedy algorithms through the example of graph coloring. The problem consists in coloring all vertices of a graph using the minimum number of colors, while enforcing that two vertices, which are connected with an edge, are not of the same color. Formally, let $G = (V, E)$ be a graph and $c : V \to \{1..K\}$ be a K-coloring such that $(x, y) \in E \Rightarrow c(x) \neq c(y)$. The objective is to minimize K, the number of colors.

3.3.1 On coloring bipartite graphs

We start with a small theorem that allows us to define a bipartite graph, defined as a graph that can be colored with only two colors.

THEOREM 3.1. *A graph can be colored with two colors if and only if all its cycles are of even length.*

Proof. Let us first consider a graph G that can be colored with two colors. Let $c(v) \in \{1, 2\}$ be the color of vertex v. We prove by contradiction that all cycles are of even length. Indeed, if G has a cycle of length $2k+1$, $v_1, v_2, \ldots, v_{2k+1}$, then we have $c(v_1) = 1$, say, which implies that $c(v_2) = 2$, $c(v_3) = 1$, until $c(v_{2k+1}) = 1$. However, since it is a cycle, there is an edge between v_1 and v_{2k+1}, so they cannot be of the same color, which leads to the contradiction.

Now, if all the cycles of the graph G are of even length, we search for a 2-coloring of this graph. We assume that G is connected (the problem is independent from one connected component to another). The idea consists in performing a breadth-first traversal of G.

Le $x_0 \in G$, $X_0 = \{x_0\}$ and $X_{n+1} = \bigcup_{y \in X_n} N(y)$, where $N(y)$ is the set of nodes connected to y, but not yet included in a set X_k, for $k \leqslant n$. Each vertex appears in one single set, and we color with color 1 the elements from sets X_{2k}, and with color 2 the elements from sets X_{2k+1}.

This 2-coloring is valid if and only if two vertices connected by an edge are of different colors. If there is an edge between $y \in X_i$ and $z \in X_j$, where i and j are both either even or odd, then we have a cycle $x_0, \ldots, y, z, \ldots, x_0$ of length $i + j + 1$, and this value is even, leading to a contradiction. The coloring, therefore, is valid, which concludes the proof. □

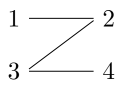
In a bipartite graph, if we partition vertices into two sets according to the colors, all edges go from one set to the other. We retrieve here the usual definition of bipartite graphs, namely graphs whose vertices are partitioned into two sets and with no edge inside these sets. We now consider colorings of general graphs, and we propose a few greedy algorithms to solve the problem.

3.3.2 Greedy algorithms to color general graphs

The first greedy algorithm takes the vertices in a random order, and, for each vertex v, it colors it with the smallest color number that has not been yet given to a neighbor of v, i.e., a node connected to v.

Let $K_{greedy1}$ be the total number of colors needed by this greedy algorithm. Then we have $K_{greedy1} \leqslant \Delta(G) + 1$, where $\Delta(G)$ is the maximal degree of a vertex (number of edges of the vertex). Indeed, at any step of the algorithm, when we color vertex v, it has at most $\Delta(G)$ neighboring vertices and, therefore, the greedy algorithm never needs to use more than $\Delta(G)+1$ colors.

Note that this algorithm is optimal for a fully connected graph (a clique), since we need $\Delta(G)+1$ colors to connect such a graph (one color per vertex). However, this algorithm is not optimal in general; on the following bipartite graph, if the order of coloring is 1 and then 4, we need three colors, while the optimal coloring uses only two colors.

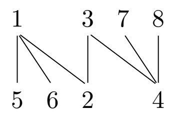


In order to improve the previous algorithm, one idea consists in ordering vertices in a smart way, and then in proceeding as before, i.e., color each vertex in turn with the smallest possible color.

Let $n = |V|$ be the number of vertices, and d_i be the degree of vertex v_i. We have $K_{greedy2} \leqslant \max_{1 \leqslant i \leqslant n} \min(d_i+1, i)$. Indeed, when we color vertex v_i, it has at most $\min(d_i, i-1)$ neighbors that have already been colored, and thus its own color is at most $1 + \min(d_i, i-1) = \min(d_i+1, i)$. To obtain the result, we take the maximum of these values on all vertices.

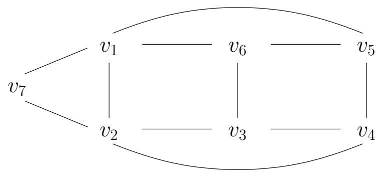
This result suggests that it would be smart to first color vertices with a high degree, so that we have $\min(d_i + 1, i) = i$. Therefore, the second greedy algorithm sorts the vertices by nonincreasing degrees.

Once again, the algorithm is not optimal. On the following bipartite graph, we choose to color vertex 1, then vertex 4, which imposes the use of three colors instead of the two required ones.



Based on these ideas, several greedy algorithms can be designed. In particular, a rather intuitive idea consists in giving priority to coloring vertices that have already many colored neighbors. We define the color-degree of a vertex as the number of its neighbors that are already colored. Initially, the color-degree of each vertex is set to 0, and then it is updated at each step of the greedy algorithm.

The following greedy algorithm is called the *Dsatur* algorithm in [20]. The ordering is done by (color-degree, degree); we choose a vertex v with maximum color-degree, and such that its degree is the largest among the vertices with maximum color-degree. This vertex v is then colored with the smallest possible color, and the color-degrees of the neighbors of v are updated before proceeding to the next step of the algorithm. We illustrate this algorithm on the following example:

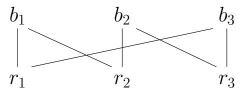


We first choose a vertex with maximum degree, for instance v_1, and it is colored with color 1. The color-degree of v_2, v_5, v_6, and v_7 becomes 1, and we choose v_2, which has the maximum degree (between these four vertices); it is assigned color 2. Now, v_7 is the only vertex with color-degree 2; it is given the color 3. All remaining noncolored vertices have the same color-degree 1 and the same degree 3, we arbitrarily choose v_3 and color it with 1. Then, v_4, with color-degree 2, receives color 3. Finally, v_5 is colored in 2 and v_6 in 3; the graph is 3-colored, and it is an optimal coloring.

The name *Dsatur* comes from the fact that maximum color-degree vertices are saturated first. We prove below that *Dsatur* always returns an optimal coloring on bipartite graphs; however, it may use more colors than needed on arbitrary graphs.

THEOREM 3.2. *The* Dsatur *algorithm is optimal on bipartite graphs, i.e., it always succeeds to color them with two colors.*

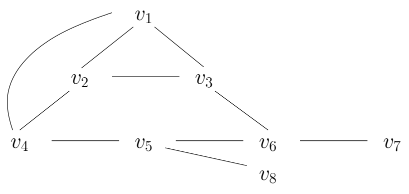
Proof. Consider a connected bipartite graph $G = (V, E)$, where $V = B \cup R$ and each edge in E is connecting a vertex in B (color 1 is blue) and a vertex in R (color 2 is red). Note first that the first two greedy algorithms may fail. Let G be such that $B = \{b_1, b_2, b_3\}$, $R = \{r_1, r_2, r_3\}$, and $E = \{(b_1, r_2), (b_2, r_3), (b_3, r_1), (b_i, r_i) | 1 \leqslant i \leqslant 3\}$, as illustrated below.



All vertices have a degree 2. If we start by coloring a vertex of B, for instance b_1, and then a nonconnected vertex of R, r_3, with the same color 1, it is not possible to complete the coloring with only two colors. The use of the color-degree prevents us from such a mistake, since once b_1 has been colored, we need to color either r_1 or r_2 with the color 2, and finish the coloring optimally.

In the general case, with *Dsatur*, we first color a vertex, for instance from B, with color 1 (blue). Then we have to color a vertex of color-degree 1, that is, a neighboring vertex. This neighboring vertex belongs necessarily to R. It is colored with color 2 (red). We prove by induction that at any step of the algorithm, all colored vertices of B are colored in blue, and all vertices of R are colored in red. Indeed, if the coloring satisfies this property at a given step of the algorithm, we choose next a vertex v with nonnul color-degree. Because the graph is bipartite, all its neighbors are in the same set and have the same color: red if $v \in B$, or blue if $v \in R$. Vertex v, therefore, is colored in red if it is in R, or in blue if it is in B. □

We exhibit a counterexample to show that *Dsatur* is not optimal on arbitrary graphs.



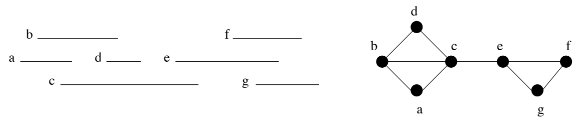
Dsatur can choose v_4 first, because it has the maximum degree 3; it is colored with 1. Between the vertices with color-degree 1, the algorithm can (arbitrarily) choose v_5, which is colored with 2. Then the algorithm can choose to color v_6, using color 1. Then, v_1 is chosen between vertices of color-degree 1 and degree 3, and it is colored with 2. We finally need to use colors 3 and 4 for v_2 and v_3, while this graph could have been colored with only three colors (v_1, v_5, v_7 with color 1, v_2, v_6, v_8 with color 2, and v_3, v_4 with color 3).

To build this counterexample, we force *Dsatur* to make a wrong decision, by coloring both v_4 and v_6 with color 1, and v_1 with color 2, which forces four colors because of v_2 and v_3. Note that it would be easy to built an example without any tie (thereby avoiding random choices) by increasing the degree of some vertices (for instance, in the example, v_7 and v_8 are just there to increase the degrees of v_5 and v_6).

The problem of coloring general graphs is NP-complete, as will be shown in Chapter 7. However, for a particular class of graphs, a smart greedy algorithm can return the optimal solution, as we detail below.

3.3.3 Coloring interval graphs

We focus now on interval graphs. Given a set of intervals, we define a graph whose vertices are intervals, and whose edges connect intersecting intervals. The following example shows such a graph, obtained with a set of seven intervals.



The problem of coloring such a graph is quite similar to the sports hall problem. Indeed, one can see each interval as representing an event, with its starting and ending time, and the color as representing a sports hall. Then, only compatible events will be colored with the same color, and we could use one sports hall per set of compatible events. If we minimize the number of colors, we minimize the number of sports halls that are needed to organize all events.

Graphs that are obtained from a set of intervals are called interval graphs. We define the following greedy algorithm: intervals (i.e., vertices) are sorted by nondecreasing starting times (or left extremity). In the example, the order is a, b, c, d, e, f, g. Then, the greedy coloring is done as before; for each chosen vertex, we color it with the smallest compatible color. On the example, we obtain the coloring $1, 2, 3, 1, 1, 2, 3$, which is optimal, as the graph contains a cycle of length 3.

We prove now that this greedy algorithm is optimal for any interval graph. Let G be such a graph, and let d_v be the starting time of interval v corresponding to vertex v. We execute the greedy algorithm; it uses k colors. If vertex v receives color k, then this means that $k-1$ intervals that start no later than d_v intersect this interval and had all been colored with colors 1 to $k-1$; otherwise, v would be colored with a color $c \leqslant k-1$. All of these intervals are thus intersecting, because they all contain the point d_v; therefore, graph G contains a clique of size k. Since all vertices of a clique must be colored with distinct colors, we cannot color the graph with less than k colors. The greedy algorithm, therefore, is optimal.

Once again, we point out that the order chosen by the greedy algorithm is vital, since we could force the greedy algorithm to make a wrong decision, even on a bipartite graph as below, if we would not proceed from left to right. We could first color a, then d, leading to the use of three colors instead of two.